Exploiting data locality in cache-coherent NUMA systems

Isaac Sánchez Barrera
Barcelona, 2022

Advisors:
Marc Casas Guix
Department of Computer Sciences
Barcelona Supercomputing Center

Miquel Moretó Planas
Department of Computer Architecture
Universitat Politècnica de Catalunya

Department of Computer Sciences
Barcelona Supercomputing Center

A thesis submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy

in the Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

A mis abuelos,
Adela, Alonso, Manuel y Mercedes.

Abstract

The end of Dennard scaling has caused a stagnation of the clock frequency in computers. To overcome this issue, in the last two decades vendors have been integrating larger numbers of processing elements in the systems, interconnecting many nodes, including multiple chips in the nodes and increasing the number of cores in each chip. The speed of main memory has not evolved at the same rate as processors, it is much slower and there is a need to provide more total bandwidth to the processors, especially with the increase in the number of cores and chips.

Still keeping a shared address space, where all processors can access the whole memory, solutions have come by integrating more memories: by using newer technologies like high-bandwidth memories (HBM) and non-volatile memories (NVM), by giving groups cores (like sockets, for example) faster access to some subset of the DRAM, or by combining many of these solutions. This has caused some heterogeneity in the access speed to main memory, depending on the CPU requesting access to a memory address and the actual physical location of that address, causing non-uniform memory access (NUMA) behaviours. Moreover, many of these systems are cache-coherent (ccNUMA), meaning that changes in the memory done from one CPU must be visible by the other CPUs and transparent for the programmer.

These NUMA behaviours reduce the performance of applications and can pose a challenge to the programmers. To tackle this issue, this thesis proposes solutions, at the software and hardware levels, to improve the data locality in NUMA systems and, therefore, the performance of applications in these computer systems.

The first contribution shows how considering hardware prefetching simultaneously with thread and data placement in NUMA systems can find configurations with better performance than considering these aspects separately. The performance results combined with performance counters are then used to build a performance model to predict, both offline and online, the best configuration for new applications not in the model. The evaluation is done using two different high performance NUMA systems, and the

performance counters collected in one machine are used to predict the best configurations in the other machine.

The second contribution builds on the idea that prefetching can have a strong effect in NUMA systems and proposes a NUMA-aware hardware prefetching scheme. This scheme is generic and can be applied to multiple hardware prefetchers with a low hardware cost but giving very good results. The evaluation is done using a cycle-accurate architectural simulator and provides detailed results of the performance, the data transfer reduction and the energy costs.

Finally, the third and last contribution consists in scheduling algorithms for task-based programming models. These programming models help improve the programmability of applications in parallel systems and also provide useful information to the underlying runtime system. This information is used to build a task dependency graph (TDG), a directed acyclic graph that models the application where the nodes are sequential pieces of code known as tasks and the edges are the data dependencies between the different tasks. The proposed scheduling algorithms use graph partitioning techniques and provide a scheduling for the tasks in the TDG that minimises the data transfers between the different NUMA regions of the system. The results have been evaluated in real ccNUMA systems with multiple NUMA regions.

Resum

La fi de la llei de Dennard ha provocat un estancament de la freqüència de rellotge dels computadors. Amb l'objectiu de superar aquest fet, durant les darreres dues dècades els fabricants han integrat més quantitat d'unitats de còmput als sistemes mitjançant la interconnexió de nodes diferents, la inclusió de múltiples xips als nodes i l'increment de nuclis de processador a cada xip. La rapidesa de la memòria principal no ha evolucionat amb el mateix factor que els processadors; és molt més lenta i hi ha la necessitat de proporcionar més ample de banda als processadors, especialment amb l'increment del nombre de nuclis i xips.

Tot mantenint un sistema d'adreçament compartit en el qual tots els processadors poden accedir a la memòria sencera, les solucions han estat al voltant de la integració de més memòries: usant tecnologies modernes com les memòries d'alt ample de banda (*high-bandwidth memories*, HBM) i memories no volàtils (*non-volatile memories*, NVM), fent que grups de nuclis (com sòcols sencers) tinguin accés més ràpid a un subconjunt de la DRAM o amb la combinació de solucions. Tot això ha provocat una heterogeneïtat en la velocitat d'accés a la memòria principal, en funció del nucli que sol·licita l'accés a una adreça de memòria en particular i la localització física d'aquesta adreça, fet que provoca uns comportaments no uniformes en l'accés a la memòria (*non-uniform memory access*, NUMA). A més, molts d'aquests sistemes tenen memòries cau coherents (*cache-coherent NUMA*, ccNUMA), la qual cosa implica que qualsevol canvi fet a la memòria des d'un nucli d'un processador ha de ser visible pels altres nuclis de manera transparent per als programadors.

Aquests comportaments NUMA redueixen el rendiment de les aplicacions i poden suposar un repte per als programadors. Per abordar el problema en qüestió, a la tesi s'hi proposen solucions, a nivell de programari i maquinari, que milloren la localitat de dades als sistemes NUMA i, en conseqüència, el rendiment de les aplicacions en aquests sistemes informàtics.

La primera contribució mostra que, quan es tenen en compte alhora la precàrrega d'adreces de memòria amb maquinari (*hardware prefetching*) i

les decisions d'ubicació dels fils d'execució i les dades als sistemes NUMA, es poden trobar millors configuracions que quan es condieren ambdós aspectes per separat. Una combinació dels resultats de rendiment i dels comptadors disponibles al sistema s'utilitza per construir un model de rendiment per fer la predicció, tant per avançat com també en temps d'execució, de la millor configuració per aplicacions que no es troben al model. L'avaluació es du a terme mitjançant dos sistemes NUMA d'alt rendiment, i els comptadors mesurats en un sistema s'utilitzen per predir les millors configuracions a l'altre sistema.

La segona contribució es basa en la idea que el *prefetching* pot tenir un efecte considerable als sistemes NUMA i proposa un esquema de precàrrega a nivell de maquinari que té en compte els efectes NUMA. Aquest esquema és genèric i es pot aplicar a diferents algorismes de precàrrega existents amb un cost de maquinari molt baix però amb molt bons resultats. Els resultats s'avaluen amb un simulador arquitectural acurat a nivell de cicle i proporciona resultats detallats del rendiment, la reducció de les comunicacions de dades i els costos energètics.

Per últim, la tercera i darrera contribució consisteix en algorismes de planificació per models de programació basats en tasques. Aquests models simplifiquen la programabilitat de les aplicacions paral·leles i proveeixen informació molt útil al sistema en temps d'execució (*runtime system*) que en controla el funcionament. Aquesta informació s'usa per construir un graf de dependències entre tasques (*task dependency graph*, TDG), un graf dirigit i acíclic que modela l'aplicació i en el qual els nodes són fragments de codi seqüencial, coneguts com a tasques, i els arcs són les dependències de dades entre les diferents tasques. Els algorismes de planificació proposats fan servir tècniques de particionat de grafs i proporcionen una planificació de les tasques del TDG que minimitza la comunicació de dades entre les diferents regions NUMA del sistema. Els resultats han estat avaluats en sistemes ccNUMA reals amb múltiples regions NUMA.

Resumen

El final de la ley de Dennard ha provocado un estancamiento de la frecuencia de reloj de los computadores. Con el objetivo de superar este problema, durante las últimas dos décadas los fabricantes han integrado más unidades de cómputo en los sistemas mediante la interconexión de nodos diferentes, la inclusión de múltiples chips en los nodos y el incremento de núcleos de procesador en cada chip. La rapidez de la memoria principal no ha evolucionado con el mismo factor que los procesadores; es mucho más lenta y hay la necesidad de proporcionar más ancho de banda a los procesadores, especialmente con el incremento del número de núcleos y chips.

Aun manteniendo un sistema de direccionamiento compartido en el que todos los procesadores pueden acceder al conjunto de la memoria, las soluciones han oscilado alrededor de la integración de más memorias: usando tecnologías modernas como las memorias de alto ancho de banda (*high-bandwidth memories*, HBM) y memorias no volátiles (*non-volatile memories*, NVM), haciendo que grupos de núcleos (como zócalos completos) tengan acceso más veloz a un subconjunto de la DRAM, o con la combinación de soluciones. Esto ha provocado una heterogeneidad en la velocidad de acceso a la memoria principal, en función del núcleo que solicita el acceso a una dirección de memoria en particular y la ubicación física de esta dirección, lo que provoca unos comportamientos no uniformes en el acceso a la memoria (*non-uniform memory access*, NUMA). Además, muchos de estos sistemas tienen memorias caché coherentes (*cache-coherent NUMA*, ccNUMA), lo que implica que cualquier cambio hecho en la memoria desde un núcleo de un procesador debe ser visible por el resto de procesadores de forma transparente para los programadores.

Estos comportamientos NUMA reducen el rendimiento de las aplicaciones y pueden suponer un reto para los programadores. Para abordar dicho problema, en esta tesis se proponen soluciones, a nivel de *software* y *hardware*, que mejoran la localidad de datos en los sistemas NUMA y, en consecuencia, el rendimiento de las aplicaciones en estos sistemas informáticos.

La primera contribución muestra que, cuando se tienen en cuenta a la vez la precarga de direcciones de memoria mediante *hardware* (o *hardware prefetching*) y las decisiones de la ubicación de los hilos de ejecución y los datos en los sistemas NUMA, se pueden hallar mejores configuraciones que cuando se consideran ambos aspectos por separado. Con una combinación de los resultados de rendimiento y de los contadores disponibles en el sistema se construye un modelo de rendimiento, tanto por avanzado como en en tiempo de ejecución, de la mejor configuración para aplicaciones que no están incluidas en el modelo. La evaluación se realiza en dos sistemas NUMA de alto rendimiento, y los contadores medidos en uno de los sistemas se usan para predecir las mejores configuraciones en el otro sistema.

La segunda contribución se basa en la idea de que el *prefetching* puede tener un efecto considerable en los sistemas NUMA y propone un esquema de precarga a nivel *hardware* que tiene en cuenta los efectos NUMA. Este esquema es genérico y se puede aplicar a diferentes algoritmos de precarga existentes con un coste de *hardware* muy bajo pero que proporciona muy buenos resultados. Dichos resultados se obtienen y evalúan mediante un simulador arquitectural preciso a nivel de ciclo y proporciona resultados detallados del rendimiento, la reducción de las comunicaciones de datos y los costes energéticos.

Finalmente, la tercera y última contribución consiste en algoritmos de planificación para modelos de programación basados en tareas. Estos modelos simplifican la programabilidad de las aplicaciones paralelas y proveen información muy útil al sistema en tiempo de ejecución (*runtime system*) que controla su funcionamiento. Esta información se utiliza para construir un grafo de dependencias entre tareas (*task dependency graph*, TDG), un grafo dirigido y acíclico que modela la aplicación y en el ue los nodos son fragmentos de código secuencial, conocidos como tareas, y los arcos son las dependencias de datos entre las distintas tareas. Los algoritmos de planificación que se proponen usan técnicas e particionado de grafos y proporcionan una planificación de las tareas del TDG que minimiza la comunicación de datos entre las distintas regiones NUMA del sistema. Los resultados se han evaluado en sistemas ccNUMA reales con múltiples regiones NUMA.

Acknowledgements

Aquesta tesi és el resultat de més de cinc anys de feina, amb moments més tranquils i d'altres més durs. Estic content amb tot el que m'ha aportat fer el doctorat: he après moltes coses noves i he fet noves amistats. També puc dir que estic desencantat amb alguns aspectes del món de la recerca, segurament el tenia massa idealitzat i me l'esperava més centrat en fer avançar el coneixement.

Però no m'extendré més amb lamentacions que ara mateix no solucionen res, perquè no hauria de ser l'objectiu d'aquests paràgrafs. No puc més que començar agraint-vos, Miquel i Marc, que m'hàgiu dirigit la tesi. Hi ha hagut moments en els quals m'heu hagut de perseguir (com perquè acabés d'escriure això) o alguna vegada que ha passat al revés, però no ha estat gaire sovint. I també moments de dubtes existencials i de qüestionar-me que el que intentava fer pogués funcionar o servís de res, però que heu pogut salvar gràcies a la vostra experiència i la confiança que m'heu donat.

Molt importants sou tots els de RoMoL i Til·lers 3. En particular, Adrián, no conozco a nadie más dedicado y trabajador que tú, aunque ya te lo he dicho más de una vez. Atrás quedan nuestros ratos de quejarnos de todo, yo creo que con motivo, y espero que estés disfrutando por Cambridge, que debes tener un tiempo más parecido al de tu tierruca que en Barcelona, aunque te falte el mar.

Vladimir, you are a great man (pun intended). Very few people take so much care in what they do as you do, and I think you deserve the best for being such a nice guy. We should do another group Serbian dinner the next time we meet, I enjoyed it a lot the last time we went to that Serbian bar in Barcelona.

Cristóbal, poca relación tuvimos al principio más allá de que alguna vez usaba tu preciado POWER 8 (y luego usamos aquel POWER 9 que sacaron del clúster para poder hacer cosas raras). Pero tu cambio de oficina en el Nexus II cambió las cosas. ¡Te mando fuerzas para que puedas terminar con lo tuyo! Si yo he podido, tú también.

Guillem, Rubén i Víctor, espero que l'experiència del doctorat us vagi molt bé i la resta de la vida també. Juntament amb el Cristóbal, bons moments vam passar dinant al Vèrtex que amb la pandèmia es van convertir en diumenges jugant al pinturillo, petant la xerrada o comentant pel·lícules a cada qual més estrambòtica. Mai us desfareu dels meus acudits. I espero que el gem5 no us deixi amb mal cos.

Obviamente, Constan, parece que no nos podemos deshacer el uno del otro, pero supongo que será una buena señal. Antes nos veíamos más en la oficina, que estábamos sentados delante uno del otro, a ver si en poco tiempo nos volvemos a ver en la oficina (aunque sea solo algún día a la semana), porque eso significará que la pandemia ha mejorado. Así podremos tener también alguna discusión sobre si hacer algo de cierta manera o de otra tiene sentido, que con una pizarra y de cara es mejor que virtualmente.

De ti no me olvido, Calvin, que aunque quizá no te lo parezca, también me has ayudado mucho en estos años de tesis. A ver si un día nos volvemos a ver en la oficina y, por supuesto, a ver si se da otra ocasión para comer con Vladimir y Constan.

Xubin, it was great having you in the office and I'm happy you are still in Barcelona. I think your work is the one that was the clearest exponent of the runtime-aware architectures. I was happy to see you when you visited my home town, and I must say I walked a lot that day!

Paul and Dimitrios, thank you also for everything that I have learnt from both of you, many things I have built upon have started with some basic stuff you provided me. Luc, I hope you are doing great in France (if you're still there). Your tools for conference dates have been really useful, and your base LaTeXtemplate for the BSC presentations has served me a lot.

Sicong, ens vam conèixer al BSC, vam parlar algun cop i ens vam ajudar en alguna cosa. Ara hem estat treballant uns mesos junts i ha estat un plaer!

Adrià i Lluc, gràcies a vosaltres també perquè, encara que no heu participat directament en la meva tesi, sí que m'heu ajudat, tant amb el gem5 com amb temes més generals de la recerca. Y gracias a ti también, Santi, que aunque hemos coincidido poco tiempo, tu granito de arena me has aportado.

No em vull oblidar de mencionar l'Helena, el Iulian i el César, y tampoco me olvido de ti, Emilio, espero que a tots vosaltres us estigui anant bé allà on sigueu! And finally, I must mention Alex, Asaf, Francesc, Jon, Louis, Max, Robin, Vatistas and Yorgos. I hope I can join you the next time you (probably Robin) organise a group dinner!

Moltes gràcies Josep, Ramon i Sara pels comentaris que em vau fer a la predefensa de la tesi. Em van resultar molt útils i crec que el document ha millorat considerablement gràcies a vosaltres. Em vaig sentir molt còmode en aquella presentació, i això no sempre és fàcil.

Gracias también a ti, Bea, por la confianza para las prácticas de docencia en el DAC. I gràcies també a tots els professors d'IC per haver-me rebut i ajudat aquests anys fent les pràctiques de docència.

Gràcies, Alex, per interessar-te en allò que vaig començar amb el miniAMR i que s'hagués quedat al calaix si no fos per tu. La veritat és que vaig gaudir col·laborant en l'article.

Aprovecho para mandar un abrazo a Helena, Vicente, Mari y el resto de personal del Vèrtex. Muchísimas veces he comido allí durante los años que he estado con el doctorado, salvo este último, y eso es porque la comida, el trato y el precio se lo merecían. No dudo en que volveré a comer allí si estoy por la zona entre semana.

Tack så mycket till alla på UART! (I hope I've written this correctly, I haven't advanced in Swedish as much as I'd have to). Nevertheless, one main contribution of this thesis comes from my research visit there. Mihail, as I've told you more than once, you were a great mentor and I'm happy we still keep in touch. It would have been great to have physical thesis defences, so

that you could visit Barcelona for that occasion, but you are still invited to visit the city any other time. And David, if UART is such a great group it is your fault in great measure! It was very good for me going to Uppsala. And I miss doing Fika with all of you, Anastasia, Chris, Gustav, Hassan, Johan, Kim, Marina, Mehdi, Paul, Per and Ricardo. The mug you gave me is in a privileged place at home.

Marta, mama, papa, es muy típico decir que no habría podido hacer esto sin vuestro apoyo, pero es que es verdad. Sé que en algún momento durante todo el proceso de la tesis no solo he estado desquiciado sino que he sido desquiciante, pero ya se termina y esto es una parte del resultado. Y la cosa es que tampoco habría podido sin mi Ada, mi Otto y mi Frida. Siempre digo que sois muy pesados (Ada, pobrecita, tú ya no), pero especialmente este último par de años se habrían hecho mucho más duros sin vosotros por casa.

Yaya, avi, abuela, abuelo, ya lo digo al principio de esta tesis, pero va por vosotros. Sé la ilusión que os hace incluso si no entendéis nada de lo que hay en el documento. Pero el mismo orgullo que sentís, lo siento yo dedicándoos este trabajo.

Alberto, qué te voy a decir a ti que no sepas ya. Entre que estás lejos y la pandemia, poco nos hemos podido ver (en persona) este último par de años, pero eres seguramente con quien más rato he estado hablando. Se acerca ya el final de nuestro último proyecto en común, con el que hemos aprendido bastante los dos, pero estoy seguro de que vamos a encontrar excusas para hablar y que yo me vaya por los cerros. I com que parlo amb l'Alberto, aprofito i et dic, Laura, que encara no he tastat els paparajotes. Quan sigui tot més normal i us passeu per aquí un altre cop, us dono fulles de llimoner, que els vull tastar!

Guillermo, estos años habrían sido bastante más difíciles sin ti. Ir al teatro no deja de ser una excusa para estar luego hablando de la vida, quejarnos de los políticos, de la sociedad, de lo bueno y lo malo del trabajo... Tenemos que recuperarlo (ahora que está todo un poco más fácil de nuevo) y mantenerlo, porque como se dice, a mí me da la vida.

No acabaré sense esmentar l'ERIE de Recerca i Rastreig amb Gossos de la Creu Roja i la Coral l'Amistat de Premià de Mar. Tots aquests anys formant part del grup dels gossos (juntament amb el meu pare) i aquest últim gairebé any i mig a la coral m'han ajudat a desconnectar de les obligacions. I és totalment necessari per poder aguantar-ho tot.

All this work does not come for free. I have been partially supported by the Spanish Ministry of Education, Culture and Sport under fellowship number FPU15/03612, and by the Spanish Ministry of Science, Innovation and Universities under fellowship number EST18/00799. This thesis has also been supported by the Spanish Government (Severo Ochoa grant SEV2015-0493), by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P), by the Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328), by the RoMoL ERC Advanced Grant (grant agreement 321253) and the European HiPEAC Network of Excellence. The Mont-Blanc project has received funding from the EU's H2020 Framework Programme (H2020/2014-2020) under grant agreement numbers 671697 and 779877.

Contents

List of Tables

List of Figures

Chapter 1.

Introduction

The end of Dennard scaling has caused a stagnation of the CPU clock frequency. A solution to keep increasing the peak performance of *high performance computing* (HPC) systems has been to integrate more and more computing units in the systems, achieved with the interconnection of many nodes (like in supercomputers and clusters), the inclusion of multiple chips in each node and the increase of the number of cores in each chip. This increase in the computing capacity also requires an increase in the memory capacity and bandwidth to be able to feed data to the computing units. However, the speed of memory has not evolved as fast as the speed of processors, causing what is called the *memory wall* [160]. In the case of HPC systems, the increase in memory bandwidth has been achieved by providing local DRAM memories for each processor node with coherent communications between nodes, but also using newer technologies such as non-volatile memories (NVM) or high-bandwidth memories (HBM). These memories increase overall bandwidth, but result in *non-uniform memory access* (NUMA) behaviours: latency and bandwidth depend both on the node accessing the data and the node where the data is stored. In some cases, like in *cache-coherent NUMA* (ccNUMA) systems, there is coherence in the memory hierarchy inside the nodes of the system.

This heterogeneity in memory access time poses many challenges and requires solutions at the various levels of the computer stack. Moreover, it also worsens the programmability in the systems: when programmers have to take into account more and more characteristics of the system where

the applications are going to be executed to improve the performance, the time for developing and maintaining the applications may increase. This can have an impact in the portability of the code, its complexity and its quality: it is not the same to have all accesses with the same latency or with different latencies. Because of this, one of the trends in the last years has been to develop approaches to abstract the programmer from having to consider the hardware specificities of the underlying systems. Many of the solutions specific for NUMA rely on the OS or runtime system level: automatic page or thread migration by the kernel [43], workload scheduling taking into account the physical location of the data [117, 144, 152] or hybrid solutions [49]. Other more general solutions can be used in systems not showing NUMA effects, like adding hardware structures in the cache hierarchy for prefetching data in advance, known as hardware prefetchers [77, 78].

In this regard, one of the most recent trends to overcome the identified challenges consists in doing a holistic design of the hardware and the runtime system software, that is, designing both simultaneously [150]. One option can be the use of special hardware units that contain information useful for the runtime system, allowing it to make informed decisions or change some hardware configurations and improve the execution performance [28]. This holistic approach with the hardware and the runtime system software can relieve the programmers from having to take the hardware characteristics into account and has been used in many of the recent proposals for improving the performance of software applications in HPC systems.

1.1. Thesis objectives and contributions

This thesis has the goal of improving the performance of applications in systems that show NUMA behaviours by improving the data locality of the applications, all in a seamless way for the programmer. In order to reach this goal, we propose various approaches at the runtime software and the hardware level that affect scheduling of the workloads and prefetching of their data.

1.1.1. Performance and configuration models for interactions between NUMA and hardware prefetchers

In the first contribution, we study how optimising the NUMA scheduling (the placement of threads and data) or the data hardware prefetcher configurations separately gives results that fall away from the optimal. We present the performance benefits of optimising both for the NUMA scheduling and the data hardware prefetcher configurations in NUMA systems by means of offline modelling and online profiling. To address the large design space, we propose a prediction model that reduces the amount of input information needed and the complexity of the prediction required. We do so by selecting a subset of performance counters and application configurations that provide the richest profile information as inputs, and by limiting the output predictions to a subset of configurations that cover most of the performance.

The proposed model is robust and can choose near-optimal NUMA scheduling and prefetcher configurations for applications from only two profile runs. We further demonstrate how to profile online with low overhead, resulting in a technique that delivers an average of 1.68× performance improvement over a locality-optimised scheduling baseline with all prefetchers enabled.

1.1.2. Hardware prefetching for NUMA systems

Extending on the relevance of optimising not just the scheduling of NUMA applications but also the hardware prefetcher configurations, the second contribution proposes a generic hardware prefetching scheme that leverages the NUMA characteristics to enhance system performance. This knowledge is used to prefetch data more aggressively depending on the physical location of the predicted accesses. The extra latency when accessing non-local data is effectively hidden by prefetching this data to the last-level of cache, which has enough capacity to store such data.

This approach is evaluated using gem5, a cycle-accurate architectural simulator, in a modelled multi-socket NUMA system, proposing a simple hardware prefetcher that is aware of NUMA effects. We achieve a 1.30× speedup

on average when compared to a standard stride prefetcher. We also get a 1.10× speedup against the best-performing state-of-the-art prefetcher. Finally, we show that the ideas can be applied to other hardware prefetchers, obtaining a 1.06× speedup over the NUMA-unaware version.

1.1.3. Task-based applications in NUMA systems

In the last contribution, we propose scheduling techniques at the runtime system level to further mitigate the impact of NUMA effects on parallel applications' performance. We leverage the runtime system metadata obtained when using a task-based programming model, which can model the application in terms of a task dependency graph, where nodes are pieces of serial code and edges are control or data dependencies between them, to efficiently reduce data transfers.

This new approach, based on graph partitioning methods, adds negligible overhead and is able to provide performance improvements up to 1.52× and average improvements of 1.12× with respect to the best state-of-the-art approach when deployed on a 288-core shared-memory system. Moreover, using the graph partitioning-based scheduling approach reduces the coherence traffic by 2.28× on average with respect to the state-of-the-art solutions.

1.2. Thesis structure

The contents of this thesis are organised as follows:

- Chapter 2 presents the background and state-of-the-art in the hardware and software topics upon which this thesis builds.
- Chapter 3 presents the experimental methodology used within the three contributions of this thesis. To evaluate our proposals, we consider both real HPC platforms and a cycle-accurate simulator, as well as representative HPC applications and benchmarks.

- Chapter 4 presents the first contribution of this thesis, which develops performance and configuration models that consider the interaction between NUMA and hardware prefetchers in HPC systems.
- Chapter 5 presents the second contribution of this thesis, which presents a hardware prefetching scheme that alleviates NUMA effects in large HPC systems.
- Chapter 6 presents the third contribution of the thesis, which analyses how a runtime system can be used to reduce coherence traffic in a very large cache coherent NUMA system by leveraging the semantic information available in task-based parallel applications.
- Finally, chapter 7 concludes by summarising the contributions of this thesis, listing the publications resulting from it and considering what future potential research directions it suggests.

Chapter 2.

Background and related work

This chapter presents the previous work related with the topics discussed in the thesis. Section 2.1 introduces parallel and shared-memory systems. Section 2.2 describes caches and the memory hierarchy, together with improvements like hardware prefetching. Section 2.3 presents the options for writing and executing parallel software in shared-memory systems. Finally, section 2.4 gives an overview of holistic approaches to optimising performance in parallel systems and the path towards runtime-aware architectures.

2.1. Parallel computing, shared memory and NUMA

One of the early observations in the history of digital computers is that some actions or computations can be executed simultaneously, also called *in parallel*. This can be done at different levels of abstraction. Parallelism starts at the bit level, with computers like the Whirlwind [1], from the 1950s, being among the first to consider numbers as a group of bits (parallel representation) instead of a sequence of bits (serial representation) [2]. This has been standard in computers since then.

Parallelism can also be at the instruction and data levels. Flynn [58] describes a classification of computer architectures considering these levels. *Single instruction, single data* (SISD) architectures are those that allow executing a single instruction on a single data source at a time. Flynn also considers some parallelism in this case when single-CPU computers decode one instruction

at a time but the pipeline is segmented in such a way that allows for multiple instructions to be in flight. An example is the IBM System/360 Model 91, from 1964 [68]. *Single instruction, multiple data* (SIMD) architectures can execute the same instruction for different groups of operands. For such an architecture, one of the first proposals was SOLOMON (in 1962, not built), later superseded by the ILLIAC IV, in 1966, which was the first built computer to support SIMD [139]. *Multiple instruction, single data* (MISD) architectures are not common, but they refer to architectures that can execute multiple instructions on the same input simultaneously. Finally, *multiple instruction, multiple data* (MIMD) architectures can execute multiple instructions, each working with their own data, simultaneously. Modern systems with multicore processors are in this group, although such kind of systems have long existed, with the IBM System/360 Model 65, from 1965 [68], being one of the first existing examples.

There is also parallelism at the program level. The architectures that allow this type of parallelism are a particular case of MIMD: they allow executing multiple sequential programs simultaneously, each with their own instructions and data. An evolution is the parallelism at the thread level, which also allows for a single program to divide the execution in various simultaneous threads.

MIMD architectures can be further classified depending on how the memory is interconnected with the CPUs. In a *shared memory* system, the memory has a single address space and all CPUs can access the whole memory (they are also called *multiprocessors*). Such an early example is again the IBM System/360 Model 65. This is in contrast with *distributed memory* systems, in which each CPU has its own private memory, not accessible by the other CPUs (in this case, to share results the CPUs have to explicitly copy the data between them using messages). For example, supercomputers and clusters of computers fall within this category, although each of the components could be considered as a shared memory system as well.

Despite the first multiprocessors with shared memory were built in the mid 1960s, it was not until a couple of decades later when they became more general. However, they could not scale much more due to the contention

caused by the multiple processors trying to access the same memory resources. This was solved by giving each CPU or group of CPUs its own local memory whilst sharing the address space (allowing all CPUs to access the whole memory), with examples early like the DASH Multiprocessor from Stanford [90] and the SGI Origin [88]. The main effect of this was the unequal access latency to memory depending on the CPU that needed the data and the physical location of that data, with lower latencies for the local accesses. This is the reason why these systems are known as *non-uniform memory access* (NUMA) systems, in contrast with the *uniform memory access* (UMA) systems, as the previous ones became known.

In NUMA systems, the remote accesses to memory can have different latencies as well, making them even more non-uniform. For example, an SGI Altix UV100 [138] system with 12 CPU sockets can have three different remote latencies, apart from the local access latency. To deal with these differences in the latencies, there have been various proposals for programming parallel systems as explained in section 2.3.

2.1.1. Virtual memory: allowing multiprogramming and larger main memory capacity

Before main memory became cheaper and with large capacity, there was a need to have an auxiliary memory (with more capacity but much slower) to be able to store the programs and the data [45]. This is without considering the storage of applications that are not executing and their data. Having to manage the main and auxiliary memories from within the applications themselves, moving the instructions and the data from one to the other depending on the needs, lowers the programmability of the systems. Moreover, running multiple applications with overlapping addresses can cause problems, and there would be the need to make sure the compiled applications used different spaces.

A solution to these issues is *virtual memory*. In broad terms, applications use addresses that might overlap with other applications' and there is a translation mechanism, supported by the operating system, to convert these

addresses (called virtual) to the real addresses (called physical) [45, 46]. In terms of security, this also has the benefit that untrusted applications cannot know the physical addresses used by trusted applications, and with later advances the operating systems can even protect the pages used by each application or process. Another side effect is that applications can see a larger memory capacity than truly available.

Early developments already decided to use fixed-size pages among other options [46]. In this case, the memory is divided in pages of a specific size (which can depend on the architecture and the operating system) and the map between the virtual addresses and the physical memory is done in a page-by-page basis. This reduces the amount of addresses that have to be translated, needing less storage for the translations compared to translating every single address.

In the case of NUMA systems, this has the benefit of allowing to move the pages to a different physical location without changing the address seen by the application. This also allows for advanced workload scheduling in the system and to develop optimisations that consider the location of the data and the CPUs in charge of doing the computations. This is further detailed in section 2.3 and is one of the characteristics that some proposals studied in this thesis benefit from.

2.2. Caches and the memory hierarchy

The different rate at which the speeds of processors and memory have evolved have caused what is known as the *memory wall* [160]. In broad terms, the memory in a system is much slower than the processors, causing a bottleneck. Therefore, by just improving the computation speed of processors the performance of the executed code would hit a limit that would only be improved by making the memory faster. The way this has been done is by creating a hierarchy for the memory: between the main memory and the processors, there are some small capacity memories that act as temporary storage and are faster to access. These memories are known as

caches and store the data as lines, with multiple contiguous bytes in a single line (a fixed number that depends on the microarchitecture).

With the idea of caches initially proposed by Wilkes [158] in 1965, the first commercial computer to include a cache was the IBM System/360 Model 85 [68] in 1968. Later developments of caches, still used to date, include the following: using multiple levels of cache, splitting data and instruction caches, predicting addresses before they are used (prefetching) or using advanced replacement policies (what lines to remove from the cache to insert new needed lines) [140].

2.2.1. Cache coherence and ccNUMA

One of the most relevant aspects when using caches is *cache coherence* [137]. Since caches store copies of the data to have faster accesses, there is a risk that the data is not the same between multiple caches or between the cache and the main memory. In some systems this can be solved manually by the software, but many systems include hardware structures that implement *coherence protocols* to make sure that any time a memory address is accessed it is with the most updated contents. A particular case of coherent systems are *cache-coherent NUMA* (ccNUMA) systems. As their name suggests, these are NUMA systems that include hardware structures to keep coherence. In the context of this thesis, all NUMA systems considered in the evaluations are ccNUMA systems.

2.2.2. Hardware prefetching

Prefetching consists in predicting the addresses that are going to be accessed and bringing the corresponding lines in advance to the cache. Prefetchers can be implemented in software or directly in hardware, with more or less complexity in the heuristics used for the prediction. They can also be tailored to prefetching for data or for instructions. Mittal published a survey [106] in 2016 about prefetching in general, though this section will provide a high

level view of hardware data cache prefetching and give some details on more modern hardware prefetching techniques as well.

One of the simplest prefetchers is the *next-k line prefetcher*. When there is a miss, or an access to a prefetched line, this prefetcher would bring the k lines that come after that. This prefetcher, when used with a small buffer to prevent polluting the cache with the prefetches before they are used, is the *stream prefetcher* [78]. An evolution to this is the *stride prefetcher* [39]. In this case, the hardware detects a stride s between accesses to lines and, after access to line L it would prefetch lines $L+s$, $L+2\,s$, $L+3\,s$, and so on. There are more extensions to this, like allowing for different strides within a single memory instruction (PC-correlated prefetchers) or correlating the stride in one instruction with the previous strides to decide which will be the next one (delta-correlated prefetchers). Variations on these two prefetchers are usually implemented in current processor families like Intel's Xeon [153] or IBM's POWER [14].

Some complex applications follow less trivial access patterns and the above prefetching algorithms might not be adequate for these applications. One of the proposals that can serve as a base for many other ideas is the *global history buffer* (GHB) by Nesbit and Smith [107]. The GHB acts as a circular queue to keep the lists of accesses needed to feed the prefetching algorithms to decide the predictions. With this structure, a stride prefetcher can be implemented but also other prefetchers that use Markov chains [77] or other advanced techniques.

Modern advanced hardware prefetching

Regarding some recent hardware prefetchers, not included in Mittal's survey [106], Ayers et al. [7] show a methodology for classifying access patterns for prefetching, which allows to use different heuristics depending on the access pattern. This idea might be implemented in some commercial systems, since some hardware vendors have filed patents for similar proposals [64]. Peled, Weiser and Etsion [119] use a neural network for prefetching arbitrary access patterns, allowing for detecting more complex patterns. Still

within machine learning, Hiebel, Brown and Wang [63] use it to fine-tune the parameters of existing hardware prefetchers by using the output of some hardware counters. Ainsworth and Jones [3] propose a programmable prefetcher, which can be seen as an array of small in-order cores that can execute arbitrary prefetching algorithms, but at the cost of requiring a special compiler or extra work for the programmer to manually build these algorithms. Often, prefetchers are designed considering just sequential executions, so Liu, Yu and Huang [95] propose the use of thread-aware prefetching to reduce contention due to shared information.

Prefetchers for NUMA systems

Disabling all prefetching on NUMA systems can improve performance for irregular access patterns [99]. Moreover, prefetching can also increase the contention and hurt performance [93].

Not many works in the literature have tried to provide hardware prefetchers for NUMA systems. Hardware vendors have registered patents for changing the aggressiveness [65], throttling the prefetchers [70] or using different thresholds [96], all of them depending on the type of memory or the source memory. However, none of these ideas have been properly evaluated. A recent proposal for a NUMA-aware hardware prefetching scheme [135] is further discussed in chapter 5.

The importance of considering NUMA and prefetching simultaneously to improve the performance of parallel applications has already been shown [134], which explained in chapter 4. In general, considering multiple aspects simultaneously is key to optimise the performance of general-purpose parallel systems. This is discussed in more detail in section 2.4.

2.3. Parallel programming in shared-memory systems

When using a parallel system, there are multiple computing resources that can be used simultaneously. One widely used option is *multiprogramming*,

which consists in executing different applications at the same time. However, another widely used option are *parallel applications*, which are applications that can use the different CPUs in the system to execute different parts of their code at the same time. Operating systems provide support for executing parallel applications by using *threads*, like POSIX Threads [141] in UNIX-like systems (Linux, BSD, Mac OS...) or Win32 Threads [15] in Windows systems, which are managed using system calls. Some programming languages and frameworks provide wrappers or abstraction layers around these system-provided options to ease code portability between systems.

To write parallel applications using the OS-provided threads or the basic wrappers from languages, programmers need to take into account all aspects from their side: creating and destroying the threads, using locks to prevent race conditions... To make programming easier, there are *application program interfaces* (API) and libraries like OpenMP [113] and Threading Building Blocks (TBB) [131] that implement the *fork-join* model.

In the fork-join model, there is a group of available threads (called a *thread pool*), which can also be created an destroyed, the code has some regions marked as "parallel" (meaning that can be executed by multiple threads) and annotations on the visibility of the variables (whether they are private, specific for each thread, or shared, when all threads use the same address for that variable). Loops can be marked as parallel and their workloads are then divided among all threads. To prevent race conditions, some code fragments can be marked as exclusive and only one thread at a time can execute them.

In the case of OpenMP, widely used in HPC for shared-memory programming, the annotations in the code are done using *compiler directives*. The preprocessor and the compiler then transform these directives into the corresponding API calls to then compile it before executing.

An alternative to these models is using a *message-passing* model such as MPI [146], widely used in clusters and supercomputers with multiple nodes and distributed memory, but also in shared-memory systems. In this case, there are multiple instances of the application running simultaneously, each doing its work and doing an explicit communication of the results to the

other instances by means of a message-passing API. This model is often combined with OpenMP or another model to build very large applications. In the case of NUMA systems, an example of use is having one instance of the application (called MPI rank or process) per NUMA node, using OpenMP inside each NUMA node but communicating the results in the different NUMA nodes with message-passing.

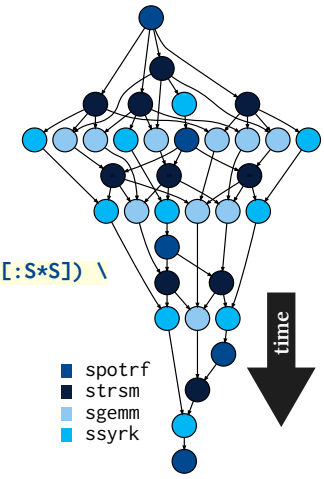
2.3.1. Task-based programming

One of the evolutions of OpenMP is the support for *tasks* since version 3.0, driven by projects like OmpSs. Tasks are pieces of sequential or fork-join parallel code that are executed by a thread or group of threads. With extra additions like the support for *dependencies* since version 4.0 of the standard, tasks can have annotations to indicate the data that is needed to execute them and the data that they output. There is also support for nested task parallelism, in which a task can create subtasks and are executed asynchronously as well.

Listing 2.1 shows an example of a task-based parallelisation for a Cholesky matrix decomposition written in C using compiler directives (`#pragma`) specific for OpenMP. The `depend` clause is used to indicate the data dependencies, indicating whether each data block is used as input or output and, optionally, its size. The sequential code is split into four task types: `spotrf` to calculate the Cholesky decomposition of the diagonal blocks, `strsm` to solve the linear systems that define the below-the-diagonal blocks, and `sgemm` and `ssyrk` to do matrix multiply and rank `S` operations to update the rest of the matrix.

A task-based code with data dependencies can be represented as a *directed acyclic graph* (DAG), as shown in the graph accompanying listing 2.1. In this graph, known as the *task dependency graph* (TDG), nodes represent tasks and edges express dependencies between them. This graph is usually built and maintained by the *runtime system* to orchestrate the parallel execution. Thanks to the automation given by the runtime system, instead of having to explicitly manage the communications between threads and their schedule,

```
void cholesky(int T, float *A[T][T], int S) {
  // each A[i][j] has size S*S
  for (int k = 0; k < T; ++k) {
#pragma omp task depend(inout: A[k][k][:S*S])
    spotrf(A[k][k]);
    for (int i = k + 1; i < T; ++i) {
#pragma omp task depend(in: A[k][k][:S*S]) \
                 depend(inout: A[k][i][:S*S])
      strsm (A[k][k], A[k][i]);
    }
    for (int i = k + 1; i < T; ++i) {
      for (int j = k + 1; j < i; ++j) {
#pragma omp task depend(in: A[k][i][:S*S], A[k][j][:S*S]) \
                 depend(inout: A[j][i][:S*S])
        sgemm(A[k][i], A[k][j], A[j][i]);
      }
#pragma omp task depend(in: A[k][i][:S*S]) \
                 depend(inout: A[i][i][:S*S])
      ssyrk(A[k][i], A[i][i]);
    }
  }
}
```



Listing 2.1.: Task-based Cholesky decomposition using OpenMP 4 annotations and its corresponding TDG when $T = 5$.

the programmer can just express the dependencies and the runtime system will be in charge of managing the execution.

Using task-based programming can improve the programmability of the applications and allow for getting better performance easily. For example, Rico et al. [133] show how using a task-based approach for a mesh refinement application instead of fork-join or message passing in a shared-memory system can substantially improve the performance of the execution, even without advanced scheduling of the tasks.

Task scheduling

A key aspect of task-based programming models is the scheduling of the tasks. When using tasks, the programmer is relieved of having to manage how they will be distributed in the system. However, this flexibility can come at the cost of reducing the execution performance if the scheduling is done blindly. One of the simplest methods is assigning the tasks of the graph to a CPU using a *first-in, first-out* (FIFO) algorithm: once a task can be executed because all the input dependencies are resolved, it is inserted at the end of

a queue and the runtime system will assign the first task of the queue to any available CPU to execute.

From this basic scheduling technique, there are various advanced proposals derived from it. One simple addition is, instead of using a normal FIFO queue, to include priorities in the tasks and using a priority queue when storing the ready tasks. Built upon this, Chronaki et al. [40] propose the criticality-aware task scheduler (CATS) for heterogeneous systems: the runtime system decorates the tasks with priorities trying to find the critical path for the execution in the TDG, and then uses this information to schedule the tasks in slower or faster CPUs.

2.3.2. Work scheduling and data placement in NUMA systems

In order to mitigate NUMA effects, techniques for migrating threads, memory pages or both already exist [43, 49, 147]. These techniques consist in moving computation near to data or vice versa with the goal of reducing memory access times. One benefit of these approaches is that they are agnostic of the application. However, this comes at the cost of not exploiting any application-specific information to predict the accesses to memory. As such, these proposed OS-level thread or page migration techniques only take action when the application is already suffering from remote memory accesses, which can give suboptimal results in many cases. Oppositely, other approaches transfer the NUMA management responsibility to the programmer [111, 151], exploiting information at the application source code level to carry out NUMA-aware scheduling decisions. The main issues with these approaches are that they may require significant code refactoring and programmer effort to be effective as well as the reduction in portability.

Offline methodologies can allow programmers to decide how to tackle with configurations when executing the applications, by means of some performance counters or execution traces. Diener et al. [50] characterise the communication and memory usage of applications to tune thread and data placement in systems. Beniamine et al. [13] show a tool to create a visualisation of their behaviour and help decide how to fix the performance

issues. Similarly, Trahay et al. [148] present a tool to understand the evolution of memory access patterns. Popov et al. [123] and Popov, Jimborean and Black-Schaffer [125] use the CERE framework to build codelets, which make the evaluation of configurations faster and allow doing a simultaneous exploration of multiple thread and page mappings using offline methods.

Radojković et al. [128, 129] evaluate how different thread placement policies perform, and Durillo et al. [55] evaluate the benefits of using higher or lower parallelism. Diener et al. published a survey [48] with details on both data and thread placement and scheduling in NUMA systems. Many of the evaluated works make use of certain runtime information and hardware counters and are analysed in section 2.4.

Task scheduling in NUMA systems

Techniques that take advantage of shared memory systems which integrate different memory devices have been studied for long time. For instance, Yan et al. [162] present the hierarchical place trees (HPT), in which the programmer describes the memory hierarchy as a tree and the tasks are distributed on the tree leaves (where the workers reside) programmatically on the source code of the application. Similarly, Chatterjee et al. [37] show a domain-specific language that allows the programmer to include the locality information using affinity groups for the tasks in a file separated from the application source code, making the approach more portable.

Drebes et al. [53, 54], as well as Virouleau et al. [152] later, present a scheduling technique that can be seen as a NUMA-aware FIFO algorithm. They propose scheduling the tasks initially with a FIFO scheme, or some other technique, but as new tasks become ready, they are scheduled in a CPU that is in the same NUMA node as most of its data dependencies (so that most local access will be local). To prevent starvation of CPUs or load imbalance, they implement a work-stealing approach. This means that a CPU that is idle can execute a task initially assigned to be executed by a CPU in another NUMA node when all CPUs from that node are busy.

Since the application can be modelled as a (directed acyclic) graph, graph algorithms can be applied to the TDG, or to the partial TDG available at a specific point in time during the execution. One example of algorithm is graph partitioning, which has been used statically in message-passing applications since they can be easily represented with an undirected graph [120]. This has been done mostly in two ways: i) dividing a graph where each vertex corresponds to a block of data and the edges represent simultaneous use of data by several processes, and ii) considering a process graph, mainly related to message-passing programming models, where each vertex corresponds to one of the processes and the edges represent communications between them. Our work is the first to dynamically apply graph partitioning to reduce NUMA effects on shared-memory systems, whereas prior proposals partition the graph statically or focus on load balancing distributed memory systems.

One recent development to guide load balancing via graph partitioning techniques in task-based applications is SPAWN by Papin et al. [117]. With SPAWN, the programmer adds geometric information to the task decomposition of the problem (which has the shape of a structured mesh, where edges are not directed, instead of a DAG). This approach assigns the tasks to the CPUs and other processing elements, like GPUs or accelerators, by using a Voronoi tessellation of the mesh. During the execution, the processing elements get an electrical charge value depending on the amount of work they have and the tessellation is thus updated. Afterwards, they move on the task mesh by means of Coulomb's law and the tessellation is updated. This implies the need to have a correspondence between the problem domain and a metric space.

There have been previous results in partitioning directed acyclic graphs using standard partitioners: Tanaka and Tatebe [144] used the multiple-constraint capabilities of METIS (that do the partition in a multidimensional space) to schedule workflows, which are typically more coarse-grained than shared-memory codes. The first use of graph partitioning to directly partition the TDG and use the partition for scheduling the tasks in a NUMA system was presented in 2018 [136]. Chapter 6 has a detailed view of this proposal.

2.4. Holistic performance optimisation and runtime-aware architectures

The end of Dennard scaling, with a halt in the frequency increase, derived in the integration of more and more computing units. This has been possible thanks to the reduction of the transistor size as predicted by Moore's law. However, since physical limits exist, the size reduction rate is decreasing and new approaches are needed to obtain better performance from these systems until new electronic technologies (like alternatives to silicon) or computer paradigms (like quantum computers) evolve and can be used successfully.

Using performance metrics and other details available at execution time, as well as some application-specific information, allow making more informed decisions when scheduling and executing applications. In order for these methodologies to be effective, the profiling and metric extraction must have low overhead. Otherwise, the overhead could hinder the optimisation gains.

Wu and Martonosi [159] consider activating or deactivating prefetchers to prevent execution interferences inside applications using some performance counters as a metric. Khan et al. [85] introduce a runtime framework that combines software and hardware prefetching to maximise the throughput of multicore processors by sampling some performance metrics. Chasapis et al. [35] show how the runtime system can be used to model the power differences in processor manufacturing and propose a task-based scheduling algorithm in power-restricted NUMA systems. Jiménez et al. [76] evaluate some prefetching configurations at runtime to build an adaptive prefetching algorithm that automatically selects the best prefetcher configuration. Similarly, Ortega et al. [114, 115] propose using a runtime-level library to modify the hardware configuration knobs (like prefetcher and SMT) at execution time using performance counters. Broquedis et al. [20] and Dashti et al. [43] profile and make optimisation decisions in the scheduling as the application runs.

Other options that use runtime information from the applications, without using performance counters, allow to speed up the execution of task-based parallel applications. Caheny et al. [24, 26] propose executing with a NUMA-aware runtime to reduce the cache coherence traffic. Brumar et al. [21] use the runtime information about the dependencies to build a memoisation approach to approximate the computations of executions and reduce the execution time. Jaulmes et al. [72–74] evaluate the use of runtime systems in reliability and fault tolerance.

2.4.1. Using models to drive configurations

Performance prediction models use input features to predict the best configuration [89, 157]. Some of the most commonly used features include thread access patterns [47], performance counters [94, 155], static code properties [157], and page, thread or inter-thread communication and sharing [50].

Wang, Davidson and Soffa [155] use integer programming (optimisation using integer variables) to predict bandwidth usage and thread allocation across different degrees of NUMA nodes, while Denoyelle et al. [47] add thread and page mappings. They build performance prediction models and conclude that performance counters and thread access patterns provide similar results as inputs. In both cases, inputs are collected by executing the application. Liao et al. [94] propose a tuning framework that predicts the best prefetching based on performance counters. Hiebel, Brown and Wang [63] extend this prediction to target more fine-grained phases of execution inside applications. Chasapis et al. [36] derive mathematical formulas to predict power consumption due to manufacturing variability and change the scheduling to meet power budgets.

Performance counters and sampled executions can be used to build machine learning models that allow to choose combined configurations of NUMA and prefetchers at execution time for applications not in the model [134]. This also shows that finding a good prefetcher configuration together with an adequate thread and data placement in NUMA systems cannot be done

independently if the goal is to obtain the best performance. Further details are given in chapter 4.

2.4.2. Runtime-aware architectures

Runtime-aware architectures, proposed by Valero et al. [150], are a paradigm that follows the path of taking a holistic approach. In these architectures, both hardware and software are designed together and are managed thanks to the use of a runtime system. As shown previously, taking an approach that considers multiple factors at the same time can provide with more performance and further optimisation options than considering them separately. Casas et al. [28] build on this idea and propose designing special hardware that can be driven by the runtime system and changes the behaviour using the available information of the system (like using hardware counters or storing some information used by the runtime system in new hardware units).

Both Garcia et al. [59] and Papaefstathiou et al. [116] propose using the runtime information about dependencies to prefetch data blocks. On the other hand, Manivannan et al. [100, 101] use the dependency and task information to predict dead blocks that can be substituted in the caches. Dimić et al. [52] design cache insertion policies managed by the runtime system based on the use of re-reference intervals. Dimić et al. [51] also propose a runtime-aware methodology to manage the computation of reductions in the cache hierarchy. Alvarez et al. [4, 5] use compiler information and the runtime system to manage the coherence of scratchpad memories. Caheny et al. [25] use runtime information to selectively deactivate cache coherence. Barredo et al. [12] propose a compaction-restoration unit to join sparse predicated vector instructions into denser vectors. Later, they propose a near-memory accelerator to rearrange sparse memory regions into dense blocks to benefit from locality [11]. Castillo et al. [30] propose scaling the voltage and frequency of processors using task criticality information and storing part of the metadata in hardware structures. Jaulmes et al. [75] present a runtime-driven configuration for error-correcting codes in

DRAM. As more general solutions, Castillo et al. [29], Etsion et al. [57], Kumar, Hughes and Nguyen [87] and Tan et al. [142, 143] evaluate the inclusion of hardware structures to implement part of the runtime system.

Runtime-aware architectures show a great deal of options for the evolution of computers. They provide flexibility to the programmer and achieve a performance level that would not be possible otherwise. Within this topic, the goal of this thesis is to improve the performance of parallel applications in NUMA systems.

Chapter 3.

Experimental methodology

This chapter describes the experimental methodology used in the development of this thesis. Part of the work presented here has been evaluated using real machines. In some cases, a simulation environment is needed to evaluate the proposals. Various sets of representative applications and benchmarks have been executed in the experiments. Section 3.1 describes the real systems used in the experiments of chapters 4 and 6. After this, section 3.2 gives an overview of the simulated systems used in chapter 5. Finally, section 3.3 lists the different benchmarks used in this thesis.

3.1. Real NUMA systems

Many of the experiments in this thesis have been executed in real NUMA systems, comparing the performance of the proposed ideas in the different systems. We consider four different x86-64 platforms with processors built by Intel. For the performance and configuration models for NUMA and prefetchers, presented in chapter 4, we use two systems. The first system is a four-socket Intel Xeon E5-4650 (Sandy Bridge-EP) with 128 GB of RAM, with the sockets interconnected using Intel QPI (Quick Path Interconnect); the second system is a dual-socket Intel Xeon Platinum 8168 (Skylake-SP) with 188 GB of RAM and sockets interconnected using Intel UPI (Ultra Path Interconnect). Both systems run Ubuntu 18.04 with Linux 4.19 LTS kernel.

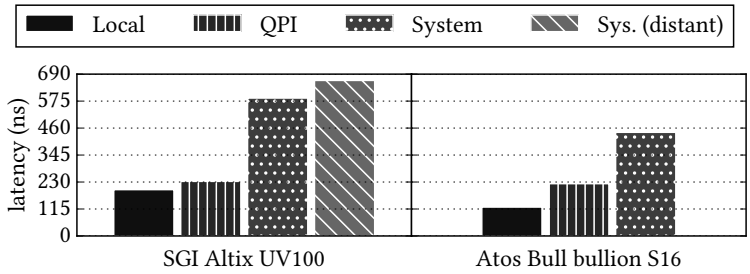


Figure 3.1.: Measured memory latencies in milliseconds as we increase the working set size with LMbench `lat_mem_rd`.

3.1.1. Large ccNUMA systems

The other experiments executed on real machines are for the scheduling of task-based applications in NUMA systems, detailed in chapter 6. The first machine is an SGI Altix UltraViolet 100 with 3 IRU (*internal rack units*) interconnected with SGI NUMAlink at 15 GB/s. Each IRU contains two IP93 blades with two 8-core Intel Xeon E7-8837 CPU (Westmere-EX) at 2.66 GHz and 24 MB of shared last-level cache, and 16 DIMM of 16 GB DDR3 RAM. Sockets in the same blade communicate via Intel QPI. The system runs SUSE Linux Enterprise Server 11 with Linux 2.6.32 kernel. The second machine is an Atos Bull bullion S16 with 8 modules, each one with two 18-core Intel Xeon E7-8890 v3 sockets (Haswell) at 2.50 GHz and 45 MB of shared last-level cache. Each socket has 512 GB of local RAM and is connected via Intel QPI to the other socket in the module; modules are interconnected using the Bull Connecting Box and communicate via the BCS2 (*Bull Coherence Switch 2*) [6]. The system runs Red Hat Enterprise Linux 6.5 with Linux 2.6.32 kernel.

We use `lat_mem_rd` from LMbench [102] to measure the true memory latencies, shown in figure 3.1, and pass that information to the partitioning library. In the Altix, accesses within the same blade have an increased latency of 17 % compared to local memory, while there is a significant latency penalty of 200 % to access data in other IRUs, and close to 240 % in the most distant sockets. In the bullion S16, the access latency via QPI has an extra penalty of 79 % and of 260 % for remote accesses via the BCS2.

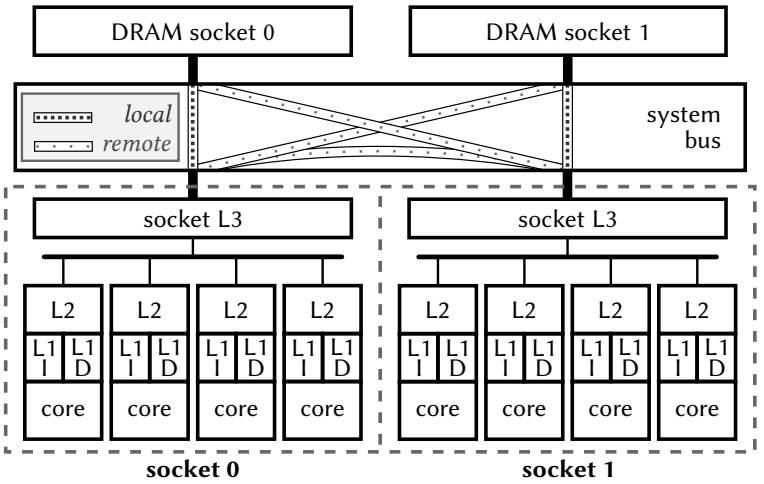
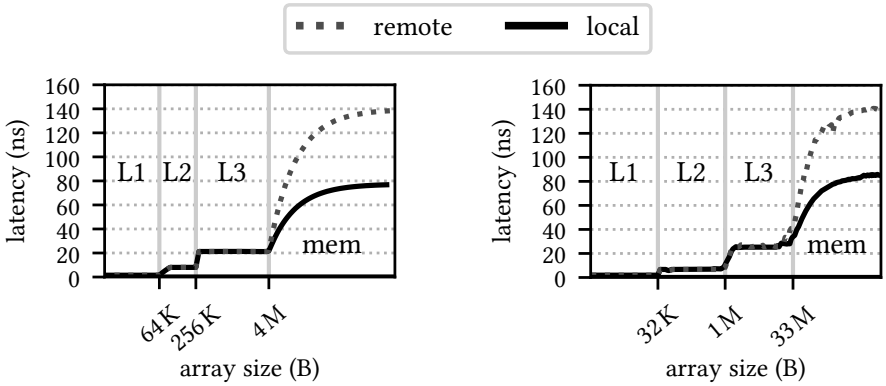


Figure 3.2.: High-level diagram of the simulated gem5 system.

3.2. Simulation of NUMA Systems

For the design and evaluation of the NUMA-aware prefetcher presented in chapter 5, we have executed simulations using the gem5 simulator, version v20.1.0.2 [97] and the classic memory model. The gem5 simulator is a cycle-accurate simulator that allows simulating a full system architecture at different levels of detail. It provides the infrastructure to simulate complete systems with real *instruction set architectures* (ISA). The full-system simulation includes a five-stage architecture, options to run multicore systems, with a cache hierarchy and various memory devices.

We have extended gem5 to be able to account for the different latencies in NUMA systems. We model and simulate the NUMA behaviour by adding some extra delays at the system bus crossbar, which interconnects the last-level cache with the memory controllers that manage the main memory space. These additional delays depend on the requesting port and the destination physical address: if the requesting port comes from a CPU that is not local to the destination address, the access will take longer. This allows for a simple but effective modelling of a real NUMA system. A high level diagram of this implementation is shown in figure 3.2.



(a) Gem5 (4 cores/socket) (b) Intel Skylake (24 cores/socket)

Figure 3.3.: Latencies of the memory hierarchy in a simulated environment and a real system

Latencies of the different cache levels up to main memory in a dual-socket gem5 simulated environment and a real Intel Xeon Platinum 8160 (Skylake) system. Latencies measured using `lat_mem_rd` from LMbench.

If the extra latency that is added for remote accesses matches with that of a real system, our implementation shows a realistic behaviour. For example, figure 3.3 shows the latency measurements with respect to the array size according to `lat_mem_rd` from LMbench [102] for both gem5 (with an extra latency of 60 ns for remote accesses) and a real dual-socket system with Intel Xeon Platinum from the Skylake family (where remote accesses cost 60 ns more than local accesses). The accesses in the simulated system are more stable and show less noise compared to the real system, and the sizes of the caches (shown with the array size and the jumps in latency) are different to accommodate for the different number of cores, but the behaviours are comparable.

The OS inside the simulated environment sees the NUMA characteristics thanks to the devicetree description, written following the NUMA binding description [108]. The crossbar does the delays using the same addresses as defined in the `/memory` nodes in the devicetree file. The common characteristics for the simulated systems, using 4 sockets, are detailed in table 3.1.

Table 3.1.: Common characteristics of the simulated systems.

System details	
Sockets	4 quad-core identical sockets
Page size	4 KiB
Cache line size	64 B
Latency to memory	80 ns (local), 160 ns (near socket), 240 ns (far sockets)
Socket details	
Cores	4 single-threaded, out-of-order cores
L3 shared cache	8 banks, 512 KiB/bank, 16-way, 1 port/bank
Core details	
L1 inst. cache	32 KiB, 2-way, single port
L1 data cache	64 KiB, 2-way, single port
L2 cache	256 KiB, 8-way, single port
Load-store queues	48 load entries, 48 store entries
Instruction queue	92 entries
Reorder buffer	192 entries
Branch predictor	Tournament predictor

Prior works have simulated NUMA systems using execution traces. In order to run these simulations, detailed traces that contain different hardware counters and machine status information are stored and then parsed with a different software that makes some modifications depending on the expected behaviour of the target system. For example, Daoudi et al. [42] develop sOMP, a simulator that uses traces obtained through the OMPT [56] interface to simulate task-based OpenMP applications with NUMA effects. However, they are not modelling the level 1 and 2 caches. Another trace-based alternative is TaskSim [132], which has been used to model current and future systems at various levels of detail [60, 62] (from cache to inter-node communication). It could potentially be used to model systems like the ones we use, but we would not get the level of detail that we are looking for.

3.3. Workloads

In this thesis we have evaluated our proposals with different benchmarks and applications, programmed using both fork-join and task-based programming

models. This section gives the details of the various benchmarks and the reasoning behind the decision of using them.

3.3.1. Fork-join parallel applications

As presented in section 2.3, one of the main approaches for programming parallel shared-memory systems is the fork-join programming model, with OpenMP being one of the most used options nowadays. Many applications and benchmarks for HPC and scientific computing use this programming model and many of them suffer from NUMA effects. For this reason, we do the performance evaluation of the interactions between prefetchers and thread and page mappings in NUMA systems (further detailed in chapter 4), and also the NUMA-aware prefetchers (explained in chapter 5), using fork-join applications.

NAS Parallel Benchmarks

One widely used benchmark suite is the NAS Parallel Benchmarks suite [8, 9]. Most of these benchmarks are derived from computational fluid dynamics applications. In the case of BT, LU and SP, they are pseudo-applications that represent a system derived from the 3D Navier-Stokes equations.

The **Block tri-diagonal solver** (BT) is a pseudo-application that solves multiple independent block-tridiagonal systems using blocks of size 5×5.

The **Lower-upper Gauss-Seidel solver** (LU) is another pseudo-application that solves a sparse linear system by splitting it into block lower-upper triangular systems. It uses the symmetric successive over-relaxation method (SSOR) to get the solution.

The **Scalar penta-diagonal solver** (SP) is the third pseudo-application, in this case it solves a system with scalar pentadiagonal bands of linear equations.

This suite also has various kernels that are widely used in computational fluid dynamics applications.

Conjugate gradient (CG) is a kernel to find the smallest eigenvalue of sparse positive-definite matrices.

The **discrete 3D fast Fourier transform** (FT) performs three one-dimensional fast Fourier transforms, one per spatial dimension, and stresses one-to-one communications.

Multi-grid (MG) is a memory-intensive kernel that is applied to a sequence of meshes to compute the solution of the 3D scalar Poisson equation.

There are other synthetic benchmarks in the suite, like an **integer sort** (IS), which does not use any floating point arithmetic, and an **embarrassingly parallel computation** (EP). We use all these benchmarks for both the modelling of interactions between prefetchers and thread and page mappings in chapter 4 and the performance evaluation of NUMA-aware prefetching in chapter 5.

For the interactions between prefetchers and thread and page mappings, instead of the official Fortran implementation, we use a C implementation [9, 124] based on version 3.0 of the benchmarks.

Rodinia benchmarks

In our evaluation, we include some benchmarks from Rodinia [38], which gathers benchmarks that have versions for heterogeneous computers. In our case, we just use the OpenMP versions for general-purpose architectures.

Breadth-first search (BFS) is a benchmark that implements a parallel version of the classic breadth-first graph traversal algorithm.

CFD solver (CFD) is solver for the 3D Euler equations for compressible flows, in a finite volume and using an unstructured grid.

Hotspot and **Hotspot 3D** are used to estimate processor temperature by means of the floor plan and simulated power measurements. The 3D version is for 3D integrated circuits.

K-means is a clustering algorithm used to find k clusters. It uses the mean values as centroids for the clusters, updating the centroids at each iteration. The algorithm is iterative and finishes once no point changes of cluster.

LU decomposition (LUD) is another version of the LU decomposition of a matrix. This is implemented for dense linear algebra, in contrast with the version from the NAS Parallel Benchmarks.

K-nearest neighbours (NN) finds the k neighbours that are nearest to a given point, using the surface of a sphere and latitude and longitude for calculating the Euclidean distances. The data is represented as an unstructured set.

Needleman-Wunsch (NW) is a non-linear method for finding global optimums, using dynamic programming, of DNA sequence alignments.

Pathfinder finds the path with the smallest weight from the bottom row to the top in a 2D grid, where each step, always moving upwards either diagonally or vertically, has an associated weight.

Streamcluster is a clustering algorithm that, for a stream of points, uses the median to find the centre of the cluster that minimises the distance. In comparison with K-means, the number of clusters is given by a range and is not a single number, so it can create a new centre (or cluster) if it minimises the distance.

Other benchmarks

We also include applications from other sources, including the CORAL-2 Benchmarks[1] suite, provided by the Advanced Simulation and Computing programme in the Lawrence Livermore National Laboratory (LLNL).

Black-Scholes, from the PARSEC benchmark suite [16, 17] executes in parallel various instances of a solver for the Black-Scholes partial differential equation. This equation models how the price of an asset updates the value of an option.

[1] https://asc.llnl.gov/coral-2-benchmarks

CLOMP [19] is from CORAL-2 and is the C version of the Livermore OpenMP benchmark. It measures OpenMP and threading overheads. Its behaviour tries to approximate typical scientific workloads with strong scaling.

HACCmk is a standalone code from Argonne National Laboratory that serves as a strong-scaling benchmark for the short-force evaluation kernel of HACC from CORAL-2. The original HACC application simulates how structure can be formed in fluids without collision, under the influence of gravity, in an universe that is expanding.

LULESH [66, 80], a standalone benchmark from LLNL, models an hydrodynamics application and approximates the equations using a mesh to divide the space, using an unstructured hex mesh.

Quicksilver, also from CORAL-2, is a proxy application for the Mercury application from LLNL that solves a simplified Monte Carlo particle transport problem, where particles are transported by different kinds of radiations.

Evaluated inputs

In order to properly evaluate the NUMA effects, the selected inputs need to be large enough that they do not fit in the cache and the latencies due to NUMA are not hidden by the cache capacity. Table 3.2 includes a summary of the inputs used in the fork-join parallel applications.

Accelerating the evaluation with codelets

As executing the full applications for the complete set of NUMA+Prefetcher configurations is impractical, in chapter 4 we use a technique called *codelet execution* [109, 124] instead. Codelet execution extracts hot regions from the application as small, representative *codelets* and uses them to characterise the application's performance. Codelets are on average 66× faster [125] to evaluate than running the full application.

Codelet execution is faster because it only executes a few instances of each region (instead of hundreds during the original run). To ensure that the

Table 3.2.: Summary of inputs for fork-join parallel applications.

	benchmark	input description
NAS Parallel Benchmarks		input class A (limited iterations)
Rodinia benchmarks	BFS	graph of one million nodes with degrees between 3 and 6 (`graph1MW_6.txt`)
	CFD	`missile.domn.0.2M`
	Hotspot	1024×1024 grid, 2 iterations
	Hotspot 3D	1024×1024 grid per layer, 1 layer/thread, 100 iterations
	k-means	KDD Cup data set (494 020 points, 35 dimensions) into 5 centres
	LUD	8000×8000 matrix
	NN	5 nearest neighbours for default input database, latitude 30°, longitude 90°
	NW	2048×2048 size, penalty 10
	Pathfinder	width of 100 000 elements, 100 steps
	Streamcluster	$1 \cdot 10^6$ points of 128 dimensions, handled in chunks of size $2 \cdot 10^5$, clustered into 10 to 20 centres, using 5000 intermediate centres
CORAL-2	CLOMP	32 threads, 16 parts, 400 zones per part, 32 bytes per zone
	Quicksilver	Problem #2 for the CORAL-2 benchmarks, 1331 mesh elements per node (`Coral2_P2_1.inp`)
	Black-Scholes	10 million inputs (`in_10M.txt`)
	HACCmk	default strong-scaling input
	LULESH	cube mesh of length 40

codelet execution matches the behaviour of the region within the application, codelets implement a short warm-up phase that configures the system state (e.g., caches) to match the application's native execution.

Codelets have been shown to be accurate enough for both micro-architectural evaluation [110] and NUMA configuration studies [125]. This is because parallel regions typically exhibit similar behaviour [148]. For our fork-join applications, we extract codelets for instances of each important OpenMP parallel region. This results in 57 codelets, which take approximately 2 days to execute across all configurations, but would take over 4 months with full executions.

The list of codelets for each application is detailed in table 3.3.

Table 3.3.: List of codelets for each fork-join parallel application.

benchmark	list of codelets
BFS	BFS 1, BFS 2
Black-Scholes	Black-Scholes
CFD	Euler 3D flux, Euler 3D time step
CLOMP	CLOMP do barrier, CLOMP mod 1, CLOMP mod 2.1, CLOMP mod 2.2, CLOMP mod 3.1, CLOMP mod 3.2, CLOMP mod 3.3, CLOMP mod 4.1, CLOMP mod 4.2, CLOMP mod 4.3, CLOMP mod 4.4
HACCmk	HACCmk
Hotspot	Hotspot
Hotspot 3D	3D
K-means	K-means
LUD	LUD
LULESH	LULESH cont, LULESH energy, LULESH eval, LULESH force, LULESH kinematics, LULESH press 1, LULESH press 2, LULESH stress
NN	NN
NPB BT	BT x_solve, BT y_solve, BT z_solve
NPB CG	CG iter, CG residual
NPB FT	FT step 1, FT step 2, FT step 3
NPB IS	IS main, IS rank
NPB LU	LU rhs, LU ssor
NPB MG	MG ps inversion, MG residual
NPB SP	SP rhs, SP x_solve, SP y_solve, SP z_solve
NW	Needle 1, Needle 2
Pathfinder	Pathfinder
Quicksilver	QuickSilver
Streamcluster	Streamcluster gains 1, Streamcluster gains 2

3.3.2. Task-based parallel applications

As mentioned earlier in section 2.3.1, task-based parallel programming provides high flexibility when programming and allows for achieving a higher performance. One key aspect is the possibility of seeing the application as a directed acyclic graph: it allows to apply graph algorithms on it to do optimisations at execution time, as we study in chapter 6.

The task-based **Conjugate gradient** (CG) is an iterative method for solving linear symmetric positive-definite systems of equations, represented with sparse matrices. It computes the solution by building a basis of orthogonal vectors in each iteration. This benchmark is not the one from the NAS

Parallel Benchmarks suite presented in the previous section: we use a sparse matrix version with the task decomposition described by Jaulmes et al. [72].

Gauss-Seidel is an algorithm solving the stationary heat diffusion problem using the iterative Gauss-Seidel method with a 4-element stencil (top, bottom, left, right). In this algorithm, the top and left elements used for calculating the current block are from the same iteration, whilst the right and bottom elements are from the previous iteration. The graph follows a wavefront shape and various iterations can be run simultaneously, as long as the dependencies between blocks of different iterations are preserved. The implementation is based on a task decomposition given by tiles with the tile contents contiguous in memory (instead of the rows) and halos between the tiles of the matrix to communicate the borders of the tiles. The use of the halos for communicating improves the data locality in the cache and reduces the amount of data that might need to be communicated between NUMA nodes.

The **Integral histogram** computes a cumulative histogram for each pixel of an image, using a cross-weave scan as described by Porikli [126]. This algorithm is used as a kernel for some image recognition methodologies. In our case, the calculations of the histograms of different images are overlapped to increase parallelism.

Jacobi solves the stationary heat diffusion problem using the iterative Jacobi method with an implementation derived from the Charm++ project [34, 79]. This implementation uses a 5-element stencil (top, bottom, left, right, centre) and a task decomposition given by blocks of rows. The double-buffer nature of Jacobi gives an embarrassingly parallel algorithm inside every iteration with a very symmetric TDG, hence it becomes simple to partition in contrast to the Gauss-Seidel code that solves the same problem.

NStream is a synthetic benchmark to measure memory bandwidth based on STREAM [103]. This task-based parallel implementation works with N independent arrays (a multiple of the number of threads, usually). Its task graph is made of N isomorphic connected components, so partitioning it should be as easy as assigning every component to one NUMA domain.

The **QR factorisation** of a matrix A is a product $A = QR$ where Q is orthogonal and R is upper triangular. This kind of factorisation can be applied to any kind of matrix, it is not required that the matrix is square or invertible. Another good property of this factorisation is that, since Q is orthogonal, its matrix inverse is obtained just by transposing it. We use a task-based implementation of the tiled algorithm, using LAPACK as described by Buttari et al. [23], which saves the R matrix and the Householder reflectors (to compute Q) in-place.

Red-Black is the third algorithm for solving the stationary heat diffusion problem. The data decomposition is exactly the same as for Gauss-Seidel, but the task graph is more similar to Jacobi; the red sub-iterations are fully parallel (by tiles) and so are the black sub-iterations. This happens because at each iteration, the red tiles depend on the black tiles of the previous iteration and, similarly, the black tiles depend on the red tiles of the current iteration.

Symmetric matrix inversion (SMI) is used to compute the inverse of a symmetric matrix in a fast way by using a Cholesky factorisation. One use-case for this algorithm is when dealing with some variance matrices. We use the tiled task decomposition of the dense linear algebra version as described by al-Omairy et al. [111], using LAPACK.

Evaluated inputs

Table 3.4 includes a summary of the inputs used in the task-based applications in the two different systems. The inputs are chosen to stress the NUMA behaviour of the system, by using large data sets that do not fit in the caches, and a large number of tasks to stress the parallelism and communications.

Table 3.4.: Summary of inputs for task-based parallel applications

benchmark	**input in SGI Altix UV100**	**input in Atos Bull bullion S**
Conjugate gradient	matrix `Hook_1498.mtx`, divided into 45 blocks	synthetic Poisson 3D matrix, 7-point stencil, $480 \times 480 \times 480$
Gauss-Seidel	6144×6144-element stencil divided into blocks of 512×512 elements	30720×30720-element stencil divided into blocks of 1024×1024 elements
Integral histogram	10 images of 12288×12288 elements divided into blocks of 512×512 elements, sorted into 32 bins	10 images of 32768×32768 elements divided into blocks of 512×512 elements, sorted into 32 bins
Jacobi	15000×15000-element matrix divided into 150 blocks of columns	50000×50000-element matrix divided into 500 blocks of columns
NStream	arrays with $N = 768 \cdot 1024 \cdot 1024$	arrays with $N = 9216 \cdot 1024 \cdot 1024$
QR factorisation	23040×23040 matrix divided into blocks of size 1280×1280	53760×53760 matrix divided into blocks of size 1280×1280
Red-Black	6144×6144-element stencil divided into blocks of 512×512 elements	30720×30720-element stencil divided into blocks of 1024×1024 elements
Symmetric matrix inversion	36000×36000 matrix divided into blocks of size 1800×1800	53760×53760 matrix divided into blocks of size 1280×1280

Chapter 4.

Performance and configuration models for interactions between NUMA and hardware prefetchers

This chapter presents the performance benefits of optimising both NUMA scheduling (threads and data placement) and prefetcher configuration at runtime through careful modelling and online profiling. Optimising NUMA scheduling and prefetcher configurations together leads to a large and complex design space that has previously been impractical to explore at runtime. To address the large design space, we propose a prediction model that reduces the amount of input information needed and the complexity of the prediction required. We do so by selecting a subset of performance counters and application configurations that provide the richest profile information as inputs, and by limiting the output predictions to a subset of configurations that cover most of the performance. Our model is robust and can choose near-optimal NUMA+Prefetcher configurations for applications from only two profile runs. We further demonstrate how to profile online with low overhead.

The contributions in this chapter are:

- Demonstrating that the co-optimisation of NUMA and prefetcher configurations can lead to a 1.77× average speedup over a locality-optimised NUMA baseline with all prefetchers enabled (section 4.3),

but that it requires the impractical evaluation of 288 distinct configurations per parallel region (section 4.2).

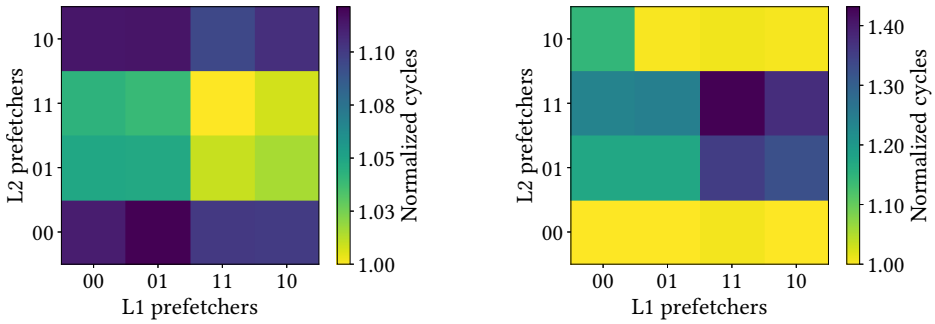
- The design of a prediction model (section 4.4) that requires the evaluation of only 2 distinct configurations and achieves an average of 1.68× (95% of the optimal performance) speedup over a locality-optimised NUMA baseline with all prefetchers enabled (section 4.5).
- A methodology for applying our model at runtime that handles inter-region configuration conflicts (section 4.6).

4.1. The relevance of NUMA and prefetcher configurations in performance

For both NUMA and prefetching, appropriately configuring the system can lead to significant performance gains. NUMA optimisations have been explored extensively [43, 49, 50, 104, 147] and focus on adjusting the thread and data placement across the nodes to minimise latency and maximise bandwidth. These optimisations are typically done via the operating system's memory manager and thread scheduler. Configuring prefetchers via hardware registers [76, 85] improves performance by adjusting where data is prefetched and how aggressively it is fetched to match the application and system cache hierarchy.

However, previous prefetcher studies have assumed fixed NUMA configurations, and, likewise, previous NUMA studies have assumed fixed prefetcher configurations. This leaves open the question as to what benefits can be achieved by co-optimising for both NUMA (thread and data placement) and prefetchers.

To appreciate the complexity of optimising for both aspects simultaneously, consider figure 4.1, which shows how part of the BT benchmark [9] is affected by prefetcher configurations (16 squares) depending on the NUMA configuration (left: all data in one node, right: optimised for locality). As the figure shows, while the middle row of prefetcher configurations improves



(a) All data in one node

(b) Optimised for locality

Figure 4.1.: Normalised cycles (lighter is faster) for BT using two NUMA configurations.

For the `x_solve` codelet from BT, these two heatmaps of normalised cycles show opposite effects of prefetchers depending on NUMA configuration, using 32 contiguous threads. There are 16 prefetcher configurations, set up with four bits (0 enabled, 1 disabled).

performance with the left NUMA configuration, the same prefetcher configurations hurt performance for the right NUMA configuration. The full complexity of this problem is shown in figure 4.2: 16 prefetcher configurations for each of 4 page mappings across 5 combinations of thread or node parallelism, and mapping for 57 parallel benchmark regions. It is clear from this figure that, while the benchmarks exhibit diverse behaviours across the NUMA+Prefetcher configurations, there are clear patterns that we can leverage for efficient optimisation.

To address this large search space we develop models that can choose near-optimal NUMA+Prefetcher configurations for applications. Our models use input profiles (performance counter values) collected by executing parallel regions from the application under a few specific NUMA+Prefetcher configurations. This provides valuable information about the applications (as they are profiled on multiple NUMA+Prefetcher configurations) at low cost (as only a few configurations need to be profiled). We demonstrate through cross-validation that the resulting models are capable of accurately predicting good NUMA+Prefetcher configurations for unseen applications, and we show how they can be gathered online at runtime.

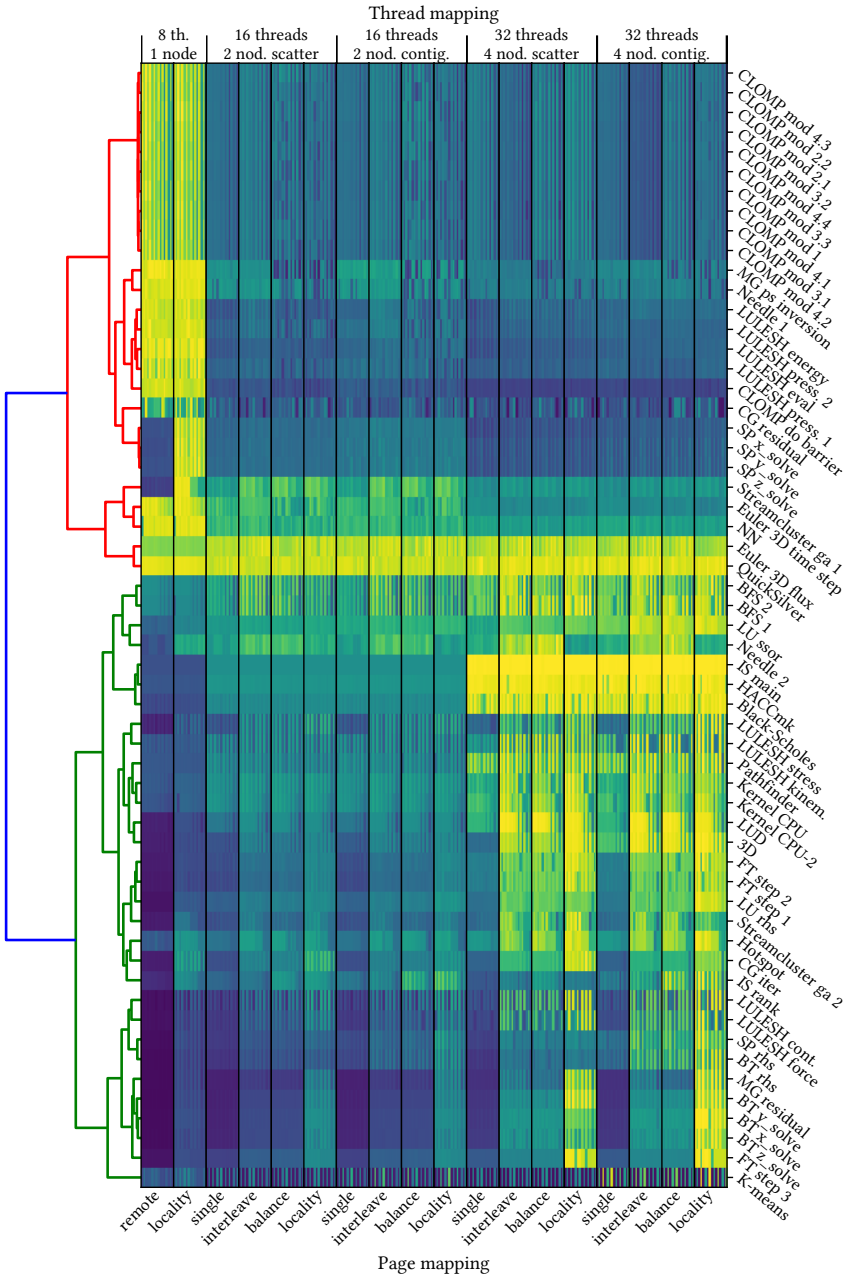


Figure 4.2.: Normalised speedups (lighter is faster) of parallel regions showing complex sensitivities to NUMA+Prefetcher configurations on a Sandy Bridge system.

The regions are clustered according to similar speedup behaviours. Each vertical line on a NUMA configuration (thread and page mapping) is 1 of 16 prefetcher configurations.

4.2. Search space

The main challenge in optimising applications for both NUMA and prefetching carrying out an efficient exploration of the large number of configurations in order to build the models. In this work we consider 18 (or 20, depending on the system) state-of-the-art NUMA optimisations for thread- and page-placement together with the 16 hardware prefetcher configurations available on our system. This leads to a search space of 288 (or 320) NUMA+ Prefetcher (thread+page+prefetching) configurations, which would take over 4 months of execution time to explore directly for our benchmarks. For this reason, we use codelets as presented in section 3.3.1, in order to obtain the performance metrics needed to build the models.

4.2.1. NUMA configurations

We focus on standard fork-join parallel HPC applications, e.g., OpenMP paralleled for-loops, as this results in predictable thread assignments. The applications are those presented in section 3.3.1 The **thread mapping** defines how the application's execution threads are assigned to the hardware cores available in the system[1]. Similarly, the **page mapping** defines how the application's memory pages are assigned to the processor nodes' DRAM.

Except in the case of a single-node system, where all NUMA configurations are equivalent, the different combinations of thread- and page-mapping can give very different results in terms of data locality, performance, and energy consumption for each application.

Thread Mapping

We consider three thread mapping parameters: **degree of parallelism** (number of threads used), **NUMA degree** (number of NUMA nodes used), and **assignment algorithm** (how threads are assigned to cores on the nodes). We consider two thread assignments: contiguous and scattered. Both evenly

[1]We ignore hardware multithreading in our policies and experiments for simplicity.

distribute the threads across the nodes. However, **scattered** uses round-robin to place the threads (e.g., if using 4 threads and 2 nodes, threads 1 and 3 are mapped to the first node and threads 2 and 4 to the second) while **contiguous** places them iteratively (e.g., if using 4 threads and 2 nodes, threads 1 and 2 are mapped to the first node and threads 3 and 4 to the second). For the configurations where we use only a subset of the cores, the remainder are idle. We pin the threads to the specific cores to keep the mappings stable throughout the execution.

Page Mapping

We consider 7 different page mappings, some of which require detailed profiler/programmer information and others which can be applied automatically by the system.

The automatic policies include: **first touch** (each page is allocated on the node that first accesses it), **single node** (all pages are allocated on one single node), and **interleaved** (pages are distributed in a round-robin fashion among the available nodes). We additionally consider two additional policies when all threads are in a single node: **local** (pages are allocated on the same node), and **remote** (pages are allocated on a different node from the execution). Remote mapping is typically a bad configuration because it increases access latency and reduces bandwidth, but is useful for exposing NUMA sensitivity. It is important to note that even though the first touch policy does not require any profiling or support from the programmer, the result of this mapping is highly-dependent on the application code, the thread mapping, and the scheduling algorithm.

The mappings that require detailed profiler/programmer support include: **locality** (each page is allocated in the node of the cores that will access the page the most), and **balance** (pages are spread across the nodes in such a way that the total amount of memory accesses to each node is approximately the same). These mappings require profiling the application's access pattern and implicitly assume that the patterns are reasonably stable across different

runs and inputs, which has been shown to be a fair assumption for these benchmarks [124, 148].

4.2.2. Prefetcher configurations

The hardware prefetcher configurations provided by Intel[2] for post-Nehalem microarchitectures provide 4 bits (16 combinations) to control four prefetchers [41, 63, 94]:

- **DCU IP-correlated prefetcher**: A stride prefetcher that brings data from the L2 into the L1 (data cache unit, or DCU) by correlating prefetches with the instruction pointer (IP, or program counter).
- **DCU prefetcher**: A next-line L1 cache prefetcher.
- **L2 adjacent cache line prefetcher**: For every access it brings the previous or next line (64 B) that completes a memory block aligned to 128 B.
- **L2 streamer prefetcher**: Detects data streams and fetches the next predicted lines to the L2 cache. Similar to the DCU IP-correlated prefetcher, but not using the instruction pointer.

4.3. Characterisation

To understand how we can simplify the search space, we first explore all NUMA+Prefetcher configurations via codelet execution. This brute-force exploration allows us to identify common behaviours across codes and configurations, which we can then use to build efficient models for choosing the best configuration.

[2] See https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors

4.3.1. Experimental setup

For our experiments we use two machines: a four-node Intel Sandy Bridge EP E5-4650 with 128 GB of RAM and a dual-node Intel Skylake Platinum 8168 with 188 GB of RAM.

All speedups presented in this chapter are against an execution using all cores (32 for the Sandy Bridge and 48 for the Skylake), scattered among all nodes, and a locality-optimised page mapping with all prefetchers on. This is an optimised configuration which tries to increase the bandwidth (scattered thread mapping and locality page mapping) and reduce the latency (locality page mapping and prefetcher activation) over a simple first touch policy.

4.3.2. Performance opportunities

We first look at the overall speedups that can be obtained from optimising NUMA and/or prefetching in table 4.1. NUMA optimisations alone (1.66× on average between Sandy Bridge and Skylake) are more significant than prefetcher optimisations alone (1.19× on average). The reduced NUMA sensitivity on Skylake is likely due to its faster interconnection links between nodes. Moreover, Sandy Bridge has more nodes, further increasing the severity of NUMA effects. Interestingly, the benefit from optimising prefetchers is the same for both systems.

A greedy optimisation for NUMA and prefetching (e.g., choosing one first, then picking the best choice for the second) delivers still better performance, but the best order depends on the system. Exploring all combinations of NUMA and prefetcher configurations (i.e., a coupled optimisation) delivers slightly higher performance of 1.77× vs. 1.72× for the greedy NUMA-first optimisation (on average). However, the majority of this benefit comes from one benchmark, K-means, which is able to find a particularly efficient configuration with the coupled optimisation.

Figure 4.2 shows the per-region (rows) speedups (lighter is faster), across all NUMA and prefetcher configurations (columns), for Sandy Bridge. The

Table 4.1.: Comparison of speedups between different optimisation searches against an optimised default (pages: locality, threads: scatter, prefetchers: on).

	Speedup (geometric mean)		
Optimisation	Sandy Bridge	Skylake	Average
Only NUMA	1.73	1.59	1.66
Only prefetchers	1.19	1.19	1.19
First NUMA, then prefetchers	1.78	1.66	1.72
First prefetchers, then NUMA	1.75	1.67	1.71
Coupled search	1.82	1.73	1.77

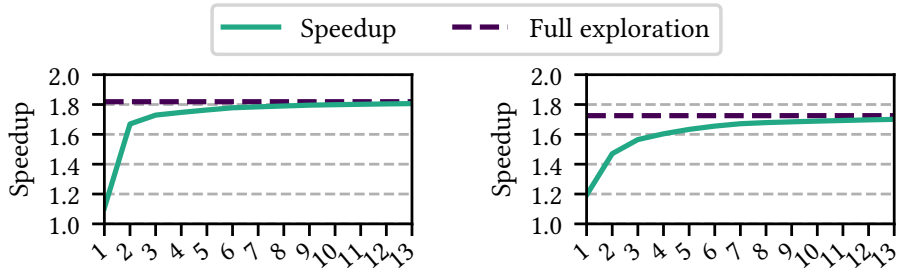
results for Skylake follow a very similar structure. On the left side, the regions have been clustered using Ward's method on the normalised speedup vectors, showing that many regions share very similar speedup patterns across the configurations.

This representation allows us to see which regions benefit from similar NUMA+Prefetcher optimisations. For instance, BT (`x_solve`, `y_solve` and `z_solve`) has very similar behaviour to MG residual, as they have similar access patterns (when the pages are appropriately mapped). As a result, they are clustered together. Similarly, all CLOMP regions, except the barrier, show similar sensitivities and are grouped together (figure 4.2, top 10 rows).

Our clustering shows that many benchmarks share common behaviours, suggesting that clustering behaviours or optimisations may be effective.

4.3.3. NUMA+Prefetcher configuration diversity

Figure 4.2 showed that there is significant *similarity in region behaviour* across the NUMA+Prefetcher configuration space. In figure 4.3 we explore the similarity of optimised NUMA+Prefetcher configurations across the parallel regions. This figure shows the speedup we can achieve with a limited number of configurations compared to the full exploration. Here we see that, by only allowing the subset of the 11 best NUMA+Prefetcher configurations, we can achieve over 98 % of the maximum speedup, and when considering the best 13, we can achieve 99 %. These results come about for two reasons:



(a) Intel Sandy Bridge

(b) Intel Skylake

Figure 4.3.: Maximum attainable speedup with respect to the number of configurations.

the first is that regions have similar behaviours, as seen previously, and the second is that in many cases different NUMA+Prefetcher configurations give essentially the same performance benefit.

4.3.4. Takeaway

Our analysis shows that many benchmarks behave in similar ways and that we can achieve nearly all of the speedup potential with a very small set of NUMA+Prefetcher configurations. This suggests that it will be possible to build a model that can recognise application behaviours (since there is a limited number of them) and predict very good NUMA+Prefetcher configurations (as only a few are needed to cover most of the benefit).

4.4. Prediction model

A brute-force approach to optimising NUMA+Prefetcher configurations would take as input the performance of all possible configurations and choose the best one as output. This guarantees the best performance but comes with the very high overhead of evaluating all configurations. To address this overhead, we train a *prediction model* that takes far fewer configurations as input but can still choose among a large enough subset

of all possible NUMA+Prefetcher configurations as output to achieve good performance.

Building an effective model requires co-designing the subset of the input configurations to evaluate and the output configurations to choose from while training the model to accurately map between them. The subset of inputs reduces the profiling overhead but still allows us to observe application differences, while the subset of output configurations allows us to simplify the model, while still obtaining high performance.

4.4.1. Machine learning models

We use the Python scikit-learn [118] package to train multiple types of models (Artificial Neural Network (ANN), Logistic Regression (LR), Decision Tree (Tree), Support Vector Machine (SVM), and Clustering) using the parameters in table 4.2. From section 4.3 we saw that:

1. Only 13 NUMA+Prefetcher output configurations are needed to cover 99% of the potential performance gains (section 4.3.3).

2. Many codes behave in a similar way across different NUMA+Prefetcher optimisations (section 4.3.2).

These two observations guide our training of different types of models. To take advantage of 1), we use supervised learning (ANN, LR, SVM, Tree) to directly predict from among only the 13 overall most efficient configurations, instead of across all possible configurations (288 for Sandy Bridge and 320 for Skylake). We also train Tree and ANN models using multi-labels: all configurations that perform within 95 % of the best configuration are labelled as best, instead of just labelling the best one.

To take advantage of 2), we use unsupervised clustering to group regions by similar optimisation choices. Unlike supervised learning, clustering cannot directly assign a configuration to a code. Therefore, we select the centroid of each cluster (in the feature space) and use its configuration across all the other regions within the same cluster. We expect efficient features to

Table 4.2.: Prediction model parameters. Single/multi-label models use the same parameters.

Model	Parameter
ANN	lbfgs, alpha=0.0001, hidden_layer_sizes=(7,)
Tree	—
SVM	gamma=scale, decision_function_shape=ovo
LR	random_state=0, solver=lbfgs, multi_class=multinomial
Clustering	Hierarchical Ward (Euclidean distance)

gather together regions that share the same optimal configuration. During validation we measure the features of the new regions and assign them to an existing cluster, and use its centroid-selected configuration.

Finally, we note that some models take much more time to train than others. For example, creating a single-label Tree model is 100× faster than a multi-labelled ANN. This directly affects the number of input feature pairs we can explore in training.

4.4.2. Model generation and inputs

As shown in figure 4.4, the models we train take as input hardware performance counter values for a region executed with different NUMA+Prefetcher configurations and predict the best NUMA+Prefetcher configuration for that region. We identify the best configuration (correct prediction) through brute-force measurement (a) of the execution time for all NUMA+Prefetcher configurations for each region (see section 4.2). Our training input features consist of hardware performance counter measurements for each region for all NUMA+Prefetcher configurations (b). With these input features (c) and the correct prediction data (d), we can train a variety of models (e) to predict the best NUMA+Prefetcher configuration[3].

[3]We do not consider first touch as a possible page mapping because it assigns pages based on how the developer designs the first accesses rather than on fixed rules. Therefore, first touch page allocation strategy differs across applications, making it an inconsistent choice for our model.

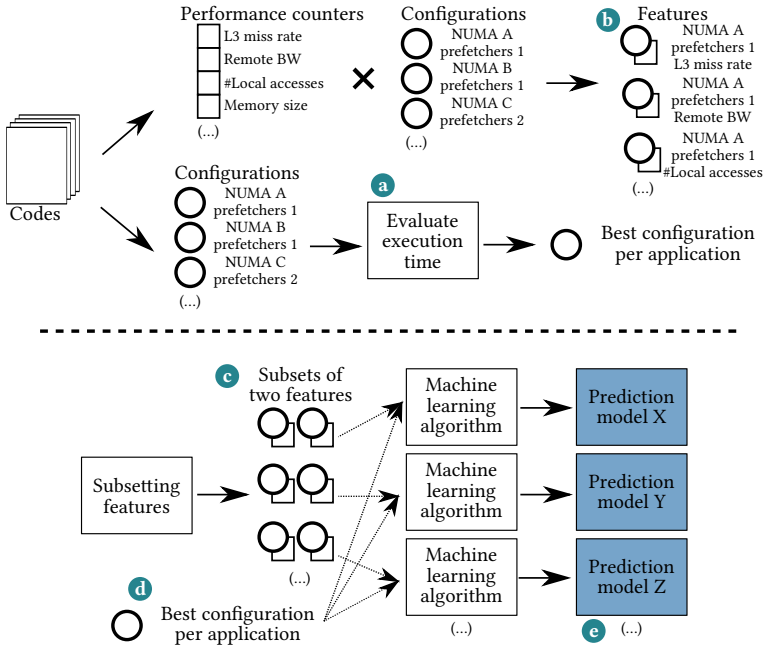


Figure 4.4.: Diagram showing how the model training scheme works.

Top: Training data collection, including brute-force evaluation of execution time for all configurations. Bottom: Training each machine learning method across multiple subsets of the input features to identify the best combination of input features and model.

Model inputs

Hardware performance counters provide precise information on the interaction between the application and system. However, while these metrics are valuable for characterising applications, they are not standard across systems. To address this, we use Likwid 3.0 [149] to abstract them to higher-level *performance groups*. We select memory-system related measurements, resulting in 19 performance counters from the Likwid NUMA, L2 CACHE, and L3 CACHE groups, as well as energy/power measurements from RAPL [44].

However, using only 19 data points for each application obtained with a single NUMA+Prefetcher configuration is not sufficient to train an accurate model (see analysis in Section 4.5.3). We therefore collect performance counter data for *all NUMA+Prefetcher configurations*, which increases our training set by a factor of 288. This is particularly valuable as we observe that

the same performance counter profiled with different NUMA+Prefetcher configurations can return significantly different values, giving us information on the impact of changing the NUMA+Prefetcher configuration.

This technique of amplifying application data by changing the execution environment and observing the reaction is inspired by previous work in compilers [33, 156]. That work measured execution times across different compiler settings and built models to predict configurations based on the programs' reactions. We extend this concept by considering more diverse metrics given by the hardware performance counters (e.g., local accesses, bandwidth). This allows us to feed our prediction model with more diverse information to improve its accuracy. We call the resulting features **reaction-based performance counters** as they show the reaction of hardware performance counters to NUMA+Prefetcher configurations.

Unfortunately, while using reaction-based performance counters as input features to our models increases our data for training, collecting them can require up to 5472 executions for each region (19 counters × 288 NUMA+Prefetcher configurations)[4] to measure the features and another 288 per code for measuring the performance. However, these executions are a one-time cost for generating training data for the model.

We can reduce the need for profiling by using the features from one system to train models for another system. For instance, we can use the reaction-based performance counter input features from Sandy Bridge together with the ground truth (best configuration execution time) from Skylake to develop a model for Skylake. This reduces the overhead to only the 288 performance measurements to train the Skylake model. We demonstrate the accuracy of this cross-training in section 4.5.1.

[4] On Intel machines this is reduced to 1440 executions as Likwid and RAPL measure multiple counters at the same time. For codelets we require an initial warm-up execution prior to the profiling execution [109].

Feature selection and generation (subsetting features)

To reduce the input needed for the model, we train models using only a *subset of the input features* for each region, specifically, two sets of performance counters and NUMA+Prefetcher configurations[5]. This allows us to benefit from the additional information provided by the reaction-based performance counters while keeping the profiling cost low: only two runs are required. To identify which subset of two features is most effective (i.e., contains helpful information for choosing configurations), we train models for many subsets of 2 input features and select the most efficient one. For each machine learning method, choosing the best model requires training for 20.4 million subsets[6]. As part of training, we use a standard 10-fold cross-validation (section 4.5.1) to validate our models' robustness.

4.5. Prediction results

We now evaluate the ability of our models to predict configurations for new regions. We use the same system setup and regions presented in section 4.3.1: we collect the reaction-based performance counters (model input features) on Sandy Bridge and do a brute-force execution time exploration for all configurations on both Sandy Bridge and Skylake. With this data (input features and execution times), we train models for predicting NUMA+Prefetcher configurations for Sandy Bridge and Skylake. We then evaluate the resulting predictions on both systems vs. the ground-truth brute-force execution time exploration.

[5] We empirically observed that two features provide good results from our models. Adding more features may achieve higher accuracy but causes a combinatorial explosion of the exploration space. We tried to only use subsets of many features collected within a single NUMA+Prefetcher configuration, but did not achieve good results.

[6] The exploration considers 21 NUMA configurations instead of 18, separating single, balance and interleave in 1-node settings, giving a total of $21 \cdot 16 = 336$ NUMA+Prefetcher configurations, with 19 counters giving $336 \cdot 19 = 6384$ total features. This results in $\binom{6384}{2} \approx 20.4\,\text{M}$ subsets of 2 features.

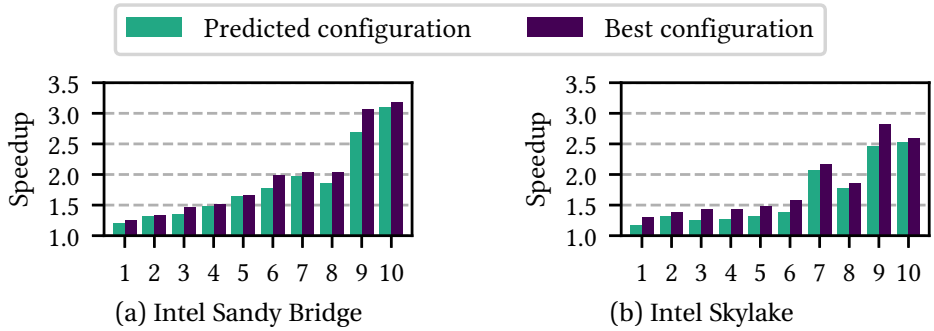


Figure 4.5.: 10-fold cross-validation of the predicted results.

This validation shows robust prediction across unseen regions and close to optimal predictions. Skylake results are trained with Sandy Bridge profiles. See also feature selection in section 4.4.2.

4.5.1. Model evaluation

In this section, we evaluate the performance of the predicted configurations for each model and validate that the model gives good results.

Model validation

Each model takes as input two features and provides a prediction of the best configuration. To evaluate the quality of the models, we quantify the performance loss between the model-predicted configuration and the brute-force best configuration on *unseen codes* with cross-validation. Cross-validation shuffles all codes and splits them into groups (folds) of similar size. Each fold is then separately used as validation set for the model trained over remaining codes. If the training accuracy is consistently high across the different folds, then it indicates that the model is able to effectively generalise to unseen codes.

Cross-validation

We illustrate in this section how cross-validation evaluates a model. We take as an example our best model for both Sandy Bridge and Skylake, Tree single-label (see table 4.3).

Figure 4.5a presents the cross-validation for Sandy Bridge. It shows the results of training the model on the benchmarks in the other folds and then using the resulting model to predict the best configuration for the remaining fold. Here all but 2 folds (6 and 9) show small differences between the model prediction and the brute-force best choice, indicating that the model has generalised well. The comparison of the predicted configuration to the best configuration (dark) shows the model is effective, obtaining 95 % of the possible speedup. The reason why folds 6 and 9 show worse prediction results is because CG residual and K-means, respectively, have distinct behaviours (see figure 4.2), and they were not included in the benchmarks used for training in these cases. However, the predicted configuration is still equivalent to or better than just optimising for NUMA (page and thread mapping), and we expect that these mispredictions would be addressed by training on more codes.

Figure 4.5b shows the validation of the model which predicts configurations on Skylake based on performance counters collected from Sandy Bridge: i.e., the two input features are from reaction-based performance counters on Sandy Bridge, with only the output configuration performance measured on Skylake. This model has higher variability between the folds and the best configuration, indicating the model is less-well generalised, and slightly less effective (92 % of the possible speedup). This is likely due to Skylake-specific behaviour that is not visible in the Sandy Bridge performance counters used for training, and shows the trade-off for reducing the training overhead by reusing the Sandy Bridge input data.

Our models' geometric mean performance gains, over a locality-optimised baseline with all prefetchers enabled, are 1.76× (Sandy Bridge) and 1.60× (Skylake) across all the folds. This shows that our best models provide significant speedups while remaining robust across new unseen benchmarks.

Table 4.3.: Best model parameters for each ML method, including the reaction-based performance counters selected as the two profile inputs. (Thread mappings: remote, locality, single, interleave, balance, contiguous. Number of model evaluations in the 2-day training period.) While the DRAM Power performance counter was selected for many of the models, the second performance counter (and the NUMA+Prefetcher configurations) varied significantly.

System	ML method	First feature (best model)						Second feature (best model)						Model speedup	# eval. models
		Perf. counter	HW pf.	NUMA configuration				Perf. counter	HW pf.	NUMA configuration					
				# th.	# nodes	Th. map.	Page map.			# th.	# nodes	Th. map.	Page map.		
Sandy Bridge	**Tree_s**	**Package pow.**	**1111**	8	1	**—**	**local**	**Package pow.**	**1011**	16	2	**contig.**	**balance**	1.76	20 368 k
	LR_s	DRAM pow.	0110	8	1	—	local	Package pow.	1011	32	4	scatter	locality	1.43	3349 k
	Clustering	DRAM pow.	0001	8	1	—	remote	DRAM pow.	1101	8	1	—	remote	1.56	641 k
	ANN_m	DRAM pow.	1011	8	1	—	remote	Rem. DRAM BW	1011	16	2	scatter	interl.	1.65	146 k
	ANN_s	DRAM pow.	0101	8	1	—	local	Energy	0001	32	4	scatter	interl.	1.66	284 k
	SVM_s	Core pow.	0100	8	1	—	local	Core pow.	1010	8	1	—	local	1.70	20 043 k
	Tree_m	Energy	1100	16	2	scatter	single	Package pow.	1010	32	4	contig.	locality	1.74	11 075 k
Skylake	**Tree_s**	**Package pow.**	**1101**	16	2	**contig.**	**locality**	**L3 miss ratio**	**1110**	16	2	**scatter**	**single**	1.60	20 368 k
	LR_s	DRAM pow.	1101	8	1	—	remote	Package pow.	1011	8	1	—	local	1.40	3349 k
	Clustering	DRAM Pow.	1001	8	1	—	local	L2 miss ratio	1111	8	1	—	local	1.40	643 k
	ANN_m	DRAM pow.	1100	8	1	—	local	L2 miss ratio	0110	32	4	contig.	locality	1.51	144 k
	ANN_s	DRAM pow.	0111	8	1	—	remote	DRAM pow.	1101	32	4	scatter	locality	1.53	293 k
	SVM_s	DRAM pow.	1001	8	1	—	remote	Loc. DRAM BW	1001	16	2	scatter	balance	1.52	20 236 k
	Tree_m	Energy	1111	16	2	contig.	interl.	Rem. DRAM vol.	1010	8	1	—	local	1.59	11 094 k

For Skylake, the selected input features are **L3 Miss Ratio** and **Package Power**, but with very different prefetcher and NUMA configurations (see table 4.3). Package Power measures the power consumption of an entire node, including cores, last level cache, and memory controller. It is interesting to note that for Sandy Bridge the selected input features also measure Package Power, but profile it on two very different configurations (i.e., different prefetchers, core counts or nodes). In other words, the performance change across configurations is a useful information to guide the configuration prediction. This illustrates how the reaction-based performance counters allow us to include sensitivities to system configurations as model inputs.

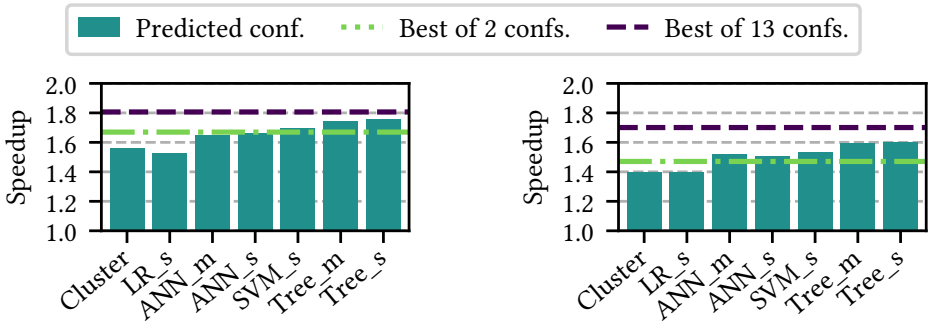
4.5.2. Comparing machine learning methods

The various machine learning methods give different models with different performance. Moreover, some of the models using some methods are faster to evaluate than using other methods and, given a time budget, it is possible to choose among more models.

Performance gains

We evaluate the 7 different machine learning methods described in section 4.4.1. For each method and system, figure 4.6 reports the geometric mean performance gain across all folds of the most efficient model.

For performance analysis we provide two baselines that use our brute-force evaluation to pick the best configuration across subsets of the whole NUMA+Prefetcher search space. We define *best of 2* and *best of 13* as optimisation strategies to compare with. Best of 13 selects the best optimisation for each region from the 13 output NUMA+Prefetcher configurations that our models choose among. As illustrated in figure 4.3, best of 13 provides similar performance to a full brute-force exploration and is therefore used as oracle. Best of 2 picks the best configuration from only the two most efficient overall configurations. That is, best of 2 has the same input overhead as our models:



(a) Intel Sandy Bridge

(b) Intel Skylake

Figure 4.6.: Geometric mean performance gains of the most effective prediction model for each Machine Learning method. _s/_m refer to single label/many-labelled training.

the user runs two configurations and picks the best one. This allows us to separate out the contribution of the model from the choice of model inputs.

Figure 4.6 shows that there is a significant difference across the the models: 1.2× between the best (Tree) and worst (Logistic Regression). However, the methods show similar relative results across the systems. In both cases, Tree (both single and many-labelled) gives the most efficient model while Logistic Regression is the worst. Finally, SVM and Tree outperform the best of 2 brute-force approach on both systems, demonstrating our model capabilities.

Understanding why some methods are more efficient

To understand why Tree is the most efficient method, we need to describe in details how we train. Each method iterates over the subsets of 2 input features, trains a model for those input features, and then does a cross-validation for the resulting model. For each method, the best performing set of input features is chosen. This process takes longer for methods that are slower to train, which therefore limits the number of input feature subsets that can be evaluated. For each method we allocated a budget of 2 days of training per system. This limits the number of input subsets explored for the methods that take longer to train. table 4.3 shows the size of this effect.

We observe that Tree_s was able to evaluate the whole search space of input subset features, thus exploring 100× more input subsets than ANN many-labelled in the 2 day training time. As a result, it is possible that the more time-consuming methods did not have a chance to evaluate some particularly good combinations of performance counters. To partially mitigate this, we explored over 10 000 random subsets, as well as trying the best input subset from Tree_s with the ANN. Unfortunately, these approaches did not improve the performance, suggesting that *input features need to be selected together with the method for best efficiency*. This is particularly interesting as we see that some performance counters such as DRAM Power (included in 5 out of 7 models) carry valuable information, but are not used in Tree_s.

We conclude that Tree is not necessarily the best method, but its combination of faster training time and good prediction allows it to beat other models that may be more accurate, but take longer to train. Therefore, investing more time to train more time consuming methods such as ANN may further improve the gains[7].

4.5.3. Reaction-based performance counters improve modelling

In the previous results, our models were all trained using the reaction-based performance counters (e.g., inputs were a performance counter measured on a specific NUMA+Prefetcher configuration). To quantify the value of this measuring the performance counter on the default NUMA+Prefetcher configuration, we compare our results with models that are only allowed to choose their 2 input features from a single NUMA+Prefetcher configuration.

Single-configuration profiling drastically reduces the number of model inputs to only $19 \cdot 19 = 361$. As a result, each machine learning model was able to evaluate all possible 2-input feature subsets. Figure 4.7 compares single-configuration-trained models[8] to our reaction-based performance

[7] We did not consider Deep Neural Networks due to our small input training set of only 57 parallel regions.

[8] Single-configuration for Sandy Bridge: 32 threads, 4 nodes, scatter, locality, and all prefetchers on.

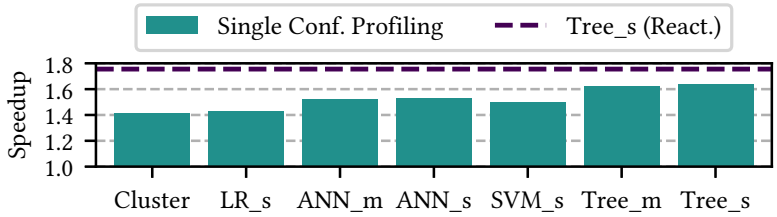


Figure 4.7.: Single-configuration profiling vs. the best Reaction-based performance counter approach (Tree_s React.).

counter models. We see that training on only a single configuration significantly hurts performance (down to 93 % of the Reaction-Based Performance Counter approach for Tree_s), even thought it is able to explore all possible input combinations. This showcases that the overhead of having to evaluate codes on multiple configurations pays off in better modelling, which leads to better performance.

4.5.4. Takeaway

We have shown that our Tree-based model can *robustly* predict combined NUMA+Prefetcher configurations that deliver a *geometric mean speedup of 1.74*× (compared to 1.82× for an oracle), even against a locality-optimised baseline with all prefetchers enabled. We observe that the Tree model wins out due to combination of faster training (which allows greater exploration of input combinations) and good prediction accuracy. Further, we have shown that the reaction-based performance counters provide significant benefits by exposing sensitivity to system configurations.

In this study we also observed that the Package Power performance counter appears to provide the best indication of overall configuration fitness, as it is chosen as an input to the best-performing machine learning models. This choice is unexpected, as IPC (instructions per cycle) or MPKI (misses per thousand instructions) are typically chosen to represent overall performance. Our analysis is that Package Power performs better as it takes into account both CPU activity (as IPC does) and cache behaviour (as MPKI does), as well as CPU idle times, link utilisation, and DRAM accesses, making it a robust overall metric.

Code region	A		A	A		A	A		A
Exec. step	Address Trace	Migration conf. 1	Warmup conf. 1	Profile conf. 1	Migration conf. 2	Warmup conf. 2	Profile conf. 2	*Migration conf. A*	Optimized execution

(a) Obtaining the best configuration of a region with a single run.

Code region		A	B	A	B		A	B	A	B		A
Exec. step	Migration conf. A	Warmup conf. A	Warmup conf. A	Profile conf. A	Profile conf. A	Migration conf. B	Warmup conf. B	Warmup conf. B	Profile conf. B	Profile conf. B	*Migration best conf.*	Optimized execution

(b) Optimising whole applications based on the per-region configurations. A and B conflicts are addressed by cross-evaluating their configurations and selecting the best one overall.

Figure 4.8.: Online evaluation of different configurations.

4.6. Optimising applications online

4.6.1. Online profiling and optimisation

The method described up until now makes predictions using information from offline profiling: first, extract codelets of the OpenMP regions and execute them with the two selected NUMA+Prefetcher configurations while recording the appropriate hardware performance counters, then use the model to predict the best NUMA+Prefetcher configuration. However, as the application itself consists of repeated executions of the OpenMP regions, we can move this profiling online by carefully setting the NUMA+Prefetcher configurations between the OpenMP region executions as the application runs and profiling online. Doing so allows us to collect the required input information online with only the overhead of changing the configuration for the two regions and appropriate warm-up.

Figure 4.8a illustrates the online profiling of OpenMP region A as the application executes. The first instance of region A is used to collect the access patterns (Address Trace) needed for the NUMA locality and balance policies[9]. We then need to execute the region for each of the two input NUMA+Prefetcher configurations required for the model (conf. 1 and conf 2. in the

[9]The overhead of collecting this information can be reduced to 12 % [148], but in this work we use Pin directly, which is around 10× slower.

Figure). However, this incurs two sources of overhead: First, we may need to migrate threads and pages if the current configuration does not match the configuration we need to measure. And, second, we execute the OpenMP region once before measuring to warm-up the caches, and this execution may experience significant cache misses due to the migration. After that setup, we profile the next execution of the region (Profile) and use the model to predict the best configuration. With the model prediction, we migrate pages and threads as needed (Migration), and then execute with the chosen configuration (Optimised execution). This approach assumes that parallel regions have similar behaviour, which is quite common in our benchmark applications [124, 148].

4.6.2. Whole-application optimisation

While our prediction model finds the most efficient configuration for each parallel region, it does not consider how regions interact. This is a problem if the NUMA page optimisations for one region hurt the performance of another region as explicit page migration between regions is generally too costly [98]. Prefetching and thread configurations can be changed with little overhead between regions, making them less problematic.

Such inter-region configuration conflicts have been shown to have a small impact on overall performance[10]. Popov, Jimborean and Black-Schaffer [125] reported that only a few applications in the NAS and Rodinia benchmarks, such as BT ans SP, are significantly affected, and the overall reduction in speedup when accounting for inter-region conflicts was only 6%. We therefore evaluate the impact of inter-region conflicts on our online profiling approach for both BT and SP.

Our approach is similar to the one proposed by Popov, Jimborean and Black-Schaffer [125]. First, we select the best configuration for each region individually. We then check the configurations for conflicts (i.e., pages mapped to different nodes). If conflicts are found, we *cross-evaluate* the

[10] Speedups reported so far are for regions and do not include these effects. In this section we evaluate two applications where these effects are significant.

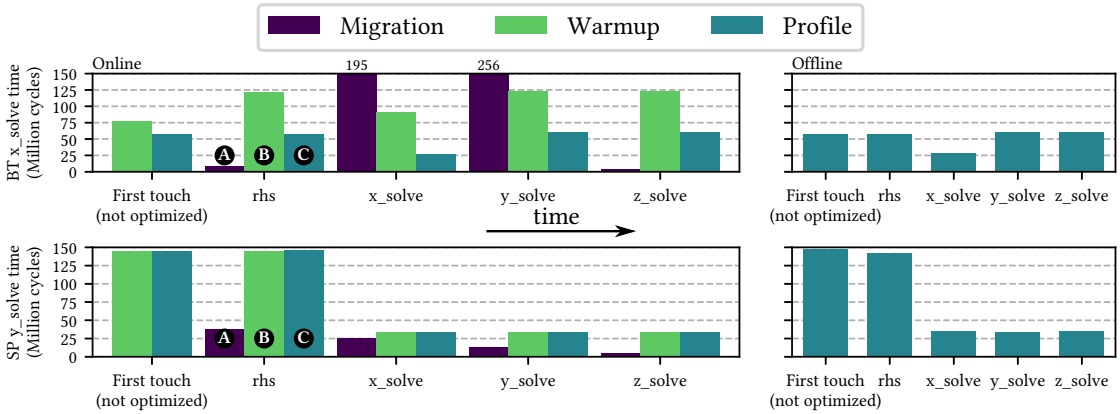


Figure 4.9.: Overheads and execution times for online and offline profiling.

Left: Overhead costs of online cross-evaluation of the parallel regions `x_solve` (top) and `y_solve` (bottom) across the best individually chosen configurations for regions `rhs`, `x_solve`, `y_solve`, and `z_solve`. Cross-evaluation requires first migration **A**, then warm-up **B**, and finally profiling **C** for each configuration. Right: Average execution time for the region when optimised for each configuration offline. Similar online/offline results show the accuracy of online profiling.

configurations of the conflicting regions and select the best overall performing configuration. Figure 4.8b shows the phases of this cross-evaluation: migration, warming, and profiling A and B with A's configuration (left) and, similarly, migration, warming and profiling A and B with B's configuration (right).

The cost and accuracy of online cross-evaluation is shown in figure 4.9 for the `x_solve` region in BT (top) and the `y_solve` region in SP (bottom). The figures show cycle counts for each step in the cross-evaluation of those two regions. From left to right: the region is first executed twice with first touch to observe the non-optimised configuration behaviour, followed by migration/warmup/execution of the region with the configurations for the `rhs`, `x_solve`, `y_solve` and `z_solve` regions. For example, **A**/**B**/**C** in the top figure show **A** the cycles for migrating pages for the `rhs` NUMA+Prefetcher configuration, **B** the cycles for executing the `x_solve` region with the `rhs` configuration for cache warm-up, and **C** the cycles for the profiling of the `x_solve` region with the `rhs` configuration. The accuracy of the resulting online profile can be seen by comparing the online measured execution

Table 4.4.: Execution times (billions of cycles) of BT and SP region conflicts and online profiling. The overhead is quickly amortised since each region is called hundreds of times.

	Locality, 32 threads, scatter (baseline)	Ignoring inter-region	Offline best inter-region	Online best inter-region
BT	33.5	16.3 (2.1×)	22.9 (1.5×)	23.9 (1.4×)
SP	179.8	45.2 (4.0×)	53.5 (3.4×)	54.9 (3.3×)

time for the region with the different configurations to the offline profiled one (e.g., for the `rhs` configuration, this is comparing the online measured time in blue, left, to the offline measured time, blue, right).

Figure 4.9 shows that there are many more cycles spent in migration for BT (top) than SP (bottom). This is because BT's `x_solve` region has a different access pattern from its `y_solve` and `z_solve` regions [98], causing more pages to be migrated. Conversely, the three regions from SP all use a single NUMA node optimisation, significantly reducing the migration cost.

For online profiling/optimisation to be effective, the overhead needs to be less than the benefits. Our approach has three sources of slowdown: migration, warm-up, and profiling[11]. These overheads are only paid once at the beginning of the application and we find they are quickly amortised by the speedups obtained.

Table 4.4 shows the execution time of the applications BT and SP with four configurations: our locality-optimised baseline applied to the whole application, ignoring inter-region conflicts and using each region's best configuration, the best overall configuration found offline, and the best overall configuration found online, including profiling overhead. For both applications there are enough page conflicts that the cost of migrating pages between regions to use each region's independently optimal configuration is prohibitive. However, our online approach finds the overall best configuration and delivers 95 % of the offline performance.

[11] Note that the cross-evaluation profiling itself will incur slowdowns if the configurations being cross-evaluated are a poor match for the region being evaluated. E.g., `y_solve` takes over three times as long to complete when being profiled with the `rhs` configuration (figure 4.9 bottom, "Profile rhs").

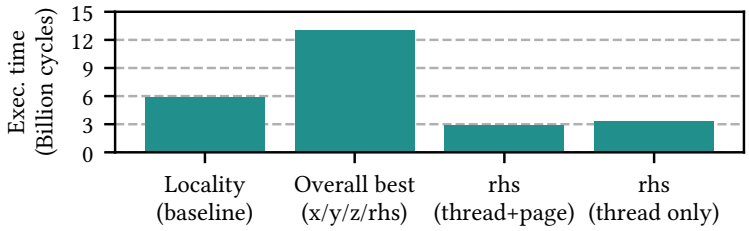


Figure 4.10.: Execution cycles for `rhs` from SP.

Overall best is the best configuration for the whole application, *thread only* uses the `rhs` optimal thread configuration with the overall best page mapping, and is only slightly slower than the combination `rhs` optimal thread- and page-mapping, but can be achieved without the overhead of migrating pages.

4.6.3. Per-region NUMA optimisation

The overhead of page migration makes it too expensive to adapt on a per-region basis. However, it can be profitable to change the thread mapping on a per-region basis, once the overall best page mapping has been chosen. As an example, our method identifies a case in the SP benchmark where different regions should have different degrees of parallelism. This is shown in figure 4.10, for the case of the `rhs` region when executed on the Sandy Bridge system.

In this case, the best configuration for `rhs` uses 32 threads and maps the pages across the whole system, while `x_solve`, `y_solve`, and `z_solve` optimally use only 8 threads and map all pages to just one node. Changing the page mapping between these regions costs more than the potential gains, but the reconfiguration overhead of changing the thread mapping is negligible. By reconfiguring threads, but not pages, we can avoid the costly page migration while retaining many of the benefits from the better thread mapping (3.9× improvement from changing just the thread-mapping on a per-region basis vs. 4.4× for changing both thread- and page-mapping on a per-region basis, compared to the overall best all-region configuration), which also outperforms the baseline locality-optimised mapping. The end result is that the page mapping is stable throughout the execution of the application while we change the thread mapping on a per-region basis, at a negligible cost.

A more sophisticated approach would be to consider which subsets of threads, pages, and prefetchers should be optimised across conflicting configurations, but we leave that for future work.

4.7. Summary

In this chapter we have shown that there is a significant performance benefit from optimising the NUMA configuration (parallelism, thread-, and page-placement) together with hardware prefetcher configurations (L1, L2). However, this benefit comes at the cost of a very large design space to explore. We tackled this problem by developing an efficient and robust performance model. We reduce the overhead of collecting data for the model by identifying two reaction-based performance counter configurations (combinations of NUMA+Prefetcher and performance counters) which allow our model to accurately predict the best configuration, and reducing the number of configurations the model has to choose among by analysing how we can reduce the configuration space without losing performance.

During training, we saw the importance of selecting the correct reaction-based performance counters as inputs for each specific model type and system and how we can do more efficient cross-system training by reusing input data. We then demonstrated how this approach can be applied for online profiling and optimisation to deliver an average of 1.68× performance increase over a NUMA-locality-optimised baseline with all prefetchers enabled. Finally, we observed that in rare cases applications can suffer from inter-region page-mapping conflicts. Using the best overall configuration and changing parallelism across regions partly overcomes the performance loss and improves over the already-optimised baseline.

Chapter 5.

Hardware prefetching for NUMA systems

5.1. Introduction

This chapter presents a generic hardware prefetching scheme that leverages the NUMA characteristics to enhance system performance. As shown in chapter 4, using adequate configurations for prefetchers and NUMA simultaneously is needed in order to achieve the highest performance possible. Just considering the optimisation of prefetcher configuration ahs shown to give worse results than tackling NUMA scheduling or both simultaneously. However, prefetchers are generally unaware of NUMA effects in the system, and adapting them to consider these aspects could provide higher performance improvements.

In this contribution, the knowledge about NUMA in the system is used to prefetch data more aggressively depending on the physical location of the predicted accesses. The extra latency when accessing non-local data is hidden by prefetching this data to the higher levels of cache, with more capacity. The prefetching scheme is evaluated using gem5, a cycle-accurate architectural simulator, in a two-socket NUMA system, proposing a simple stride prefetcher that is aware of NUMA effects. Finally, we show that the ideas are really generic and can be applied to other prefetchers.

All things considered, we propose a NUMA-aware hardware prefetching scheme with the main goal of making all accesses as if they were local. The contributions in this chapter are the following:

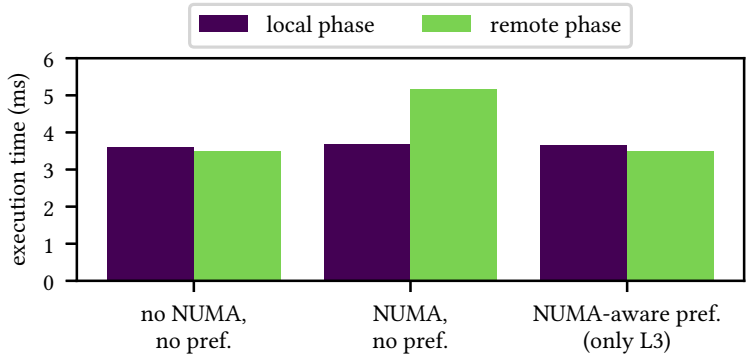


Figure 5.1.: Execution times of a microbenchmark with local and remote accesses under different configurations.

- A generic NUMA-aware hardware prefetching scheme. This scheme is compatible with any L1 hardware prefetching algorithm that can generate a list of predicted addresses. The main implementation is the *NUMA-aware stride prefetcher* (NASP), a stride prefetcher in the L1 that fetches some lines in the L3 depending on the physical location of the predicted addresses (and when they are expected to be used).

- An evaluation of NASP, with 1.10× performance speedup with respect to a standard stride prefetcher in the L1 and 1.02× over a state-of-the-art prefetcher in a 4 socket system. The evaluation includes also details about the communications and cost.

- A comparison with different prefetchers from the state of the art and a NUMA-aware implementation of one of them, using the ideas from NASP. The evaluated prefetcher with NUMA-aware changes has an average speedup of 1.04× with respect to the original implementation.

- Incidentally, an implementation of NUMA using gem5 with the classic memory mode.

5.2. Motivation

5.2.1. Background on hardware prefetchers

The latency gap between CPUs and memory has increased as CPUs have become faster at a higher rate than memory. This been overcome with more complex memory hierarchies, with multiple levels of cache memories that store a small subset of the data to make accesses faster. To take advantage of the lower latencies obtained by accessing the cache instead of main memory, the data must be in the cache. This can happen both with data reuse (temporal locality) or by bringing data into the cache in advance.

Hardware prefetchers are structures that serve as a solution for bringing data into the caches. These structures model a heuristic that, given a memory access, predicts a new address that will be accessed later. A simple heuristic is the next-line prefetcher: if line n memory is accessed, it predicts that the next access will be to its contiguous line, $n + 1$. It can also predict other lines for later use, like $n + 2$, $n + 3$ and so on.

Another more complex heuristic is used by the stride prefetcher. Here, the prefetcher records a number of consecutive memory accesses (by program counter, or general for the whole system) and calculates the difference. If this difference d is constant after some accesses (a confidence threshold), it considers d as the stride and it can issue predictions for lines $n + d$, $n + 2\,d$, $n + 3\,d$, etc. Other options are explored in section 2.2.2.

Since data accesses from the CPU happen in a hierarchical way, the data is first checked on the lowest cache level or level 1 (L1), then on the L2 and so on up until main memory. This has the effect that only the L1 observes all accesses. This also means that a prefetcher in the L1 has more information to recreate the access patterns for prefetching. However, doing the prefetches just in the lowest-level cache may reduce the performance due to thrashing (if data is requested too far in advance and it is removed before being used due to capacity constraints) or lateness (if the data is requested later to avoid thrashing but does not arrive soon enough). A natural way to overcome these issues would be to bring more urgent lines (those that are going to be

used in the nearer future) to the lower-level cache, and bring the less urgent ones to a higher-level cache, with much more capacity[1].

However, the latency gap between memory and the CPUs is not the only issue. CPUs can be grouped in clusters (sockets), each cluster with their local memory and an increased latency for accessing memory in remote clusters. This results in a new problem: not all accesses cost the same in terms of latency (a NUMA system), and having a cache miss for data that is physically in a remote memory can have a huge impact in the performance of the application.

5.2.2. Opportunity for NUMA-aware prefetchers

Hardware vendors are designing systems that are becoming more and more complex, with more NUMA effects. For applications that present an embarrassingly parallel behaviour or good weak scaling, these effects can be hidden more easily, but more complications arise for workloads with non-trivial access patterns [67]. Moreover, the latencies and bandwidths can be so different between local and remote accesses that the execution times are greatly affected. Partly, some of the proposals from hardware vendors come from taking into account how the underlying memory behaves. They are considering prefetching technologies which behave differently depending on the characteristics of the downstream memory [65, 70, 96], mainly changing the prefetcher algorithm depending on the remote memory after some training.

A possible solution for the latency gap is being more aggressive when prefetching data from remote memory to prevent these long-latency misses. The problem is that careless prefetching would hide the benefits of prefetching and caching in general due to thrashing. This can also be solved by storing these prefetched lines in the higher cache levels (with higher aggressiveness, the extra lines have a lower confidence because they are expected

[1] This should not be confused with the *urgency* parameter for prefetchers in IBM POWER 7+ processors. That parameter corresponds to how fast the maximum prefetch depth should be reached by the prefetcher.

to happen further in the future). If the data is finally going to be used, the prefetcher would request it again later in the future and the data could be moved from the higher-level cache to a lower-level cache. Since the data from the remote memory is cached in a high-level cache, the NUMA effects should be reduced.

In order to evaluate this, we have built a microbenchmark that runs in a two-socket system with sockets A and B. The microbenchmark has the following phases:

1. Allocation: allocate a contiguous array.
2. Initialisation: initialise the data using core 1 of socket A. Due to the default first-touch policy in Linux systems, the data will be physically stored in socket A.
3. Local operations (1): execute read and write operations on the array at positions with a stride of one cache line using core 1 of socket A (all accesses local). Repeat *N* times.
4. Remote operations: execute the same operations using core 1 of socket B (all accesses remote). Repeat *N* times.
5. Local operations (2): execute the same operations again using core 1 of socket A (all accesses local). Repeat *N* times.

We measure the execution times of the three operation phases (3, 4, 5) under the following settings:

- In a system without NUMA effects and without a prefetcher.
- In a system with NUMA effects without a prefetcher.
- In a system with NUMA effects and a NUMA-aware prefetcher. This is a stride prefetcher that observes accesses in the level 1 cache and does the access predictions. However, instead of inserting predicted addresses in the prefetch queue of the cache, only the predictions for addresses in a remote NUMA node are considered, and they are inserted in the queue for the L3 (instead of the L1).

This is evaluated in a simulated system like the one presented in section 5.4.1, with a remote latency 1.5× larger than local latency for the NUMA systems. The results of this evaluation are shown in figure 5.1, which includes the average execution time for local and remote phases in each system. In the system without NUMA effects and without a prefetcher, the execution times of local and remote phases is very similar. When NUMA effects are included, the remote phase has a slowdown similar to the latency ratio between local and remote accesses. Once we add a stride prefetcher that only brings lines to the L3 cache (and only when they are remote), the resulting execution is like if there were no NUMA effects.

Thus, designing a hardware prefetcher that is aware of the NUMA effects in the system has the potential to improve the performance of parallel applications.

5.3. Proposal: NUMA-aware prefetching

We propose a prefetching scheme that leverages the non-uniform latency of the memory in NUMA systems. The proposal is generic and can be built on top of existing prefetchers because it only needs three parameters:

- the CPU that executes the access that triggers the address prediction or its NUMA node,
- the NUMA node containing the data of the predicted address, and
- how far in the future the access to the predicted address is expected to occur.

Our scheme is a modification on the general behaviour of prefetchers that internally use a queue to make the requests. It is also irrelevant whether the cache is indexed by physical or virtual addresses, but the prefetcher should be able to do prefetches across page boundaries, so the prefetcher would benefit from using virtual addresses and having access to a TLB. Without this, all the predicted addresses would fall on the same page as the triggering address. For simplicity, our explanations focus on a stride prefetcher to

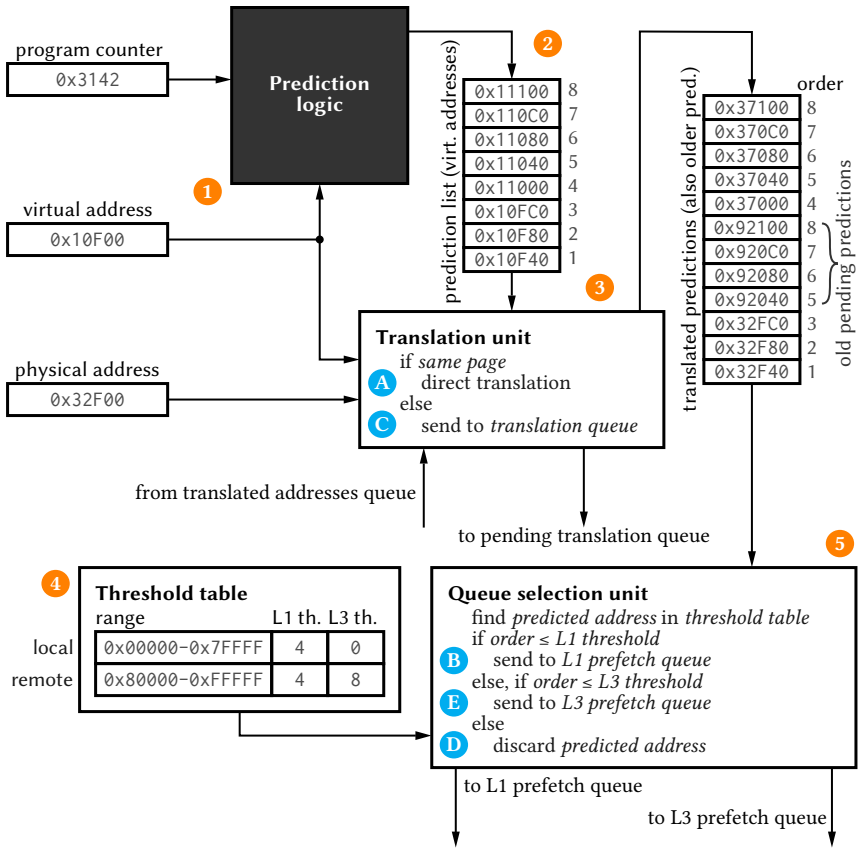


Figure 5.2.: Generic diagram of the NUMA-aware prefetching scheme.

build a NUMA-aware stride prefetcher (NASP), but the ideas should be independent of the algorithm that is used to predict the addresses.

5.3.1. The NUMA-aware stride prefetcher

Figure 5.2 shows a generic diagram of our NASP proposal as described below. In figure 5.3, we include an example code to explain the execution of our proposal together with the diagram. This simple code is equivalent to a loop that works with two vectors `A` and `B` and stores in `A` the sum of `A` and `B`.

Let us suppose the prefetcher is at the level 1 data cache and works with virtual addresses, but the cache uses physical addresses. Moreover, the

```
0x3140          cmp.lt r0, N    ; index < N ?
0x3141          bz endloop      ; end loop if false
0x3142  loop:   r1 = A[r0]      ; load operand 1
0x3143          r2 = B[r0]      ; load operand 2
0x3144          r1 = r1 + r2    ; compute result
0x3145          A[r0] = r1      ; store result
0x3146          r0 = r0 + 1     ; increment index
0x3147          cmp.lt r0, N    ; index < N ?
0x3148          bnz loop        ; continue loop if true
0x3149  endloop:                ; program continues
```

Figure 5.3.: Sample pseudo-assembly code for explaining the proposal. On the left, the PC (program counter) is shown.

prefetcher does predictions correlated with the PC (program counter). Suppose that we are using a system with pages of 4 KiB and that `A` and `B` are two arrays of 16 KiB each (four pages), each element in the array occupies one cache line and the cache lines are of 64 B (i. e., each page is 64 cache lines). Moreover, suppose also that `A` and `B` are aligned at the page level (start virtual addresses `0x10000` and `0x20000`, respectively), there are two sockets and `A` is physically allocated completely in socket 0 and `B` in socket 1. Finally, suppose that the code is executed by a CPU in socket 0, so accesses to `A` are local and to `B` are remote.

Let us consider that, for both local and remote address predictions, the L1 threshold is set to 4 prefetches and, only for remote predictions, the L3 threshold is set to 8. This means that, for any triggering address, the first four predictions will be to the L1 and, if the predicted addresses are remote, four predictions more will be added to the prefetch queue in the L3. This is done this way to hide the latency of remote accesses.

Imagine we are at an iteration where `r0` has a value of 60 and we are at PC `0x3142`, where the loop starts. The load will be of address `0x10F00` (1), and this will trigger the prefetcher to do eight predictions (2), from `0x10F40` to `0x11100`. The first three addresses, `0x10F40` to `0x10FC0`, are on the same physical page as the triggering address, so the translation is direct (A) and the translation unit (3) inserts them in the translated predictions list. In the threshold table (4), they correspond to an address in the local range and

their order is smaller than the L1 threshold, so the queue selection unit 5 sends them to the L1 prefetch queue B. The other five prefetches are on a different page and are sent to the pending translation queue for the TLB to respond, with their order attached C.

The execution continues, prefetches will be resolved when the system has a slot to do the accesses. A point in time will arrive when the translations are ready and inserted in the translated predictions list. Once they get to the queue translation unit, the translation for `0x11000`, corresponding to the fourth prediction, will be stored in the L1 prefetch queue B because its order is equal the L1 threshold. The other four addresses, corresponding to the translations of `0x11040` to `0x11100`, will be discarded D because they are local to the executing CPU and farther in the future than the thresholds for the L1 and L3 (their order is larger than both thresholds).

Similarly, once the load in PC `0x3143` is executed, the address `0x20F00` will trigger the prefetcher 1. Like before, the prediction logic will give eight predictions 2, from `0x20F40` to `0x21100`. The first three (`0x20F40` to `0x20FC0`) have a direct translation A, are below the threshold for the L1 4 and will be directly inserted into the prefetch queue of the L1 B.

However, there is a difference for the other five predicted addresses. Following the same process as with the previous load, once the TLB has the translation for them, all of them will correspond to remote addresses. The fourth, the translation for `0x21000`, will go to the queue in the L1 due to the threshold B. The last four are also remote, but their orders fall above the threshold for the L1 and below the threshold for the L3 for remote addresses, so the translations for `0x21040` to `0x21100` will be inserted in the queue for the L3 E.

5.3.2. Other considerations

The number of parameters for the NUMA-aware decisions can be increased to consider more options. For example, some prefetchers have a confidence value and a threshold for this confidence before starting to do the predictions

or to regulate the number of predicted addresses. A NUMA-aware prefetcher could make an aggressive prefetching by bringing remote lines to the L3 if the confidence is still low to hide the latency of remote accesses. This complicates the design, which is why we have not considered the parameter. In any case, accesses that are expected in the short term will have higher confidence than those expected to happen later.

Regarding other aspects like multithreading (i. e., SMT) or multiprogramming (and the problems with context switching), where multiple threads are sharing resources of a core, our proposal should not have a negative effect and even have some benefits in general. NUMA-aware prefetching helps reducing the number of prefetches in the L1 cache, thus reducing the risk of thrashing due to having multiple threads running in the same core (higher levels of cache have more capacity, and therefore thrashing is not such a risk).

With respect to the use of the extra queues and buffers, their behaviour is the same as for the already existing queues, flushing them or not in a context switch according to the hardware implementation. Regarding the threshold table 4, in our evaluations the contents are hardcoded and not exposed to the user. The thresholds could be exposed in a similar way to Intel's MSR `x1A4` register or IBM's DSCR to set up the prefetchers, that could have more performance implications but that is left as future work.

5.4. Methodology

5.4.1. Simulation environment

We use the gem5 simulator with an environment as described in section 3.2. We have modified the `Prefetcher::Queued` class in gem5 in order to include the mechanism described in section 5.3, with two variants (one for the L1 cache and one for the higher-level caches). In the case of the version for the L1 cache, if the prediction is for the same page as the triggering access, the physical address is known by simple arithmetic and the prediction is

stored either in the L1 prefetcher queue or communicated to the higher levels according to criteria from section 5.3. If it is in a different page, the prediction is pushed to the pending translations queue, and once the translation is available the prefetcher decides whether it should go to the L1 or a higher cache level. If the TLB does not have the translation, it is simply discarded (no page walks are done), as defined in gem5 by default for prefetches.

For the L2 and higher, the prefetcher itself just receives the messages from the L1 prefetcher with the addresses that it should prefetch, so they are simply pushed to the prefetch queue. The rest of the behaviour is like the standard implementation in gem5.

To simplify the proposal, we will just consider prefetches to the L1 cache and the last-level cache (in this case, L3).

5.4.2. Design space exploration

We evaluate our ideas by selecting some configuration points that cover different scenarios:

- All prefetches go to the L1 cache (default).
- All prefetches go to the L3 cache.
- Some prefetches go to the L1 cache and some go to the L3 cache, combining the following:
  - For predicted local addresses, we set two constants k_{loc} and $k_{\text{loc}}^{\text{L1}}$. The first $k_{\text{loc}}^{\text{L1}}$ predictions go to the L1 and the rest $(k_{\text{loc}}^{\text{L1}}+1, \dots, k_{\text{loc}})$ go to the L3.
  - For predicted remote addresses, we also set two constants k_{rem} and $k_{\text{rem}}^{\text{L1}}$. The first $k_{\text{rem}}^{\text{L1}}$ predictions go to the L1 and the rest $(k_{\text{loc}}^{\text{L1}}+1, \dots, k_{\text{loc}})$ go to the L3.

In our executions, we set values such that $k^{\text{L1}} \coloneqq k_{\text{loc}}^{\text{L1}} = k_{\text{rem}}^{\text{L1}}$ and $k_{\text{loc}} = 0$ (only prefetching remote lines in the L3 cache, not lines in the local DRAM).

We use the notation NASP k^{L1}:k_{rem} to refer to these prefetching schemes. Similarly, we use stride k^{L1} for a stride prefetcher in the L1. For example, in NASP 8:64 and stride 8, we have $k^{L1} = 8$ (and $k_{rem} = 64$ for NASP).

Evaluated benchmarks

We evaluate the performance of the NAS Parallel Benchmarks, presented in section 3.3.1, in NUMA and non-NUMA simulated systems with different prefetcher configurations. In the case of NUMA systems, using 4 sockets, the latency for accessing data the local DRAM is 80 ns. Accesses to data in the non-local DRAM is 80 ns greater for the neighbour socket, and 160 ns greater for the two farther sockets. In non-NUMA systems, all accesses have the same latency as local accesses in the NUMA systems.

For the evaluation of the prefetchers in the NUMA system, we consider the default configuration (all prefetches go to the L1) and our proposal where part of the prefetches go to the L1 the rest go to the L2, depending on different parameters. We compare the performance of these systems against a non-NUMA system with the default prefetcher configuration, to see how much improvement we get with our NUMA-aware scheme.

In all simulations, the MSHR queue size has at least 24 positions for the L1 and L2, and 32 for the L3, with the value increased to the prefetcher queue size if it is larger. The queue size for the prefetcher is set to the maximum prefetch degree plus 8, and for NASP in the L3 it is set to $k_{rem} - k^{L1} + 4$ for each cache slice.

5.5. Results and evaluation

In this section we show the results of the evaluation of our proposal. We start with the results of the design space exploration presented in section 5.4.2 (section 5.5.1). We continue with the performance evaluation, not only in terms of execution time but also of reduction in the misses and how the proposal is affected by the NUMAness (section 5.5.2). After the general

performance evaluation, we do a comparison with different prefetchers from the state of the art (section 5.5.3), we extend one of these prefetchers to use our ideas (section 5.5.4) and finish with the evaluation of the communication and energy costs incurred by the proposal (section 5.5.5).

5.5.1. Design space exploration

We have developed a microbenchmark that does sequential accesses to the data in a controlled manner, using both "local" and "remote" CPUs, to evaluate how the different parameters affect the executions. This microbenchmark is presented in section 5.2.

Initially, we intended to do the parameter exploration with the microbenchmark. However, the results were unsuccessful for various reasons, mainly due to the density of memory instructions (one memory read/write every six instructions) and expected miss rate in a system without hardware prefetching (50 % of all accesses). This forces the L1 prefetcher to do prefetches very far in advance, which at the same time causes thrashing in the cache. Considering the prefetches in the L3 for remote accesses, the amount of prefetches needs to be very large (starting from 128 prefetches, the NUMA effects are completely hidden), which are the results in figure 5.1.

When evaluating more complex applications, with many threads running simultaneously, the stride prefetcher in the L1 already starts to give some performance improvements. The results are detailed further in the following section.

5.5.2. Performance evaluation

Figure 5.4 shows the performance results of the different NAS Parallel Benchmarks (NPB) with input class A in a 4-socket NUMA system where the latency of accesses to local DRAM is 80 ns, accesses to the DRAM of the nearby socket are 160 ns (2× the local latency) and, finally, accesses to the DRAM of the farthest sockets is 240 ns. The results are normalised to stride 8.

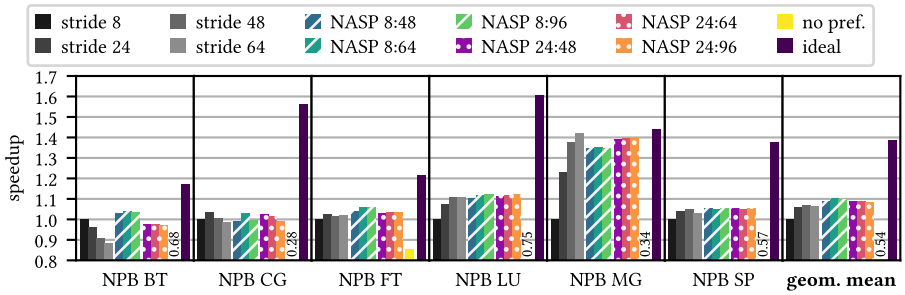


Figure 5.4.: Speedups of standard stride prefetcher configurations and our proposal.

For the standard stride prefetchers, everything is prefetched to L1. Speedups are normalised to stride 8 (stride prefetcher that brings 8 lines to the L1 cache). Executions are in a simulated NUMA system like the one in table 3.1.

For most applications, except for MG, all prefetchers are far from hiding the NUMA effects.

When we consider the standard stride prefetcher in the private L1 caches (the first group of bars), in general there is some speedup already for stride 24 (1.06× on average) with up to 1.07× on average for stride 64. However, bringing 64 lines to the L1 cache can be unnecessarily aggressive.

In most cases, the NASP 8:64 scheme, which does a standard stride 8 in the L1 and brings up to 64 remote lines (1 page) to the L3, gives very good results. It is the best on average with 1.10× speedup over stride 8 and also has some speedup over stride 64. In comparison, NASP 24:64 has an average speedup of 1.09× over stride 8, although for some benchmarks it gives better performance than NASP 8:64.

Only in the case of the MG benchmark all evaluated NASP configurations underperform the best L1 stride prefetcher, which matches the ideal case without NUMA. In particular, NASP 8:64 (the best on average) has a speedup of 1.35× while the stride 64 (best stride on average) has a speedup of 1.42× and NASP 24:64 has a speedup of 1.39×. This is because the memory access pattern for MG is well predicted by stride prefetchers and aggressive prefetching in the L1 improves performance without causing thrashing.

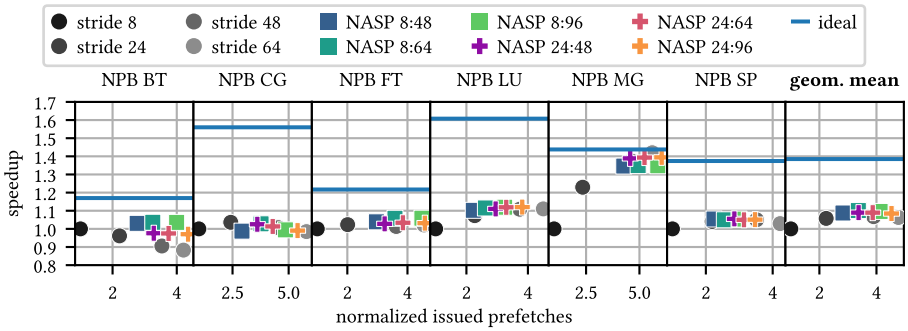


Figure 5.5.: Speedups vs. total number of issued prefetches (both normalised to stride 8) at all cache levels.

In a single-socket environment, NASP 8 would behave like stride 8 and NASP 24 would behave like stride 24, no matter the threshold for remote L3 prefetches (as long as all data is local). When manufacturing a system with NASP, it could be set so that the threshold is 0 for the L3 when there is only one NUMA node in the system to prevent spurious calculations.

Reduction of the issued prefetches

The purpose of our proposal is not only to have higher speedups. With NASP there is an effective reduction in the total amount of predictions that are issued by the prefetchers to the caches compared with other prefetchers that achieve the same speedup. This is shown in figure 5.5, which has the speedups of each prefetcher configuration in the *Y* axis against the number of issued prefetches for all cache levels (normalised to stride 8) in the *X* axis. If we only considered the prefetches issued for the L1, the NASP 8 and NASP 24 schemes issue the same number as stride 8 and 24, respectively, since they basically act as a stride prefetcher when not considering the prefetches into the L3 cache.

What we take from the plots in figure 5.5 is that, on average, NASP can obtain a similar speedup as stride 64 with just issuing 3/4 of the prefetches (using NASP 8:64 or NASP 24:64). This is thanks to bringing remote lines to the L3, instead of bringing all lines. Moreover, most of these prefetches will go to

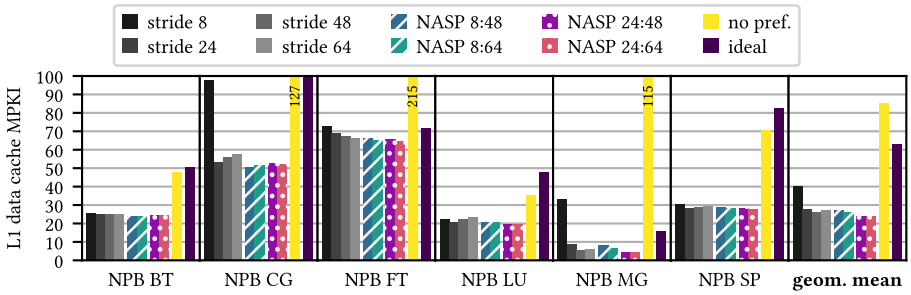


Figure 5.6.: Level 1 data cache misses per 1000 instructions.

the L3 cache instead of the L1 cache, improving the total communications as shown in section 5.5.5.

The results are not always the same, however. For CG, the NASP configurations that have similar results to the best stride prefetcher (stride 24) issue more prefetches than that stride prefetcher. However, the difference is small enough not to mean a big increase in the number of issued prefetches.

Reduction of the L1 cache misses and unused prefetches

Another key result, as figure 5.6 shows, is that there is a reduction in the misses per 1000 instructions in the L1 cache (considering only the misses due to actual accesses from the executed code). Using NASP, just bringing 8 lines to the L1 (the NASP 8 schemes) we get MPKI values similar to bringing 24 lines using a normal stride prefetcher, and even lower in some cases. We do not show such results for the L3 cache because the values are generally low.

Figure 5.6 is a proof for the complex access patterns for CG and FT, compared to the rest of the applications: they show a large number of misses per 1000 instructions in the L1 cache for any of the prefetcher configurations. In the case of CG, it is a sparse linear algebra application. This means that memory accesses within the application are generally irregular and stride prefetchers make bad predictions. For this reason, prefetching too aggressively in the L1 cache can cause thrashing and NASP 8 has a lower MPKI value than NASP 24 or aggressive stride prefetchers. The high MPKI rate of stride 8 is hidden by

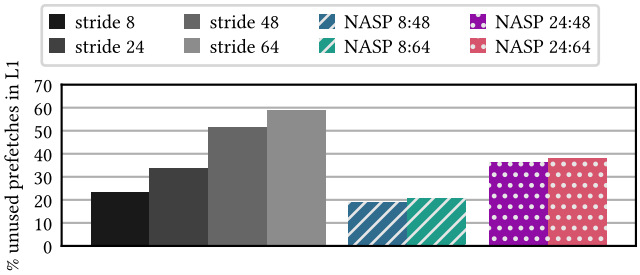


Figure 5.7.: Geometric mean of the percentage of unused cache lines coming from prefetches in the L1 for different prefetcher configurations.

NASP 8 thanks to the prefetches in the L3 cache, which reduce the latency between L1 and memory.

On the opposite side, there is MG with a very low number of misses thanks to the regularity of its memory access patterns. The memory accesses in this application are very regular. However it is very memory intensive as well. As figure 5.4 showed and we mentioned in section 5.5.2, the performance results for NASP in this application do not follow the same trend as the rest of the applications. Being so memory intensive can mean that data needs to arrive earlier to the L1 cache to be used, requiring a more aggressive prefetching approach in the L1 to have the data in time. This is supported by the L1 MPKI for MG, which in general shows fewer misses for aggressive stride prefetchers (stride 64) than for NASP 8.

Figure 5.7 shows that NASP reduces the ratio of prefetches that are evicted from the cache before being used (those that arrive too early to the cache or are mispredictions) when compared to stride configurations that achieve the same speedup. In particular, the NASP 8 configurations have less unused prefetches (in the L1) than stride 24 and both NASP 8 and NASP 24 fall below stride 64. This is expected because the total number of lines that are prefetched to the L1 cache is much lower. These ratios are calculated just considering the lines that are marked as coming from the prefetcher in the L1. Unlike in figure 5.5, the predicted lines that are already in the cache are not accounted for in the total prefetched lines. However, these results come with a caveat as we explain in section 5.5.2.

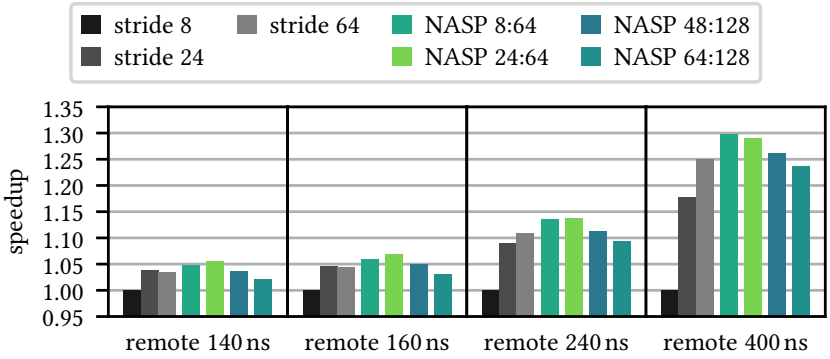


Figure 5.8.: Speedups for various 2-socket systems with local latency of 80 ns and different remote latencies.

Each system is normalised to its execution with stride 8.

Reducing prefetches in the L1

One key observation is that reducing the latency between the L1 cache and main memory means that prefetches arrive earlier to the cache. This has the effect of allowing to reduce the aggressiveness of the L1 prefetcher (the maximum prefetch depth in the case of the stride prefetcher), but at the same time it can be a requirement to do so. Otherwise, there could be thrashing in the cache or some prefetched lines might be discarded before being used.

This is the reason why our NASP proposal can have lower performance when the prefetch depth in the L1 is high compared to a standard stride prefetcher with the same depth in the L1 for lower latencies. For instance, figure 5.8 shows that NASP 64:128 has worse performance than stride 64 for all latencies, when the data prefetched into the L1 should be roughly the same among both prefetchers. This is also one of the reasons why, when we fix the number of lines prefetched into the L1 and we increase the number for the L3, we can see a reduction in the speedups we obtain. For reference, figure 5.7 shows that while calculating the same amount of prefetches for the L1 (NASP 8 or NASP 24), when increasing the prefetches in the L3 it can happen that the proportion of unused prefetches in the L1 increases as well.

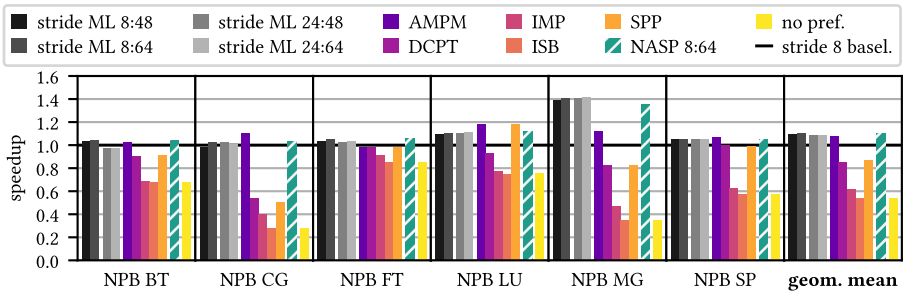


Figure 5.9.: Comparison of state of the art prefetchers with our proposal.

Sensitivity to the NUMAness

There are systems in which the remote accesses have even higher latency than in the system we have evaluated. This can often happen in systems with more than two NUMA nodes [26, Table 2], [136, Figure 4]. Figure 5.8 shows the sensitivity of different stride and NASP prefetcher configurations to increasing remote latencies in a dual socket system. While at small latency differences the speedups obtained by NASP with respect to a stride 8 prefetcher are just 1.05× and not far from stride 24, as the latency grows the benefits of using a NUMA-aware prefetcher like NASP become clearer. In a NUMA system with various latencies, one approach like NASP can help improve the performance as we have shown in the previous sections: the prefetches to NUMA nodes with smaller latencies will not suffer from using this approach and those to farther nodes can greatly benefit from it.

5.5.3. Comparison with the state of the art

Up until now, all the comparisons in this chapter have been with a standard stride prefetcher, which is one of the prefetching techniques implemented by vendors in a general manner. We have also evaluated our proposal against various state-of-the-art prefetchers: *access map pattern matching* (AMPM) by Ishii, Inaba and Hiraki [69], *best-offset prefetcher* (BOP) by Michaud [105], *delta-correlating prediction tables* (DCPT) by Grannaes, Jahre and Natvig [61], *indirect memory prefetcher* (IMP) by Yu et al. [163], *irregular stream buffer* (ISB) by Jain and Lin [71], and *signature path prefetcher* (SPP) by Kim

et al. [86]. For all prefetchers we use the implementation available on gem5 without further customisation. The results are shown in figure 5.9. BOP is for the L2 cache, so we combine it with the stride prefetcher in the L1. The results, however, do not show any benefit for us when compared with the standard stride prefetchers in the L1 (without BOP in the L2). For this reason, we have included a comparison against a multilevel prefetcher, similar to those implemented by hardware vendors and analysed in section 5.5.3.

The prefetchers that could be contenders for NASP (1.10×) are the stride multilevel prefetchers (1.10× for stride ML 8:64), AMPM (1.08×), and, in some cases like the LU benchmark, SPP (0.87× on average). The rest of the prefetchers generally show worse performance than a stride prefetcher. As for AMPM, having a performance more on par with NASP, it has a much more complex logic behind its ideas. Interestingly, for the MG benchmark the performance of AMPM falls way behind NASP, achieving only 1.13× speedup, much lower than the 1.35× speedup of NASP.

We have not compared against the *programmable prefetcher* by Ainsworth and Jones [3] due to the complexity of adapting it to executions with multiple cores. With some core changes, we believe it could be reasonable to implement NUMA-aware algorithms in this prefetcher, but this falls out of the scope of the thesis.

Comparing with a NUMA-unaware multilevel stride prefetcher

One of the prefetcher options implemented by vendors is to make some of the prefetchers in the higher cache levels. Our NUMA-aware proposal is very similar to this, with the exception that only prefetches to remote addresses go to the last-level cache. To show that most of the benefits come from prefetching the remote addresses and not from using the L3 cache, we have executed experiments comparing against a multilevel stride prefetcher (stride ML) that is like NASP but disregarding the NUMA distance of the predicted addresses. The results are shown in figure 5.9, together with the state of the art.

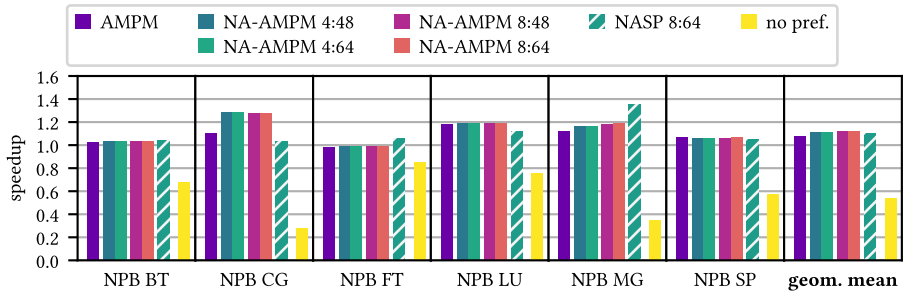


Figure 5.10.: Extending AMPM prefetcher with NUMA-aware capabilities.

In most cases, the difference between bringing just remote lines or bringing all lines to the L3 is around 1 %, needing more than three significant digits for the speedup. This means that, in practice, NASP can give the same performance as a stride prefetcher that brings lines to multiple cache levels but issuing less prefetches: the number of issued prefetches is reduced by 15 % by using NASP 8:64 instead of stride ML 8:64. Only for MG the performance of stride ML is better, the reason being that MG has very good predictability for stride as explained in the analysis in section 5.5.2.

5.5.4. Making other prefetchers aware of NUMA

We mentioned earlier that the NUMA-aware prefetching scheme can be applied to extend other prefetchers that generate multiple predictions. We have extended the AMPM prefetcher, which is the best in the state of the art for our benchmarks, to a NUMA-aware AMPM prefetcher or NA-AMPM: in it, only some prefetches go to the L1 and the rest of the prefetches go to the L3, but only if the predicted addresses are remote. The results are shown in figure 5.10.

Just like AMPM performs slightly worse than NASP (1.08× vs. 1.10×) on average, the NA-AMPM proposal shows an improvement, with 1.12× speedup with respect to stride 8. However, the results are very different depending on the benchmark; three example cases are CG, MG and FT. For the first one, AMPM has a performance better (1.10×) than NASP 8:64 (1.03×), but any of the NA-AMPM 4 configurations gives 1.28× speedup, showing great

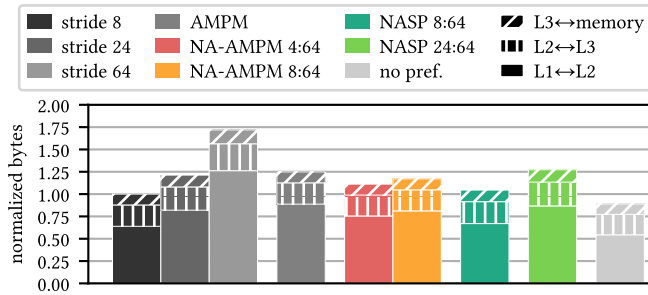


Figure 5.11.: Average transferred data in the memory hierarchy, including snoop packages, for different prefetcher configurations (normalised to stride 8).

improvements thanks to being NUMA-aware. For MG the improvement of NA-AMPM with respect to a standard AMPM prefetcher in the L1 is lower (1.19× to 1.28×), and NASP shows much better performance thanks to the use of the stride prefetcher underneath. Finally, FT does not show any improvement with using a NUMA-aware AMPM with respect to AMPM, although NASP gives some performance benefits to a simple stride prefetcher in the L1.

To summarise, with these benchmarks the use of a NUMA-aware prefetcher does not make the performance worse than a NUMA-unaware prefetcher, on average there is some improvement and in some cases the results are much better.

5.5.5. Cost evaluation

Data transfer

One of the goals of NASP is reducing unnecessary data transfers to obtain a higher speedup at a lower cost. Figure 5.11 shows the normalised bytes transferred in the memory hierarchy, on average for all the evaluated applications, and using a selection of prefetchers including NASP, stride, the best of the state of the art (AMPM) and its NUMA-aware version (NA-AMPM). The reduction of transferred bytes for NASP compared against prefetchers with similar performance is very clear. NASP 8:64 transfers slightly more

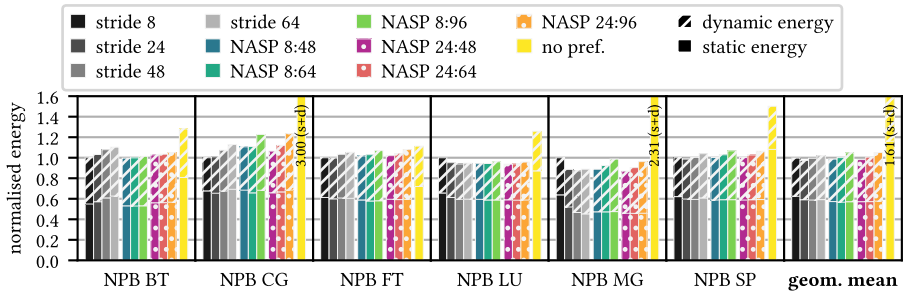


Figure 5.12.: Normalised energy consumption in the sockets (cores, L1, L2 and L3) for the different prefetchers.

bytes than stride 8 (1.05×), and that is expected because it explicitly brings more data to the L3 cache. However, stride 24 and stride 64 bring much more data (1.21× and 1.72×, respectively) and they generally have worse performance than NASP 8:64. The same happens when considering the AMPM prefetcher in the L1 (1.25×), which also has the disadvantage of requiring more complex data structures. The NUMA-aware versions, NA-AMPM 4:64 and NA-AMPM 8:64, transfer more bytes than NASP 8:64 (1.11× and 1.17×, respectively), but still less than AMPM.

In a similar way to NASP 8:64 and stride 8, the results show that NASP 24:64 brings slightly more data than stride 24 (1.27× with respect to stride 8, compared to 1.21× of stride 24), but the average improvement in the performance is better than the increase in the amount of transferred data. As it would be expected, using no prefetcher has the least amount of communications, but the reduction in data transfer is not that much compared to the extremely reduced performance of the applications when not using a prefetcher even as simple as stride 8.

Energy consumption

The results of the energy consumption evaluation can be found on figure 5.12. The modelling has been done with McPAT [91] (and CACTI-P [92], used by McPAT) including the modifications proposed by Xi et al. [161]. We have considered a 22 nm process technology. In the modelling, we include the

different capacities of the queues used by the prefetchers as well as all the communications incurred by them in the cache hierarchy. The total area considering both sockets and the L3 caches is 45 mm^2 (or 27.6 mm^2 not considering the L3), with a negligible difference in size between the different evaluated prefetchers.

The results show that, in general, the energy consumed by the socket when using NASP is not higher than for the stride prefetchers. This is not the case for the power, as NASP has a higher power consumption. However, this higher power consumption is hidden in the energy by the much faster execution times as the results show. On average, NASP 8:64 and NASP 24:64 consume the same energy as stride 8, but stride 64, the stride with the best speedup, consumes 1.02×.

5.6. Summary

In this chapter we have shown that using a NUMA-aware hardware prefetcher can improve the performance in systems with a high NUMAness. The ideas presented here are simple and generic and can be implemented for different hardware prefetchers at a low cost and with simple logic. However, prefetches are not perfect and NUMA effects cannot be completely hidden with our proposal. Moreover, not all applications perform equally with the same prefetcher parameters. In this regard, future work could go in the direction of exposing the parameters to the user. With the adequate runtime system, this could enable choosing the parameters by the runtime, like many proposals for existing hardware have done already.

Chapter 6.

Task-based applications in NUMA systems

This chapter proposes techniques at the runtime system level to further mitigate the impact of NUMA effects on parallel applications' performance. Both scheduling and prefetching have a major impact in the performance of parallel applications in NUMA systems, as shown in chapter 4. And even though considering both aspects simultaneously achieves the best performance, scheduling plays a major role in this improvement. This means that doing an adequate scheduling, using the available information from the system and modern programming models can further improve the results.

This contribution leverages system metadata expressed in terms of a task dependency graph, introduced in section 2.3.1, to efficiently reduce data transfers. The presented approach adds negligible overhead and considers the information contained in this TDG data structure to drive two techniques applied at the runtime system level; they apply advanced graph partitioning algorithms to break down the TDG into several pieces or parts. These partitions aim at minimising data transfers across the parallel system.

In summary, the contributions presented in this chapter are the following:

- Two schemes that dynamically perform graph partitioning over the TDG: The *runtime-informed partitioning with dependency easy placement* (RIP-DEP) and the *runtime-informed partitioning with moving*

window (RIP-MW). Both approaches partition an initial subgraph containing the firstly created tasks but propagate this partition in different ways: RIP-DEP exploits information regarding the allocation of tasks' input data while RIP-MW repartitions the initial TDG subgraph as new tasks are added.

- A complete performance evaluation of the proposed techniques against 3 other methods: an expert programmer-driven policy, a locality-unaware *distributed first-in-first-out* (DFIFO) approach and an implementation of a state-of-the-art technique [53, 54, 152], *dependency easy placement* (DEP), that automatically schedules tasks depending on where their input and output data are allocated. Our evaluations consider 8 different OpenMP codes and 2 different parallel systems with up to 288 cores. Our proposals incur minimal runtime system overhead while keeping the parallel workloads well balanced. Our experiments show how RIP-DEP achieves speedups of up to 1.52× and average improvements of 1.12× on 288 cores with respect to DEP, the best state-of-the-art approach.

- An exhaustive evaluation of the coherence traffic triggered by all the considered approaches. The evaluation includes categories like control traffic, which is composed of messages carrying coherence protocol signalling activities without a data payload, and data traffic, which is composed of messages carrying a single cache line payload. Our coherence traffic evaluation explains the performance benefits of our techniques as it demonstrates that RIP-DEP achieves outstanding coherence traffic reductions of 172.2× and 2.28× on average compared to DFIFO and DEP, respectively.

6.1. Graph partitioning

Throughout the literature [22, Section 2], the graph partitioning problem is defined as in problem 1.

Problem 1 (Graph partitioning)**.** *Given a positive integer k and an undirected graph $G = (V, E)$ with positive edge weights $\omega\colon E \longrightarrow \mathbb{R}^+$, find a partition Π of the set of vertices V composed of k parts V_i with the following properties:*

1. *$V_1 \cup \cdots \cup V_k = V$ (the parts cover all the vertices),*
2. *$V_i \cap V_j = \emptyset$ if $i \neq j$ (the parts are disjoint).*

In general, we want partitions that are balanced, that is $|V_i| \leq (1+\varepsilon)\lceil |V|/k \rceil$ for some $\varepsilon \geq 0$, and such that some metric is minimal. If we define the mapping $\varphi\colon V \longrightarrow 1, \ldots, k$ that assigns every vertex to the partition where it belongs, or $\varphi(v) = i$ if $v \in V_i$ in Π, we want to minimise the function

$$\sum_{\substack{uv \in E \\ \varphi(u) \neq \varphi(v)}} \omega(uv). \tag{6.1}$$

This function (6.1) is known as the *edge cut* of the solution and corresponds to the total weight of the edges connecting pairs of vertices from two different parts in Π.

Under these constraints, the problem is NP-hard, but there are known algorithms and heuristics for approximating it [22]. Some of the commonly used libraries in the HPC scenario are Metis [81, 82], Scotch [120, 121], Zoltan [18] and Metapart [127]. These libraries aim at reducing data transfers across parallel distributed memory systems by statically splitting input data like meshes or matrices.

6.1.1. Graph partitioning algorithms

In this contribution we use standard graph partitioning tools for undirected graphs to partition the TDG of applications, which are directed acyclic graphs. In particular, we partition the TDG in k parts, where k equals the number of NUMA regions (or sockets), and use the amount of transferred data between parts as edge cut function. While there is a wide range of graph partitioning algorithms (exact, recursive, greedy, local search...), this thesis makes use of a multilevel approach combined with Dual recursive

bipartitioning, Fiduccia-Mattheyses and Graph growing algorithms, which are summarised below. Complete details of these and other approaches are described in the literature [22].

Dual recursive bipartition is one of the most basic and used methods. It is a recursive divide-and-conquer algorithm that consists in doing a 2-partition of the set of parts, and a 2-partition of the set of vertices and map the last two to the pair of sets of parts. The mapping is done recursively until what is assigned is a set of tasks to a single part. The bipartitions are done using some heuristics that use the information from the edge weights to make good decisions.

Rather than a partitioning/mapping algorithm, *Multilevel mapping* is a scheme to do the partition in an easier way or with higher quality. It coarsens the input graph (makes it rougher, joining vertices), then applies the partitioning algorithm to the coarsened graph, projects back the partition to the original graph and refines it. This is well shown in SCOTCH User's Guide [122, Figure 3].

Fiduccia-Mattheyses is a local-search algorithm extended to not stall in a local minimum, and it is an evolution of the Kernighan-Lin method (another algorithm using local search). Starting with a given partition, it tries to improve it by moving vertices from one part to another or by swapping vertices in different parts. The selection is done with the vertices that make the edge cut decrease the most.

Graph growing algorithms are based on a breadth-first search that starts from some seed vertices and grows the parts greedily. The parts are grown in an order such that the next part to get a vertex is always the smallest one. Local search is then applied to balance the load of the parts, and new seed nodes are selected for the next step.

The ways in which we use these methods are detailed in section 6.2.3, where we introduce and explain our proposals for reducing NUMA effects based on graph partitioning.

6.2. Exploiting the task dependency graph to mitigate NUMA effects

In order to automatically orchestrate a parallel execution while optimally mitigating NUMA effects on large shared memory nodes, we exploit the information contained in the TDG of the application. To do so, we consider either techniques that analyse the TDG by means of a simple heuristic or techniques based on advanced graph partitioning algorithms.

In order to be able to apply the proposed techniques, throughout the rest of the work, we assume a first-touch memory placement policy and page-aligned memory blocks. This means that a data page is physically allocated in memory the first time it is used, and the allocation is done in the NUMA domain of the core making the access, which is the default behaviour in a Linux system.

6.2.1. Dependency easy placement (DEP)

By *dependency easy placement* (DEP), we refer to the approach proposed by Drebes et al. [53, 54] in terms of a dynamic task and data placement policy based on two concepts: i) *deferred allocation*, which implies that the memory to store task output data is not allocated until the task placement is known, and ii) *enhanced workpushing*, which means that tasks are scheduled to the NUMA region where most of their data dependencies are allocated. A similar method is proposed by Virouleau et al. [152]. In our context, the enhanced workpushing mechanism is implemented by means of a table to map the dependencies to sockets kept by the runtime system. The first address of a data dependency is used as its identifier; this way, we avoid invoking high cost system calls to figure out the sockets where the data is allocated. Also, data dependencies are allocated in the socket where the first task accessing them is executed, which is equivalent to the deferred allocation mechanism.

At the time of scheduling a task, the runtime explores its dependencies and weights the sockets using the size of the allocated dependencies (input and output), considering a virtual extra socket for unallocated data (also weighted using the size). Then, the task is scheduled to the socket with the highest weight. If the highest weight is for the virtual socket (unallocated data), the final socket is chosen via a discrete uniform distribution considering all the sockets available to the runtime system. In case of a tie, the socket is chosen via a discrete uniform distribution among the tied ones. Observe that DEP can also be seen as a *propagation* technique: once the data is placed physically in memory (using some kind of heuristic), tasks can be scheduled in cores that are near the data they use to be able to consume it faster.

This thesis demonstrates in section 6.4 how techniques based on graph partitioning achieve better performance and dramatically reduce the amount of data transfers carried out by techniques like the one proposed by Drebes et al. [54].

6.2.2. Considerations about applying graph partitioning on applications' TDGs

To exploit the structure of the application we use graph partitioning algorithms. Considering the whole TDG is not an option because partitioning schemes target undirected graphs, which implies that they typically split TDGs with deep task paths in a way that all potentially concurrent tasks are assigned to the same part. Intuitively, when the graph is wide rather than tall, the partitioning algorithm will decide that it is better to cut the edges (i.e., partition the graph) vertically because there will be fewer edges than horizontally. Using *hypergraph* partitioning software packages does not help in our context either since they use algorithms with high computational cost [31, 32, 83, 84] that require large distributed memory systems to run [18]. Also, since in practice the dependency graph is built simultaneously with the execution, the complete TDG is never available at runtime. In this context, the natural way to proceed is operating over small task subgraphs instead of

over the whole TDG, that is, partitioning subgraphs and then extrapolating this partition to the upcoming tasks following a certain policy.

6.2.3. Runtime informed partitioning (RIP)

Under the *runtime informed partitioning* (RIP) family of policies, task scheduling decisions are based on graph partitioning techniques. The TDG is built at run time by leveraging information in terms of task dependencies. The graph is updated every time new tasks are instantiated, and partitioned once the execution goes through a barrier point or a limit in terms of the total number of tasks contained in the graph —called the *window size* limit— is reached. The partitioning algorithm uses the TDG as input, weights its edges depending on the amount of bytes they represent and assigns tasks to a particular part (corresponding with a specific socket) taking into account the machine NUMA distances.

In order to partition the initial subgraph, we use the dual recursive bipartition, the multilevel mapping and the Fiduccia-Mattheyses methods described in section 6.1.1, and available in the Scotch [121] graph partitioning library, version 6.0.4. The graph growing algorithm, also described in section 6.1.1, is available within the Metapart framework [127]. We represent the target architecture as a complete graph with as many vertices as NUMA domains, and with edge distances proportional to the NUMA distances measured as explained at the beginning of section 6.3. For doing the partitions, the TDG is transformed to an undirected graph for Scotch. Once the complete graph that defines the target architecture is set, the initial partition is obtained by calling `SCOTCH_graphMap` with the default settings. Such default settings make use of a multilevel approach with dual recursive bipartitioning combined with the Fiduccia-Mattheyses local search algorithm.

We partition the initial subgraph given by the first *window size* tasks. The partitioning is done asynchronously while the runtime system creates new tasks. Once the initial subgraph has been partitioned, we consider two possible options to proceed: the first one consists in propagating the partition

across the whole execution following a memory-allocation-aware policy, which corresponds to the RIP-DEP technique. The other alternative is to keep partitioning the different subgraphs the runtime system generates as the execution advances, which corresponds to the RIP-MW approach. The first technique aims at reducing the overhead due to graph partitioning as much as possible while RIP-MW aims at dynamically adapting the TDG partition during the parallel run. The overhead of partitioning the initial subgraph is small enough to be overcome by the benefits of graph partitioning (see section 6.4 and section 6.4.4 in particular, which provide performance results accounting for task creation and scheduling, as well as partitioning the TDG).

When tasks are ready to run (i.e., all their input dependencies are solved) but the partition is not done yet, they are stored in a temporary queue. Tasks are transferred to the ready queue as soon as they have been assigned to a socket. The temporary queue is not used often since, in general, the partition is obtained much before the tasks are ready.

RIP with dependency easy placement (RIP-DEP)

This technique based in graph partitioning consists in propagating the partition obtained from the initial subgraph by taking into account where the tasks data dependencies reside. More specifically, this approach uses DEP to propagate the partition to the rest of the graph, already described in section 6.2.1. The main difference between DEP and RIP-DEP is the way of doing the initial partition: while DEP does the allocation using a uniform distribution, RIP-DEP partitions the TDG.

RIP with moving window (RIP-MW)

In this case, the graph partitioning is performed many times throughout the execution of the program. Once the subgraph contains a particular amount of tasks —the *window size*—, or a barrier point is reached, the partitioning algorithm is run. Once a partition is obtained, the oldest tasks are flushed

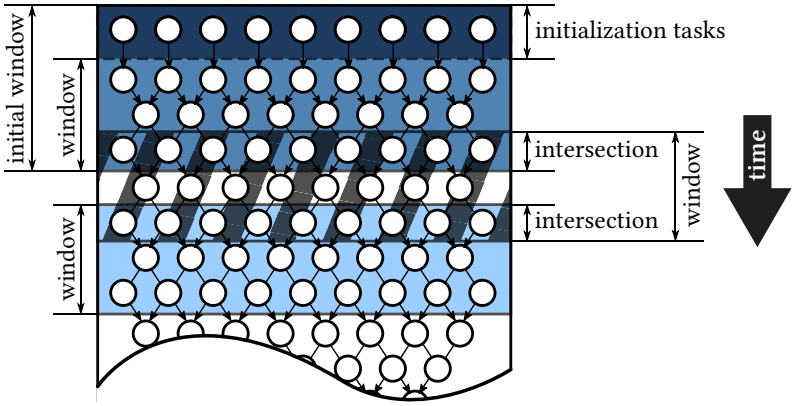
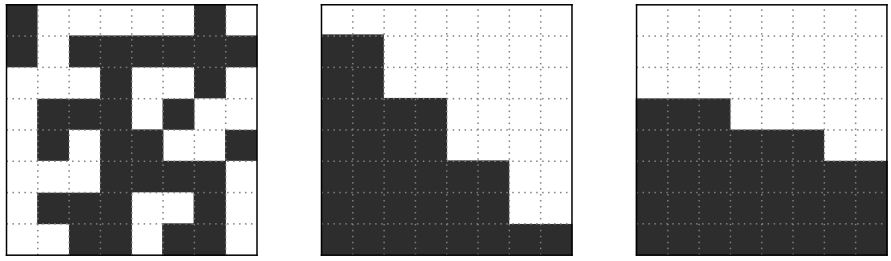


Figure 6.1.: Diagram showing how RIP-MW works over time. The most relevant parameters for RIP-MW are represented.

from the subgraph and a new one is built. As it is shown in section 6.4.4, the overhead of graph partitioning is minimal (1.18 % on average). Moreover, the partitions are scheduled asynchronously as tasks, effectively overlapping the execution of the user-level tasks with the partitioning of new subgraphs. The user can set up the window size, an initial extra amount of tasks for the first window and the size of the intersection between two consecutive windows. This intersection is used by the partitioner to reduce the algorithmic complexity and preserve data locality from previous partitions.

Once the initial subgraph is partitioned in the way we describe above, RIP-MW keeps partitioning task subgraphs by calling the method `partitionGraphSCOTCHK` from Metapart, which uses the Graph growing algorithm with support for fixed vertices [127] in a multilevel framework combined with Fiduccia-Mattheyses. The reason for using Graph growing is that Dual recursive bi-partitioning methods can perform badly under fixed vertex constraints [127, Fig. 1]. Using fixed vertices is required to exploit information from previous partitions and avoid as much as possible the mapping of tasks to NUMA domains distant from the data they consume.

Figure 6.1 shows the way RIP-MW works. First, an initial subgraph composed of the initialisation tasks plus the first window is partitioned. After this, a new subgraph is built, including the tasks in the intersection plus the new ones, until the window size is reached. Then, a second partition with fixed



(a) uniform distrib. (b) optimal (c) RIP

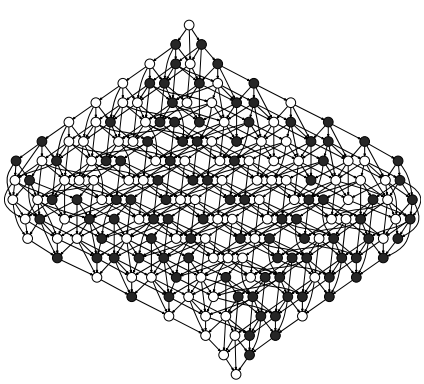
Figure 6.2.: Task and data allocations into two sockets (dark and light) on the first iteration of Gauss-Seidel (8×8 grid).

vertex constraints is carried out. The following subgraphs are built and partitioned in the same way.

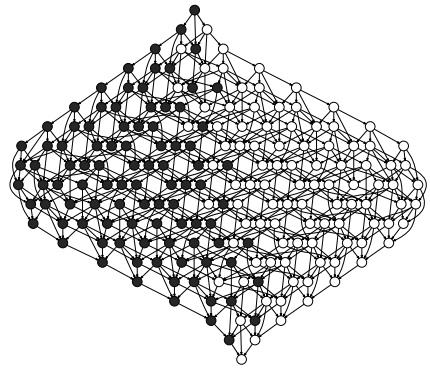
6.2.4. Benefits of graph partitioning

While simple heuristics based on data locality, like DEP, are able to produce good partitions in some scenarios, in other cases they fail to optimally partition the graph. This is especially relevant as the number of NUMA regions in the system increases. At the same time, automatic mechanisms based on graph partitioning can make the codes more architecture-agnostic and easier to program than manual assignment of the tasks to the sockets.

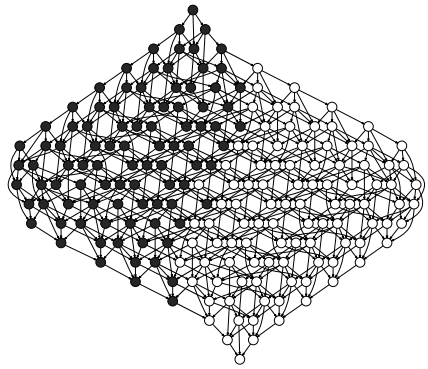
As an example, we consider the stationary heat diffusion problem using the iterative Gauss-Seidel method with a 4-element stencil (top, bottom, left, right) in an 8×8 regular grid, which corresponds to the Gauss-Seidel application later described in section 6.3.1. Each task operates over one cell of the grid. In each iteration, computations over every cell depend on the data of the four neighbouring cells, the algorithm execution follows a wavefront scheme in the direction of the main diagonal, and tasks in the same anti-diagonal are independent between them. For this reason, when targeting two sockets, the optimal partition consists in dividing the domain along the main diagonal. As a result, at each instant, half of the anti-diagonal can be executed in a different socket. Figure 6.2 shows the allocation of the data and the corresponding tasks for Gauss-Seidel for a



(a) uniform distribution + DEP



(b) programmer-given partition (optimal)



(c) RIP-DEP

Figure 6.3.: Task dependency graph corresponding to three iterations of the Gauss-Seidel code comparing a uniform distribution placement with locality awareness (DEP) to a programmer-given partition and the RIP-DEP technique in a two-socket system.

discrete uniform placement (e.g., DEP), the explained optimal partition and using a RIP method (equivalent for all RIP proposals in the case of the first iteration).

Figure 6.3 shows the same partitions expressed at the TDG level on three iterations of Gauss-Seidel. Clearly, data transfers among tasks assigned to different sockets are minimised in the expert programmer-given partition and the one obtained via graph partitioning (RIP-DEP): the graphs are cut almost *vertically*, increasing parallelism while grouping neighbouring tasks in the same socket. In contrast, the DEP approach produces a sub-optimal

partition, with more edges connecting different parts. The implications of these results in terms of the total performance are detailed in section 6.4.

6.2.5. Assumptions of the proposals

Our proposals make some general assumptions. First of all, the system needs to have a first-touch policy with local allocation, the default in the Linux kernel, and data blocks have to be initialised in page-aligned addresses using tasks in order to take full benefit from this first-touch policy. In the case of the RIP-MW techniques, barriers in the middle of a window of tasks may reduce the quality of the subgraph to partition since no new tasks are created after a barrier until all previous tasks have finished. As such, this technique benefits from a reasonably high ratio of tasks per barrier point. The RIP-DEP and RIP-MW methods need the user to set the window size as a parameter for the runtime. This is simple to achieve as intuition and experiments show that it is enough to include the initialisation tasks and the first couple of computation phases of the application (iterations in the case of iterative algorithms). An alternative solution is to apply existing techniques of automatic detection of the phases [27] and use this information to decide at execution time what is a correct window size. Another possible approach is editing the application source code by making a call to the runtime system API indicating that the partition must be done at the specific point of the call.

6.3. Experimental environment

We do the evaluations in the large ccNUMA systems presented in section 3.1.1. In all cases, we use the OpenMP programming model with a customised Nanos++ v0.10 runtime system and the companion Mercurium 2.0.0 (rev. c5a91d5) compiler [10, 145]. In the case of programs that need LAPACK, we use the open-source implementation from OpenBLAS 0.2.19 [112, 154] compiled for each architecture. Threading of the library is disabled so as not to interfere with OpenMP.

6.3.1. Manual scheduling and graph windows

This section describes the configuration for the scheduling of the parallel codes considered in this chapter, as well as the annotations for the programmer-driven scheduling using the Socket-aware (SA) scheduler, described in section 6.4. We test our proposals by considering the task-based benchmarks presented in section 3.3.1

For the *Conjugate gradient* (CG), the manual scheduling assigns tasks to sockets in a round-robin fashion. The window size corresponds to all tasks belonging to a single iteration. When applying the RIP-MW technique, the intersection is equivalent to half iteration.

In *Gauss-Seidel* the graph follows a wavefront shape, as shown in figure 6.3. The source-code-level annotations divide the columns contiguously into as many groups as NUMA domains. The window size covers the tasks of three iterations, with an intersection of a whole iteration for RIP-MW.

For *Integral histogram*, the vertical and horizontal halos used for the reduction of the histograms are allocated in a round-robin fashion, in both dimensions. The image data and scan tasks are assigned to a socket in a round-robin manner using the column identifier so that they match with the corresponding vertical halos for the programmer-driven scheduling. For the schedulers based on graph partitioning, the window size corresponds to the tasks of two iterations, with an intersection of a whole iteration.

In *Jacobi*, the source code level annotations for assigning the blocks of columns to a socket follow a round-robin approach. The double-buffer nature of Jacobi gives an embarrassingly parallel algorithm inside every iteration with a very symmetric TDG, hence it becomes simple to partition in contrast to the Gauss-Seidel code that solves the same problem. The window size includes the tasks of two iterations, with an intersection of a whole iteration.

In *NStream*, the user-level annotations for the NUMA-aware scheduler assign each array and the related tasks to a socket following a round-robin approach,

effectively assigning every array to a single NUMA node. The window size is $5N$ and the intersection is $2N$.

The *QR factorisation* manual scheduling assigns the blocks in a round-robin fashion using the row identifier, while the subsequent tasks are assigned where most blocks reside (using the row identifier). The window size is equivalent to the total number of blocks the matrix is broken into and the intersection considered by RIP-MW corresponds to two rows of blocks.

For *Red-Black*, the source-code-level annotations defining the manual scheduling divide the columns contiguously into as many groups as NUMA domains, like in Gauss-Seidel. Similarly, the window size is for the tasks of three iterations, with an intersection of a whole iteration for RIP-MW.

Finally, for *Symmetric matrix inversion* (SMI) we use the tiled task decomposition of the dense linear algebra version and the manual NUMA-aware scheduling as described by al-Omairy et al. [111], using LAPACK. The window size corresponds to the tasks of the lower triangle of the matrix (it is symmetric), with an intersection of half a triangle.

6.4. Evaluation

In this section we evaluate the performance of the proposed mechanisms considering the eight applications and two platforms described in section 6.3. Our evaluation considers five different scheduling techniques:

- *Distributed First-In First-Out* (DFIFO), unaware of data locality. In this technique, each thread has its own ready queue and tasks are assigned to threads in a round-robin manner. When the queue of a thread is empty, it applies a work stealing mechanism to get tasks from other threads.

- *Socket Aware* (SA) scheduler, which is driven by a partition expressed in terms of annotations at the source code level done by an expert programmer. SA makes use of an API call that specifies the precise socket

where tasks should run. The specific annotations of each benchmark are explained in section 6.3.1.

- The *DEP* approach, which is described in section 6.2 and represents the current state-of-the-art. All results reported in this section are normalised against DEP.
- Our two proposals based on graph partitioning algorithms: *RIP-DEP* and *RIP-MW*.

For every application, platform and method we repeat each experiment five times. In all speedup plots shown, values are averaged among the different repetitions and normalised to DEP (horizontal line at 1.0). Bar height represents the mean value, a horizontal thick line is the median, and error bars show the standard deviation. For each system configuration we include a plot of the geometric mean computed over the arithmetic means of the eight benchmarks. Our experiments are run with the following four configurations: On 24 cores of the UV100, using 8 cores per socket and 3 sockets (2 in the same blade, 1 in a different blade); on 32 cores of the bullion S16, using 8 cores per socket and 4 sockets (1 per module); on 32 cores of the bullion S16, using 4 cores per socket and 8 sockets (1 per module), and, finally, on all 288 cores of the bullion S16, using 18 cores per socket and 16 sockets (2 per module).

6.4.1. SGI Altix UV100

For the SGI Altix UV100 machine, we have done experiments using 3 sockets and 24 cores in total. All parallel runs use two sockets in the same blade (which communicate via QPI) and a third one from a different blade, although not a distant one. Results are shown in figure 6.4. On average, RIP-DEP achieves speedups of 1.03× over the DEP baseline, RIP-MW only gets up to 0.84×, while the scheduling policies driven by an expert programmer (SA) provide a 1.13× speedup. On the other hand, the locality-unaware scheduler DFIFO has a general underperformance (0.45×) except in the Symmetric matrix inversion (1.10×). The relatively small benefits shown

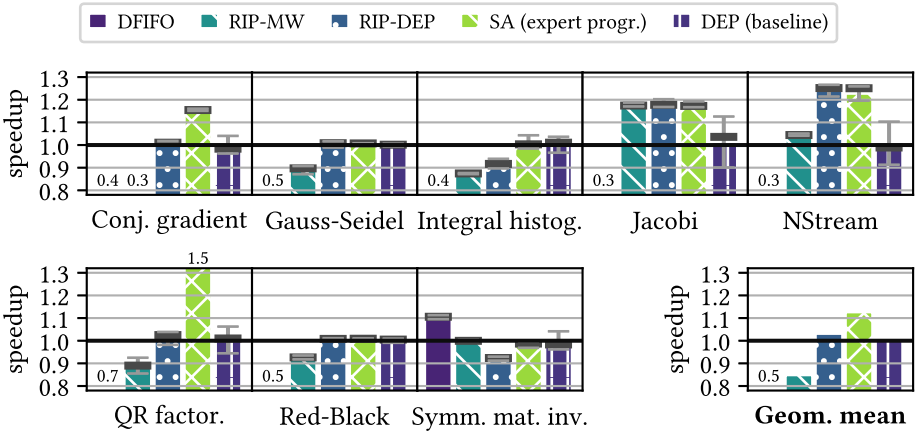


Figure 6.4.: Speedup results in the SGI Altix UV100 using 3 sockets, 24 cores. DFIFO is locality-unaware, SA is manual, the rest are automatic methods.

by RIP-DEP with respect to the DEP baseline are explained by the reduced number of NUMA domains, just three, considered in the experiments run in the UV100 machine.

However, even though the average benefits of RIP-DEP over DEP in the UV100 machine are small, there are some specific cases for which they are significant: for instance, in Jacobi the automatic partition using RIP-DEP achieves a 1.19× speedup over DEP and RIP-MW goes up to 1.18×. The benefit with RIP-DEP is more noticeable in larger machines such as the bullion S16, presented in section 6.4.2, as the number of NUMA regions increases.

6.4.2. Atos Bull bullion S16

In the case of the Atos Bull bullion S16 machine we provide experiments considering 4 sockets (32 cores), 8 sockets (32 cores) and the full system (16 sockets and 288 cores). Overall, the results in the Atos Bull bullion S16 system show how RIP-DEP provides average performance improvements of 1.08× on 4 sockets, 1.16× on 8 sockets and 1.12× on 16 sockets with respect to the state-of-the-art. RIP-MW achieves very similar improvements on

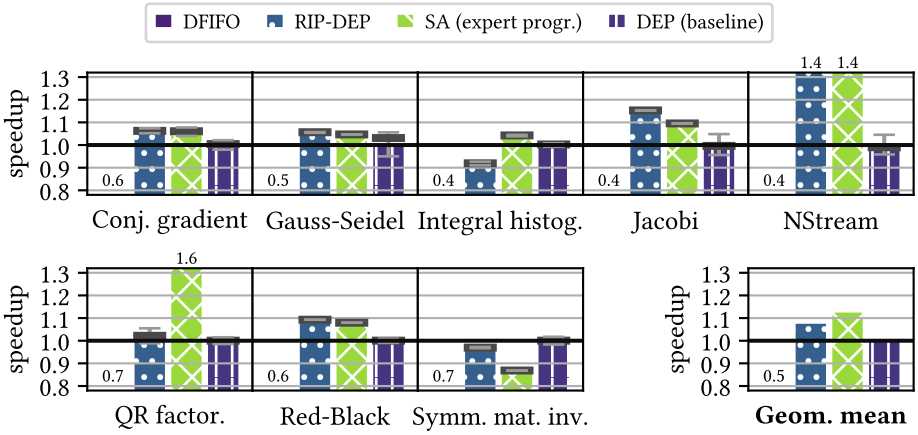


Figure 6.5.: Speedup results in the bullion S16 using 4 sockets, 32 cores. DFIFO is locality-unaware, SA is manual, the rest are automatic methods.

the experiments involving 4 and 8 sockets (i.e. 32 cores), as section 6.4.1 shows. In the case of 288 cores, RIP-MW provides worse performance than RIP-DEP since the frequent graph partitions become a significant performance bottleneck. For readability purposes, the RIP-MW technique does not appear on the experiments regarding the Atos Bull bullion S16 system.

Using four sockets

Results using four sockets in the bullion S16 system are shown in figure 6.5. Under this configuration, the average speedup obtained using RIP-DEP is of 1.08× with respect to DEP. As in the Altix machine presented in section 6.4.1, the execution times of an expert programmer-driven schedule (SA) attain a 1.13× speedup when compared with DEP. The naive DFIFO gets 0.51× performance degradation with respect to the state-of-the-art DEP approach.

RIP-DEP behaves better than the DEP baseline for the Conjugate gradient and Gauss-Seidel applications (1.05× improvement for both application) and much better for the Red-Black, Jacobi and NStream parallel codes (1.09×, 1.15× and 1.45×, respectively). These results are explained by the good

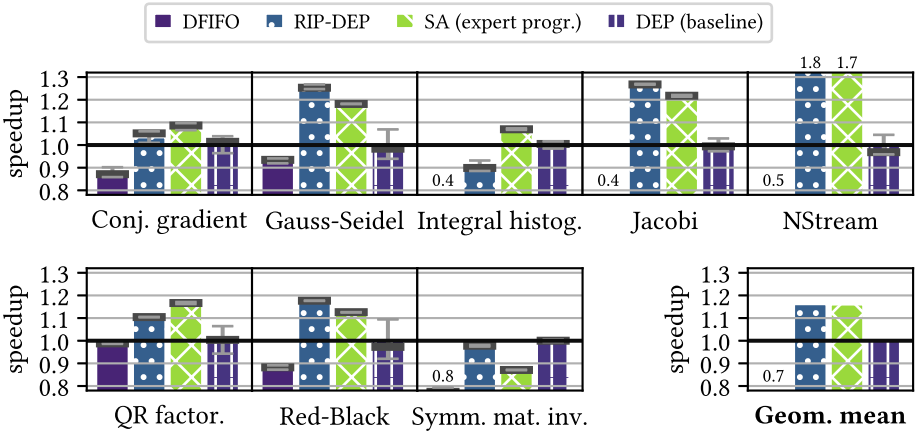


Figure 6.6.: Speedup results in the bullion S16 using 8 sockets, 32 cores. DFIFO is locality-unaware, SA is manual, the rest are automatic methods.

structure of the task graphs of these codes, which benefit from partitioning the initial subgraph and, at the same time, the iterative access pattern to the blocks of data allows for a good locality-aware propagation. In particular, Jacobi shows very good performance under this system configuration for RIP-DEP, with a higher performance than the programmer-driven partition (SA, achieving a speedup of 1.09×).

Using eight sockets

Results with eight sockets, in figure 6.6, display larger speedups of the RIP-DEP approach with respect to DEP than previous scenarios. Here, RIP-DEP attains an average speedup of 1.16× over DEP, which is matched by the expert programmer-driven partition.

Under this setting is where RIP-DEP obtains the highest benefit for Gauss-Seidel and Red-Black (1.26× and 1.18×, respectively). For the Integral histogram, the executions are somewhat slower using RIP-DEP than DEP (0.91×). The benchmark operates on a 2D domain and accumulates results in both dimensions, which creates horizontal and vertical data dependencies across tasks, which forces the partitioning algorithm to split the TDG in a way that

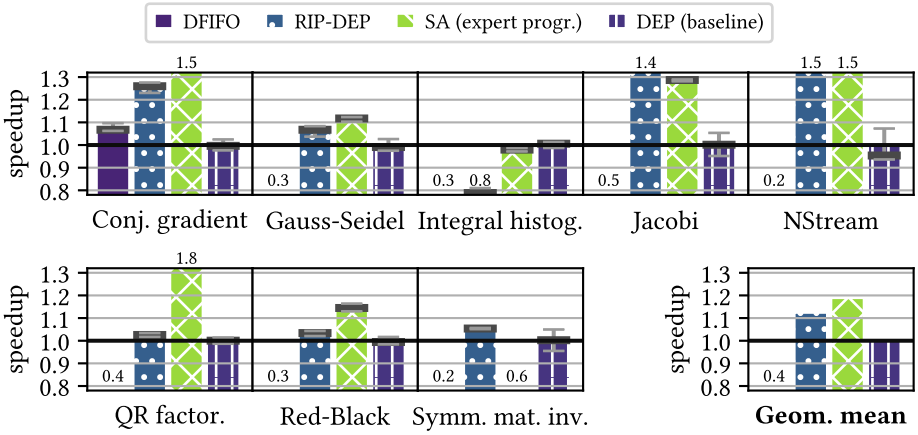


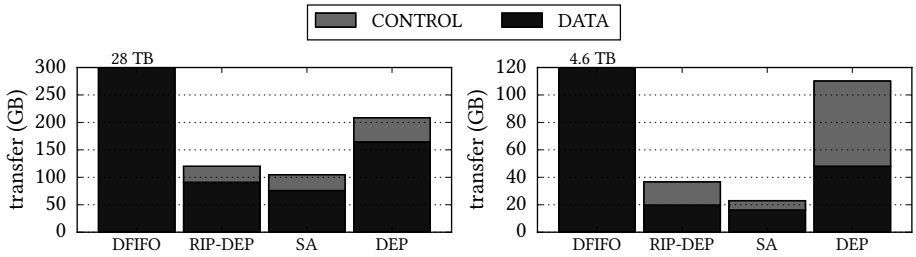
Figure 6.7.: Speedup results in the bullion S16 using all 16 sockets, 288 cores. DFIFO is locality-unaware, SA is manual, the rest are automatic methods.

the execution of some parallel tasks is serialised. However, as section 6.4.3 shows, combining both graph partitioning and a locality-aware propagation (RIP-DEP) significantly reduces data movement with respect to DEP in the case of the Integral histogram application.

Full system

Results when using all the 288 cores of the bullion S16 system are shown in figure 6.7. The input set of some applications is increased to achieve good scalability on 288 cores (e.g., CG). When running on all cores of the bullion S16 system, the RIP-DEP approach achieves a remarkable average speedup of 1.12× with respect to the state-of-the-art DEP technique. For Jacobi, RIP-DEP achieves an outstanding 1.43× speedup over DEP, only surpassed by NStream, which achieves a speedup of 1.52× when using RIP-DEP due to its simple graph.

When using all cores of the bullion S16 system, the expert programmer-driven partition (SA) obtains an average speedup of 1.19× with respect to DEP. While in some cases (e.g., QR factorisation) the non-automatic expert-driven SA partition achieves better performance than the automatic RIP-DEP



(a) Gauss-Seidel (b) Integral histogram

Figure 6.8.: Coherence traffic to and from the BCS for selected applications using 32 cores in 8 sockets in the bullion S16.

method, in the case of the Symmetric matrix inversion code the policy driven by the expert programmer performs poorly. Symmetric matrix inversion's TDG is so complex that a proper partition needs to know where data are allocated, which is impossible to be statically determined unless very simple memory allocation policies are applied, which do not provide performance benefits either. For all settings, the SA technique applied to the Symmetric matrix inversion code performs below the DEP baseline, which shows the need for dynamic and automatic methodologies in the case of very complex TDGs.

6.4.3. Reduction of coherence traffic within the bullion S16 Machine

This section provides an evaluation of the coherence traffic triggered within the bullion S16 system by all the 5 approaches considered in this chapter. This evaluation demonstrates how the RIP-DEP method we propose achieves remarkable reductions of coherence traffic. The bullion platform uses a sophisticated ccNUMA architecture composed of sets of 2 sockets grouped into entities called modules. The Bull Coherence Switch (BCS) [6], a proprietary ASIC, manages the inter-module interface and enables scaling up to a maximum of 8 modules (i.e., 16 sockets of Intel Xeon CPUs) in a single shared memory system. We use the measurement capabilities of the BCS to provide a precise analysis of the coherence traffic [24, 26]. We divide the coherence traffic in the system into two categories: *data messages*, which carry a single cache line payload, and *control messages*, which carry coherence protocol signalling activities without a data payload.

Figure 6.8 shows the differences in data transfer to and from the BCS for Gauss-Seidel and Integral histogram. Results are obtained in the bullion S16 running with 8 sockets. We have data for the other 6 applications, though we do not display them since they are qualitatively equivalent to the ones we show. When compared with the DEP baseline, SA and RIP-DEP achieve significant reductions in total coherence traffic of 1.99× and, 1.74×, respectively, in the case of Gauss-Seidel. Similarly, SA and RIP-DEP transfer 4.79× and 3.00× less data, respectively, in the case of Integral histogram. On average, using the geometric mean, SA and RIP-DEP achieve reductions of 3.08× and 2.28× with respect to DEP. These results clearly show the superiority of RIP-DEP over DEP as it dramatically reduces DEP's coherence traffic to similar levels to the partitions done by an expert programmer.

6.4.4. Load imbalance and overhead

We measure the overhead and load imbalance incurred by the different methods. Results are calculated using

$$LB = \frac{\sum_{i \in threads} \text{useful time of thread } i}{\max_{i \in threads} \{\text{useful time of thread } i\} \cdot \#threads} \cdot 100 \tag{6.2}$$

for the load balance, where the useful time of a thread is the total time the thread is executing user-level tasks. The overhead (OH) is defined as the percentage of time running runtime system routines over the wall clock time and the graph partitioning overhead (GP), included in OH, is the same ratio restricted to graph partitioning procedures. table 6.1 reports maximum, minimum and mean results computed over the eight applications described in section 6.3 running on the UV100.

Although its lack of data locality awareness makes DFIFO worse than the other approaches in terms of performance, it achieves the best load balance and the smallest runtime overhead with an average of 96.1 % and 1.77 %, respectively. RIP-DEP achieves very well balanced partitions, with an average of 88.7 %. The cost of doing an initial partition and propagating is, on average, equivalent to 3.02 % of the total execution time. In the case of

Table 6.1.: Load balance (LB), runtime overheads (OH) and graph partitioning overheads (GP) in the SGI Altix UV100 using three sockets (as percentages, %).

		mean	min.	max.
DFIFO	**LB**	96.1	78.7	99.5
	OH	1.77	0.02	8.07
RIP-MW	**LB**	90.8	85.5	97.7
	OH	3.02	0.05	6.33
	GP	1.18	0.01	3.44
RIP-DEP	**LB**	88.7	79.7	94.9
	OH	3.02	0.03	13.44
	GP	0.030	0.000	0.089
SA	**LB**	92.9	86.9	99.2
	OH	3.84	0.03	23.28
DEP	**LB**	86.5	69.9	98.3
	OH	3.12	0.04	12.71

RIP-MW repartitioning can slightly improve load balancing (90.8 %). Overall, our proposals incur minimal overheads and do not produce unbalanced partitions in the considered applications.

6.4.5. Adding page migration mechanisms

In figure 6.9, we show experiments adding page migration mechanisms to the automatic locality-aware proposals (DEP, RIP-DEP and RIP-MW). These mechanisms take care of moving the physical memory pages that contain the output data of the tasks to the socket that hosts the core executing the task. As the figure shows, page migration does not give benefit in general and is detrimental in many cases. This is mainly the case when not much data is written, which makes the migration an unnecessary overhead.

In the particular case of the QR factorisation benchmark, however, RIP-MW with page movement is the only automatic approach with positive results (1.12×). In order to understand this, consider the shape of the graph, which is a triangle pointing downwards similar to the Cholesky graph from listing 2.1, and the way RIP-MW advances, shown in figure 6.1. The partitioning

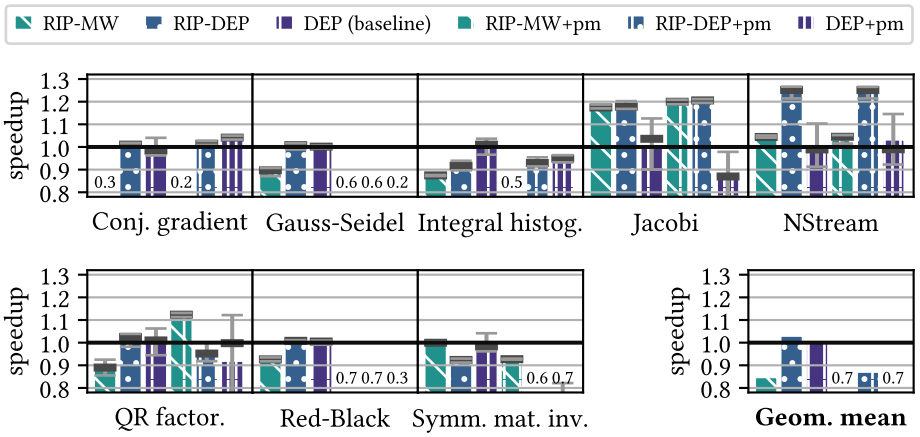


Figure 6.9.: Speedup results in the SGI Altix UV100 using 3 sockets, 24 cores, with page migration mechanisms (marked as *pm*).

algorithms generally aim at clustering connected tasks, so the first window is partitioned as three blocks (as many as sockets). At the same time, the QR algorithm does each step by working on an element of the diagonal, applying it to the rest of the matrix and discarding the whole row and column of that element afterwards. This means that load balancing mechanisms used by the graph partitioning algorithms schedule tasks created in consecutive windows in such a way that inter-socket data movement is sometimes unavoidable. For this reason, migrating the pages helps overcoming the remote accesses and makes sure that, when future partitions use the intersection, the sockets where the tasks are executed are the ones containing the data.

6.5. Summary

This chapter shows how graph partitioning methods can be leveraged to improve performance of parallel shared memory codes as well as to reduce data transfers across the system. The benefits of automatic approaches based on graph partitioning overcome the state-of-the-art without requiring expert programmer hints to drive the scheduling decisions.

Future work will go in the direction of taking even more advantage of the structure of the graph. The partitioner will be extended to get better performance with RIP-MW, which has the potential for achieving further performance improvements in applications that drastically change the structure of their TDG on runtime.

Chapter 7.

Conclusions

This thesis has presented a range of techniques leveraging information already captured in modern architectures and task-based programming models. These techniques tackle the challenges of scaling the memory hierarchy to service ever larger core counts in ccNUMA systems. This chapter details the main conclusions from the contributions of this thesis and then outlines the possibilities for future research they suggest. Finally, this chapter lists the publications resulting from this thesis and acknowledges the financial and technical support that made it possible.

7.1. Goals and contributions

The developments of HPC systems in the last couple of decades have opened many fronts. The stagnation of frequency scaling has implied the need to have multicore systems, with more and more cores every time. At the same time, this ever growing number of cores needs to have access to a large amount of memory, and to keep the benefits of using a single shared address space one solution have been NUMA systems, particularly with groups of cores having some memory as local and being able to access it faster than the rest of the memory. While this can help reduce the gap between the computation speed and the memory access speed when accessing the local memory, accesses to remote memory are much slower and can have a huge

performance penalty. This hurts the programmability of such systems when the intent is to make software with as much performance as possible.

This thesis has provided three contributions to help reduce the gap between local and remote accesses in NUMA systems, by means of both software and hardware improvements in a transparent way for the programmers in these systems. The sections below provide the conclusions of the contributions in this thesis.

7.1.1. Performance and configuration models for interactions between NUMA and hardware prefetchers

Although hardware prefetchers are designed to reduce the access latency in general by filling the caches with the needed data beforehand, not just in NUMA systems, the results in the first contribution show that their performance benefits can vary highly with the thread scheduling and data allocation. Similarly, although scheduling and allocation play a very important role in the performance of applications in NUMA systems, the best policies can change depending on the underlying hardware prefetcher configuration. This contribution proves that there is a significant performance benefit from optimising the NUMA configuration (parallelism, and placement of threads and memory pages) simultaneously with hardware prefetcher configurations (L1, L2).

The main issue is that the design space is very large. The solution we present builds a performance model and uses two reaction-based performance counter configurations (combinations of NUMA+Prefetcher and performance counters) in each system, which allow our model to make accurate predictions of the best configuration for each application while reducing the number of configurations the model has to choose among without losing performance.

The tainting of the models can be done in a new system by reusing input data from another system. This approach can be applied for online profiling

and optimisation to deliver an average of 1.68× performance increase over a NUMA-locality-optimised baseline with all prefetchers enabled.

7.1.2. Hardware prefetching for NUMA systems

Considering that hardware prefetchers can show a different behaviour (performance-wise) in NUMA systems depending on the optimisations for NUMA, as illustrated in the first contribution, the second contribution tries to solve this issue by means of a NUMA-aware hardware prefetching technique. This is a general scheme that can be used with different prefetching algorithms and uses the cache hierarchy, with larger capacity at the higher levels, to hide the latency of remote accesses.

The evaluation of the proposal is done with gem5, a cycle-accurate architectural simulator, in a two-socket NUMA system, proposing a simple stride prefetcher that is aware of NUMA effects. This NUMA-aware stride prefetcher (NASP) achieves a 1.30× speedup on average when compared to a standard stride prefetcher or 1.10× speedup against the best-performing state-of-the-art prefetcher. Since the NUMA-aware scheme is general enough, other prefetchers can be adapted to be NUMA-aware without much complexity. In the case of the best prefetcher in the state of the art, its NUMA-aware version has a speedup of 1.06× compared to the NUMA-unaware version.

7.1.3. Task-based applications in NUMA systems

One of the trends to ease the programmability of parallel systems is task-based programming, with OpenMP and OmpSs being two great examples of programming models that support this abstraction. In these programming models, parallel applications can be mapped to a directed acyclic graph with sequential code in the nodes, called tasks, and using the edges when there are data dependencies between two tasks. This changes the way scheduling works in a system, making it more difficult to build a model like in the first contribution but allowing for more flexibility when scheduling the tasks (instead of the threads) and allocating the data.

The third contribution of the thesis provides different methodologies to use graph partitioning in order to schedule task-based applications in NUMA systems, leveraging all the information available in the runtime system about the application and the underlying NUMA characteristics. The benefits of automatic approaches based on graph partitioning overcome the state-of-the-art without requiring expert programmer hints to drive the scheduling decisions. This approach attains performance improvements up to 1.52× and average improvements of 1.12× with respect to the best state-of-the-art approach for scheduling task-based applications when deployed on a 288-core shared-memory system with 16 NUMA nodes.

7.2. Future work

The work presented in this thesis suggests many possible avenues for future work. Detailed below are three which stand out as of particular potential.

- *NUCA-aware hardware prefetching and task scheduling.* In the work presented in this thesis we consider NUMA effects in a system with multiple processors. The same principles can be applied to large-scale multicore processors that suffer from *non-uniform cache access* (NUCA) effects. In these systems, the shared cache is divided in blocks that can be accessed faster by some cores than others, and usually there is associativity that allows addresses to be in different blocks of the cache.

  Scheduling threads in the various CPUs sharing a NUCA cache can have performance impacts depending on how they share data, and so can prefetching impact the performance as well when predicting an access that can be migrated from cache block to make the (predicted) access faster. Building a model that helps scheduling the threads and choosing the configuration for the prefetchers in NUCA systems is a natural step forward from the first contribution of this thesis. Similarly, a NUCA-aware prefetcher could potentially request for data more aggressively when it is in a ”far” higher-level cache block, or it

could predict when to move the data between cache blocks so that it is where it is needed. Finally, for task-based applications like in the third contribution, graph partitioning could be used to decide in which CPU core (or group of cores, if they have the same access latency to a cache block) to maximise the reuse of a cache block for an address and minimise the latency due to NUCA.

- *Runtime-driven management of coherence islands.* The contributions in this thesis all build upon a fully cache coherent baseline. As compute unit parallelisation and specialisation proceeds, memory hierarchies are likely to add features which break the coherent paradigm, such as scratchpad memories, disjoint memory spaces and coherence islands within multicore processors. The contributions developed in this thesis are interesting in such non-coherent hardware also. The second and third contributions in this thesis are also applicable to settings where shared memory abstractions are provided across disjoint memory spaces attached to specialised heterogeneous compute units, or where islands of coherence are implemented among groups of cores to enhance scalability in homogeneous architectures.

- *Hardware acceleration of TDG partitioning.* As discussed in the third contribution of this thesis, the execution time overhead of partitioning the TDG of regular HPC architectures is reduced. However, with the ongoing trend towards fine-grain tasking, the size of the TDG will significantly increase, increasing the overhead of TDG partitioning. We believe that an interesting future work could consist in designing a custom hardware accelerator to perform such task. This accelerator could be a stand-alone accelerator or an enhanced extension of a vector processing unit or a general purpose CPU via ISA extensions. Incorporating such accelerator in the exploding RISC-V ecosystem could be also very interesting.

7.3. Publications

This section lists below the publications that resulted from the work on this thesis.

- **Isaac Sánchez Barrera**, Marc Casas, and Miquel Moretó. NUMA-Aware Hardware Prefetching. Under submission.
- **Isaac Sánchez Barrera**, David Black-Schaffer, Marc Casas, Miquel Moretó, Anastasiia Stupnikova, and Mihail Popov: Modeling and optimizing NUMA effects and prefetching with machine learning. Proceedings of the 34th International Conference on Supercomputing (ICS), 2020: pages 34:1-34:13.
- Alejandro Rico, **Isaac Sánchez Barrera**, José A. Joao, Joshua Randall, Marc Casas, and Miquel Moretó: On the Benefits of Tasking with OpenMP. Proceedings of the 15th International Workshop on OpenMP (IWOMP), 2019, pages 217-230.
- **Isaac Sánchez Barrera**, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, Mateo Valero, and Marc Casas: Reducing Data Movement on Large Shared Memory Systems by Exploiting Computation Dependencies. Proceedings of the 32nd International Conference on Supercomputing (ICS), 2018, pages 207-217.
- **Isaac Sánchez Barrera**, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, and Mateo Valero: Graph partitioning applied to DAG scheduling to reduce NUMA effects. Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2018, pages 419-420.

7.4. Financial and technical support

This thesis has been supported by the Spanish Government (Severo Ochoa grants SEV2015-0493), by the Spanish Ministry of Science and Innovation

(contracts TIN2015-65316-P), by the Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328), by the RoMoL ERC Advanced Grant (grant agreement 321253) and the European HiPEAC Network of Excellence. The Mont-Blanc project receives funding from the EU's H2020 Framework Programme (H2020/2014-2020) under grant agreement numbers 671697 and 779877.

I. Sánchez Barrera has been partially supported by the Spanish Ministry of Education, Culture and Sport under fellowship number FPU15/03612, and by the Spanish Ministry of Science, Innovation and Universities under fellowship number EST18/00799.

Bibliography

[1] John A. Ackley. *Whirlwind*. In: *Encyclopedia of Computer Science*. Ed. by Edwin D. Reilly, Anthony Ralston and David Hemmendinger. 4th ed. Chichester, UK: John Wiley and Sons, Ltd., 2003, pp. 1847–1848. isbn: 978-0-470-86412-8.

[2] Charles W. Adams and P. A. Fox. *Notes on the logical design of digital computers and on special coding techniques*. Based on lectures given by Charles W. Adams as part of a special MIT course: 6.68 - Practice in the Use of Digital Computers. Massachusetts Institute of Technology. Cambridge, MA, US, 1951. url: http://www.bitsavers.org/pdf/mit/whirlwind/Notes_on_the_Logical_Design_of_Digital_Computers_and_on_Special_Coding_Techniques_1951.pdf (visited on 23/04/2021).

[3] Sam Ainsworth and Timothy M. Jones. 'An Event-Triggered Programmable Prefetcher for Irregular Workloads'. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 578–592. isbn: 978-1-4503-4911-6. https://doi.org/10.1145/3173162.3173189.

[4] Lluc Alvarez, Miquel Moretó, Marc Casas, Emilio Castillo, Xavier Martorell, Jesús Labarta, Eduard Ayguadé and Mateo Valero. 'Runtime-Guided Management of Scratchpad Memories in Multicore Architectures'. In: *2015 International Conference on Parallel Architecture and Compilation*. PACT 2015 (San Francisco, CA, US, Oct. 2015). Piscataway, NJ, US: IEEE Press, Mar. 2016, pp. 379–391. isbn: 978-1-4673-9524-3. https://doi.org/10.1109/PACT.2015.26.

[5] Lluc Alvarez, Lluís Vilanova, Miquel Moreto, Marc Casas, Marc Gonzàlez, Xavier Martorell, Nacho Navarro, Eduard Ayguadé and Mateo Valero. 'Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures'. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15 (Portland, Oregon, June 2015). New York, NY, US: Association for Computing Machinery, 2015, pp. 720–732. isbn: 978-1-4503-3402-0. https://doi.org/10.1145/2749469.2750411.

[6] *Bull bullion S16 Technical Specifications*. Technical specifications. url: https://bull.com/wp-content/uploads/2016/08/f-bullion_s16_e7v3-en2_web.pdf.

[7] Grant Ayers, Heiner Litz, Christos Kozyrakis and Parthasarathy Ranganathan. 'Classifying Memory Access Patterns for Prefetching'. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, Mar. 2020, pp. 513–526. isbn: 978-1-4503-7102-5. https://doi.org/10.1145/3373376.3378498.

[8] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan and S.K. Weeratunga. 'The Nas Parallel Benchmarks'. In: *The International Journal of Supercomputing Applications* 5.3 (Sept. 1991), pp. 63–73. issn: 0890-2720. https://doi.org/10.1177/109434209100500306.

[9] David H. Bailey, E. Barszcz, John T. Barton, D. S. Browning, R. L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, Tom A. Lasinski, Robert S. Schreiber, Horst D. Simon, V. Venkatakrishnan and Sisira K. Weeratunga. 'The NAS Parallel Benchmarks — Summary and Preliminary Results'. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing '91 (Albuquerque, New Mexico, US, Aug. 1991). New York, NY,

US: Association for Computing Machinery, 1991, pp. 158–165. ISBN: 978-0-89791-459-8. https://doi.org/10.1145/125826.125925.

[10] Jairo BALART, Alejandro DURAN, Marc GONZÀLEZ, Xavier MARTORELL, Eduard AYGUADÉ and Jesús LABARTA. 'Nanos Mercurium: A Research Compiler for OpenMP'. In: *6th European Workshop on OpenMP*. EWOMP 2004. Oct. 2004, pp. 103–109. URL: http://people.ac.upc.edu/eduard/papers/paper_a31.pdf.gz.

[11] Adrián BARREDO, Adrià ARMEJACH, Jonathan C. BEARD and Miquel MORETÓ. 'PLANAR: A Programmable Accelerator for Near-Memory Data Rearrangement'. In: *Proceedings of the 35th ACM International Conference on Supercomputing*. ICS '21 (virtual event, US, June 2021). New York, NY, US: Association for Computing Machinery, 2021. ISBN: 978-1-4503-8335-6. https://doi.org/10.1145/3447818.3460368.

[12] Adrián BARREDO, Juan M. CEBRIAN, Miquel MORETÓ, Marc CASAS and Mateo VALERO. 'Improving Predication Efficiency through Compaction/Restoration of SIMD Instructions'. In: *2020 IEEE International Symposium on High Performance Computer Architecture*. HPCA 2020 (San Diego, CA, US, Feb. 2020). Piscataway, NJ, US: IEEE Press, 2020, pp. 717–728. ISBN: 978-1-7281-6149-5. https://doi.org/10.1109/HPCA47549.2020.00064.

[13] David BENIAMINE, Matthias DIENER, Guillaume HUARD and Philippe O. A. NAVAUX. 'TABARNAC: Visualizing and Resolving Memory Access Issues on NUMA Architectures'. In: *Proceedings of the 2nd Workshop on Visual Performance Analysis*. VPA '15 (Austin, TX, US, Nov. 2015). New York, NY, US: Association for Computing Machinery, 2015. ISBN: 978-1-4503-4013-7. https://doi.org/10.1145/2835238.2835239.

[14] Peter BERGNER, Brian HALL, Alon Shalev HOUSFATER, Madhusudanan KANDASAMY, Tulio MAGNO, Alex MERICAS, Steve MUNROE, Mauricio OLIVEIRA, Bill SCHMIDT, Will SCHMIDT, Bernard King SMITH, Julian WANG, Suresh WARRIER and David WENDT. *Performance Optimization and Tuning Techniques for IBM Power Systems Processors Including IBM POWER8*. 2nd ed. RedBooks. Poughkeepsie, NY, US: IBM Corp., Aug. 2015. xx+248. ISBN: 978-0-7384-4092-7. URL: https:

//www.redbooks.ibm.com/redbooks/pdfs/sg248171.pdf (visited on 23/04/2021).

[15] Jim Beveridge and Bob Wiener. *Multithreading Applications in Win32: The Complete Guide to Threads*. Boston, MA, US: Addison-Wesley Longman Publishing Co., Inc., 1997. isbn: 978-0-201-44234-2.

[16] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis. Princeton University, Jan. 2011.

[17] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh and Kai Li. 'The PARSEC Benchmark Suite: Characterization and Architectural Implications'. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. Oct. 2008. https://doi.org/10.1145/1454115.1454128.

[18] Erik Boman, Karen Devine, Lee Ann Fisk, Robert Heaphy, Bruce Hendrickson, Vitus Leung, Courtenay Vaughan, Ümit V. Çatalyürek, Doruk Bozdag and William Mitchell. *Zoltan*. Sandia National Laboratories. 1999. url: http://www.cs.sandia.gov/Zoltan.

[19] Greg Bronevetsky, John Gyllenhaal and Bronis R. De Supinski. 'CLOMP: Accurately Characterizing OpenMP Application Overheads'. In: *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*. IWOMP '08. West Lafayette, Indiana, USA: Springer, 2008, pp. 13–25. isbn: 978-3-540-79560-5. https://doi.org/10.1007/978-3-540-79561-2_2.

[20] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier and Raymond Namyst. 'ForestGOMP: an efficient OpenMP environment for NUMA architectures'. In: *International Journal of Parallel Programming* 38.5 (2010), pp. 418–439. issn: 1573-7640. https://doi.org/10.1007/s10766-010-0136-3.

[21] Iulian Brumar, Marc Casas, Miquel Moretó, Mateo Valero and Gurindar S. Sohi. 'ATM: Approximate Task Memoization in the Runtime System'. In: *2017 IEEE International Parallel and Distributed Processing Symposium*. IPDPS 2017 (Orlando, FL, US, May–June 2017). Piscataway, NJ, US: IEEE Press, 2017, pp. 1140–1150. isbn: 978-1-5386-3914-6. https://doi.org/10.1109/IPDPS.2017.49.

[22] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders and Christian Schulz. 'Recent Advances in Graph Partitioning'. In: *Algorithm Engineering: Selected Results and Surveys*. Ed. by Lasse Kliemann and Peter Sanders. Vol. 9220. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 117–158. isbn: 978-3-319-49486-9. https://doi.org/10.1007/978-3-319-49487-6_4.

[23] Alfredo Buttari, Julien Langou, Jakub Kurzak and Jack Dongarra. 'Parallel Tiled QR Factorization for Multicore Architectures'. In: *Concurr. Comput. Pract. Exp.* 20.13 (July 2008), pp. 1573–1590. issn: 1532-0626. https://doi.org/10.1002/cpe.1301.

[24] Paul Caheny, Lluc Alvarez, Said Derradji, Mateo Valero, Miquel Moretó and Marc Casas. 'Reducing Cache Coherence Traffic with a NUMA-Aware Runtime Approach'. In: *IEEE Transactions on Parallel and Distributed Systems* 29.5 (2018), pp. 1174–1187. issn: 1045-9219. https://doi.org/10.1109/TPDS.2017.2787123.

[25] Paul Caheny, Lluc Alvarez, Mateo Valero, Miquel Moretó and Marc Casas. 'Runtime-Assisted Cache Coherence Deactivation in Task Parallel Programs'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '18 (Dallas, TX, US, Nov. 2018). Piscataway, NJ, US: IEEE Press, Mar. 2019. isbn: 978-1-5386-8384-2. https://doi.org/10.1109/SC.2018.00038.

[26] Paul Caheny, Marc Casas, Miquel Moretó, Hervé Gloaguen, Maxime Saintes, Eduard Ayguadé, Jesús Labarta and Mateo Valero. 'Reducing Cache Coherence Traffic with Hierarchical Directory Cache and NUMA-Aware Runtime Scheduling'. In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. PACT '16 (Haifa, IR, Sept. 2016). New York, NY, US: Association for Computing Machinery, 2016, pp. 275–286. isbn: 978-1-4503-4121-9. https://doi.org/10.1145/2967938.2967962.

[27] Marc Casas, Rosa M. Badia and Jesús Labarta. 'Automatic Phase Detection and Structure Extraction of MPI Applications'. In: *The International Journal of High Performance Computing Applications* 24.3 (Aug. 2010), pp. 335–360. issn: 1094-3420. https://doi.org/10.1177/1094342009360039.

[28] Marc Casas, Miquel Moretó, Lluc Alvarez, Emilio Castillo, Dimitrios Chasapis, Timothy Hayes, Luc Jaulmes, Oscar Palomar, Osman Unsal, Adrián Cristal, Eduard Ayguadé, Jesús Labarta and Mateo Valero. 'Runtime-Aware Architectures'. In: *Euro-Par 2015: Parallel Processing*. 21st International Conference on Parallel and Distributed Computing. Euro-Par 2015 (Vienna, AT, 2015). Lecture Notes in Computer Science 9233. Berlin and Heidelberg, DE: Springer, Aug. 2015, pp. 16–27. isbn: 978-3-662-48096-0. https://doi.org/10.1007/978-3-662-48096-0_2.

[29] Emilio Castillo, Lluc Àlvarez, Miquel Moretó, Marc Casas, Enrique Vallejo, José Luis Bosque, Ramón Beivide and Mateo Valero. 'Architectural Support for Task Dependence Management with Flexible Software Scheduling'. In: *2018 IEEE International Symposium on High Performance Computer Architecture*. HPCA 2018 (Vienna, AT, Feb. 2018). Piscataway, NJ, US: IEEE Press, 2018, pp. 283–295. https://doi.org/10.1109/HPCA.2018.00033.

[30] Emilio Castillo, Miquel Moreto, Marc Casas, Lluc Alvarez, Enrique Vallejo, Kallia Chronaki, Rosa Badia, Jose Luis Bosque, Ramon Beivide, Eduard Ayguade, Jesus Labarta and Mateo Valero. 'CATA: Criticality Aware Task Acceleration for Multicore Processors'. In: *2016 IEEE International Parallel and Distributed Processing Symposium)*. IPDPS 2016 (Chicago, IL, US, May 2016). Piscataway, NJ, US: IEEE Press, 2016, pp. 413–422. isbn: 978-1-5090-2140-6. https://doi.org/10.1109/IPDPS.2016.49.

[31] Ümit V. Çatalyürek. *PaToH Graph Partitioner*. 2011. url: http://www.cc.gatech.edu/~umit/software.html%5C#patoh.

[32] Ümit V. Çatalyürek and Cevdet Aykanat. 'Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication'. In: *IEEE Trans. Parallel Distrib. Syst.* 10.7 (July 1999), pp. 673–693. ISSN: 1045-9219. https://doi.org/10.1109/71.780863.

[33] John Cavazos, Christophe Dubach, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, Grigori Fursin and Olivier Temam. 'Automatic Performance Model Construction for the Fast Software Exploration of New Hardware Designs'. In: *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES '06. Seoul, Korea: ACM, 2006, pp. 24–34. ISBN: 978-1-59593-543-4. https://doi.org/10.1145/1176760.1176765.

[34] *Charm++ Programming Model*. 2016. URL: http://charmplusplus.org/.

[35] Dimitrios Chasapis, Marc Casas, Miquel Moretó, Martin Schulz, Eduard Ayguadé, Jesus Labarta and Mateo Valero. 'Runtime-Guided Mitigation of Manufacturing Variability in Power-Constrained Multi-Socket NUMA Nodes'. In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS '16 (Istanbul, TR, June 2016). New York, NY, US: Association for Computing Machinery, 2016. ISBN: 978-1-4503-4361-9. https://doi.org/10.1145/2925426.2926279.

[36] Dimitrios Chasapis, Miquel Moretó, Martin Schulz, Barry Rountree, Mateo Valero and Marc Casas. 'Power Efficient Job Scheduling by Predicting the Impact of Processor Manufacturing Variability'. In: *Proceedings of the ACM International Conference on Supercomputing*. ICS '19 (Phoenix, AZ, US, June 2019). New York, NY, US: Association for Computing Machinery, 2019, pp. 296–307. ISBN: 978-1-4503-6079-1. https://doi.org/10.1145/3330345.3330372.

[37] Sanjay Chatterjee, Nick Vrvilo, Zoran Budimlic, Kathleen Knobe and Vivek Sarkar. 'Declarative Tuning for Locality in Parallel Programs'. In: *45th International Conference on Parallel Processing*. ICPP 2016. IEEE, Aug. 2016, pp. 452–457. https://doi.org/10.1109/ICPP.2016.58.

[38] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee and Kevin Skadron. 'Rodinia: A Benchmark Suite for Heterogeneous Computing'. In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*. IISWC '09. Austin, Texas, USA: IEEE, 2009, pp. 44–54. ISBN: 978-1-4244-5156-2. https://doi.org/10.1109/IISWC.2009.5306797.

[39] Tien-Fu Chen and Jean-Loup Baer. 'Effective hardware-based data prefetching for high-performance processors'. In: *IEEE Transactions on Computers* 44.5 (May 1995), pp. 609–623. ISSN: 0018-9340. https://doi.org/10.1109/12.381947.

[40] Kallia Chronaki, Alejandro Rico, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta and Mateo Valero. 'Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures'. In: *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, CA, US, June 2015). New York, NY, US: Association for Computing Machinery, 2015, pp. 329–338. ISBN: 978-1-4503-3559-1. https://doi.org/10.1145/2751205.2751235.

[41] Henry Cook, Miquel Moretó, Sarah Bird, Khanh Dao, David A. Patterson and Krste Asanović. 'A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-Efficiency While Preserving Responsiveness'. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. Tel-Aviv, Israel: ACM, 2013, pp. 308–319. ISBN: 978-1-4503-2079-5. https://doi.org/10.1145/2485922.2485949.

[42] Idriss Daoudi, Philippe Virouleau, Thierry Gautier, Samuel Thibault and Olivier Aumage. 'sOMP: Simulating OpenMP Task-Based Applications with NUMA Effects'. In: *OpenMP: Portable Multi-Level Parallelism on Modern Systems*. Ed. by Kent Milfeld, Bronis R. de Supinski, Lars Koesterke and Jannis Klinkenberg. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 197–211. ISBN: 978-3-030-58144-2. https://doi.org/10.1007/978-3-030-58144-2_13.

[43] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema and Mark Roth. 'Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems'. In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13 (Houston, Texas, US, Apr. 2013). New York, NY, US: Association for Computing Machinery, 2013, pp. 381–394. isbn: 978-1-4503-1870-9. https://doi.org/10.1145/2451116.2451157.

[44] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna and Christian Le. 'RAPL: Memory Power Estimation and Capping'. In: *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*. ISLPED '10. Austin, Texas, USA: ACM, 2010, pp. 189–194. isbn: 978-1-4503-0146-6. https://doi.org/10.1145/1840845.1840883.

[45] Peter J. Denning. 'Virtual Memory'. In: *ACM Computing Surveys* 2.3 (Sept. 1970), pp. 153–189. issn: 0360-0300. https://doi.org/10.1145/356571.356573.

[46] Peter J. Denning. *Virtual memory*. In: *Encyclopedia of Computer Science*. Ed. by Edwin D. Reilly, Anthony Ralston and David Hemmendinger. 4th ed. Chichester, UK: John Wiley and Sons, Ltd., 2003, pp. 1832–1835. isbn: 978-0-470-86412-8.

[47] Nicolas Denoyelle, Brice Goglin, Emmanuel Jeannot and Thomas Ropars. 'Data and Thread Placement in NUMA Architectures: A Statistical Learning Approach'. In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019 (Kyoto, JP, Aug. 2019). New York, NY, US: Association for Computing Machinery, 2019. isbn: 978-1-4503-6295-5. https://doi.org/10.1145/3337821.3337893.

[48] Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux and Israel Koren. 'Affinity-Based Thread and Data Mapping in Shared Memory Systems'. In: *ACM Computing Surveys* 49.4 (Dec. 2016). issn: 0360-0300. https://doi.org/10.1145/3006385.

[49] Matthias Diener, Eduardo H.M. Cruz, Philippe O.A. Navaux, Anselm Busse and Hans-Ulrich Heiß. 'KMAF: Automatic Kernel-Level Management of Thread and Data Affinity'. In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT '14 (Edmonton, AB, Canada, Aug. 2014). New York, NY, US: Association for Computing Machinery, 2014, pp. 277–288. isbn: 978-1-4503-2809-8. https://doi.org/10.1145/2628071.2628085.

[50] Matthias Diener, Eduardo HM Cruz, Laércio L. Pilla, Fabrice Dupros and Philippe O.A. Navaux. 'Characterizing communication and page usage of parallel applications for thread and data mapping'. In: *Performance Evaluation* (June 2015), pp. 18–36. issn: 0166-5316. https://doi.org/10.1016/j.peva.2015.03.001.

[51] Vladimir Dimić, Miquel Moretó, Marc Casas, Jan Ciesko and Mateo Valero. 'RICH: Implementing Reductions in the Cache Hierarchy'. In: *Proceedings of the 34th ACM International Conference on Supercomputing*. ICS '20 (Barcelona, ES, June 2020). New York, NY, US: Association for Computing Machinery, 2020. isbn: 978-1-4503-7983-0. https://doi.org/10.1145/3392717.3392736.

[52] Vladimir Dimić, Miquel Moretó, Marc Casas and Mateo Valero. 'Runtime-Assisted Shared Cache Insertion Policies Based on Re-reference Intervals'. In: *Euro-Par 2017: Parallel Processing*. 23rd International Conference on Parallel and Distributed Computing. Euro-Par 2017 (Santiago de Compostela, ES, Aug.–Sept. 2017). Ed. by Francisco F. Rivera, Tomás F. Pena and José C. Cabaleiro. Lecture Notes in Computer Science 10417. Cham, CH: Springer International Publishing, 2017, pp. 247–259. isbn: 978-3-319-64203-1. https://doi.org/10.1007/978-3-319-64203-1_18.

[53] Andi Drebes, Karine Heydemann, Nathalie Drach, Antoniu Pop and Albert Cohen. 'Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages'. In: *ACM Transactions on Architure and Code Optimization* 11.3 (2014), p. 30. issn: 1544-3566. https://doi.org/10.1145/2641764.

[54] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen and Nathalie Drach. 'Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management'. In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. PACT '16 (Haifa, IR, Sept. 2016). New York, NY, US: ACM, 2016, pp. 125–137. ISBN: 978-1-4503-4121-9. https://doi.org/10.1145/2967938.2967946.

[55] Juan J Durillo, Philipp Gschwandtner, Klaus Kofler and Thomas Fahringer. 'Multi-Objective region-Aware optimization of parallel programs'. In: *Parallel Computing* 83 (2019), pp. 3–21. https://doi.org/10.1016/j.parco.2018.03.010.

[56] Alexandre E. Eichenberger, John Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh and Daniel Lorenz. 'OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis'. In: *OpenMP in the Era of Low Power Devices and Accelerators*. Ed. by Alistair P. Rendell, Barbara M. Chapman and Matthias S. Müller. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 171–185. ISBN: 978-3-642-40698-0. https://doi.org/10.1007/978-3-642-40698-0_13.

[57] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta and Mateo Valero. 'Task Superscalar: An Out-of-Order Task Pipeline'. In: *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-43 (Atlanta, GA, US, Dec. 2010). Piscataway, NJ, US: IEEE Press, 2010, pp. 89–100. ISBN: 978-0-7695-4299-7. https://doi.org/10.1109/MICRO.2010.13.

[58] Michael J. Flynn. 'Very high-speed computing systems'. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909. https://doi.org/10.1109/PROC.1966.5273.

[59] Victor Garcia, Alejandro Rico, Carlos Villavieja, Paul Carpenter, Nacho Navarro and Alex Ramirez. 'Adaptive Runtime-Assisted Block Prefetching on Chip-Multiprocessors'. In: *International Journal of*

Parallel Programming 45.3 (1st June 2017), pp. 530–550. ISSN: 1573-7640. https://doi.org/10.1007/s10766-016-0431-8.

[60] Constantino Gómez, Francesc Martínez, Adrià Armejach, Miquel Moretó, Filippo Mantovani and Marc Casas. 'Design Space Exploration of Next-Generation HPC Machines'. In: *2019 IEEE International Parallel and Distributed Processing Symposium*. IPDPS 2019 (Rio de Janeiro, BR, May 2019). IEEE Press, 2019, pp. 54–65. https://doi.org/10.1109/IPDPS.2019.00017.

[61] Marius Grannaes, Magnus Jahre and Lasse Natvig. 'Multi-Level Hardware Prefetching Using Low Complexity Delta Correlating Prediction Tables with Partial Matching'. In: *High Performance Embedded Architectures and Compilers*. Ed. by Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi and Xavier Martorell. HiPEAC 2010. Berlin, Heidelberg: Springer, 2010, pp. 247–261. ISBN: 978-3-642-11515-8. https://doi.org/10.1007/978-3-642-11515-8_19.

[62] Thomas Grass, César Allande, Adrià Armejach, Alejandro Rico, Eduard Ayguadé, Jesus Labarta, Mateo Valero, Marc Casas and Miquel Moretó. 'MUSA: A Multi-Level Simulation Approach for Next-Generation HPC Machines'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 526–537. ISBN: 978-1-4673-8815-3. https://doi.org/10.1109/SC.2016.44.

[63] Jason Hiebel, Laura E. Brown and Zhenlin Wang. 'Machine Learning for Fine-Grained Hardware Prefetcher Control'. In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019 (Kyoto, JP, Aug. 2019). New York, NY, US: Association for Computing Machinery, 2019. ISBN: 978-1-4503-6295-5. https://doi.org/10.1145/3337821.3337854.

[64] Rodney E. Hooker, Douglas R. Reed, John Michael Greer and Colin Eddy. 'Multiple Data Prefetchers That Defer to One Another Based on Prefetch Effectiveness by Memory Access Type'. US9817764B2. Nov. 2017. URL: https://patents.google.com/patent/US9817764B2/en (visited on 14/05/2020).

[65] Rodney E. Hooker, Douglas R. Reed, John Michael Greer and Colin Eddy. 'Prefetching with Level of Aggressiveness Based on Effectiveness by Memory Access Type'. US10387318B2. Aug. 2019. URL: https://patents.google.com/patent/US10387318B2/en (visited on 14/05/2020).

[66] Rich Hornung, Jeff Keasler and Maya Gokhale. *Hydrodynamics Challenge Problem*. Tech. rep. LLNL-TR-490254. Livermore, CA, US: Lawrence Livermore National Laboratory, July 2011, pp. 1–17. URL: https://computing.llnl.gov/projects/co-design/spec-7.pdf.

[67] Connor Imes, Steven Hofmeyr, Dong In D. Kang and John Paul Walters. 'A Case Study and Characterization of a Many-Socket, Multi-Tier NUMA HPC Platform'. In: *IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. Virtual event: IEEE Press, Nov. 2020. https://doi.org/10.1109/LLVMHPCHiPar51896.2020.00013.

[68] International Business Machines Corporation. *IBM System/360 System Summary*. 13th ed. IBM Corp. Poughkeepsie, NY, US, 1974. URL: http://bitsavers.org/pdf/ibm/360/systemSummary/GA22-6810-12_360sysSumJan74.pdf (visited on 23/04/2021).

[69] Yasuo Ishii, Mary Inaba and Kei Hiraki. 'Access Map Pattern Matching for High Performance Data Cache Prefetch'. In: *Journal of Instruction-Level Parallelism* 13 (Jan. 2011), p. 24. ISSN: 1942-9525. URL: https://www.jilp.org/vol13/v13paper3.pdf.

[70] Ashok Jagannathan, Prabhat Jain, Krishna N. Vinod and Avinash Sodani. 'Instruction and Logic for Prefetcher Throttling Based on Counts of Memory Accesses to Data Sources'. US9507596B2. Nov. 2016. URL: https://patents.google.com/patent/US9507596B2/en (visited on 14/05/2020).

[71] Akanksha Jain and Calvin Lin. 'Linearizing Irregular Memory Accesses for Improved Correlated Prefetching'. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. New York, NY, USA: Association for Computing

Machinery, Dec. 2013, pp. 247–259. ISBN: 978-1-4503-2638-4. https://doi.org/10.1145/2540708.2540730.

[72] Luc Jaulmes, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta and Mateo Valero. 'Exploiting Asynchrony from Exact Forward Recovery for DUE in Iterative Solvers'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15 (Austin, TX, US, Nov. 2015). New York, NY, US: Association for Computing Machinery, 2015. ISBN: 978-1-4503-3723-6. https://doi.org/10.1145/2807591.2807599.

[73] Luc Jaulmes, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, Mateo Valero and Marc Casas. 'Asynchronous and Exact Forward Recovery for Detected Errors in Iterative Solvers'. In: *IEEE Transactions on Parallel and Distributed Systems* 29.9 (2018), pp. 1961–1974. ISSN: 1045-9219. https://doi.org/10.1109/TPDS.2018.2817524.

[74] Luc Jaulmes, Miquel Moretó, Mateo Valero and Marc Casas. 'A Vulnerability Factor for ECC-protected Memory'. In: *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design*. IOLTS 2019 (Rhodes, GR, July 2019). Piscataway, NJ, US: IEEE Press, Oct. 2019, pp. 176–181. ISBN: 978-1-7281-2490-2. https://doi.org/10.1109/IOLTS.2019.8854397.

[75] Luc Jaulmes, Miquel Moretó, Mateo Valero, Mattan Erez and Marc Casas. 'Runtime-Guided ECC Protection Using Online Estimation of Memory Vulnerability'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20 (Atlanta, GA, US, Nov. 2020). Piscataway, NJ, US: IEEE Press, 2020. ISBN: 978-1-7281-9998-6. https://doi.org/10.1109/SC41405.2020.00080.

[76] Victor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose and Francis P. O'Connell. 'Making Data Prefetch Smarter: Adaptive Prefetching on POWER7'. In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*. PACT '12. Minneapolis, Minnesota, USA:

ACM, 2012, pp. 137–146. ISBN: 978-1-4503-1182-3. https://doi.org/10.1145/2370816.2370837.

[77] Doug JOSEPH and Dirk GRUNWALD. 'Prefetching Using Markov Predictors'. In: *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, CO, US, 1997). New York, NY, US: Association for Computing Machinery, 1997, pp. 252–263. ISBN: 978-0-89791-901-2. https://doi.org/10.1145/264107.264207.

[78] Norman P. JOUPPI. 'Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers'. In: *Proceedings of the 17th Annual International Symposium on Computer Architecture*. ISCA '90 (Seattle, WA, US, 1990). New York, NY, US: Association for Computing Machinery, 1990, pp. 364–373. ISBN: 978-0-89791-366-9. https://doi.org/10.1145/325164.325162.

[79] Laxmikant V. KALÉ and Sanjeev KRISHNAN. 'Parallel Programming with Message-Driven Objects'. In: *Parallel Programming Using C++*. Ed. by Gregory V. Wilson and Paul Lu. Cambridge, MA, USA: MIT Press, 1996, pp. 175–213. ISBN: 978-0-262-73118-8.

[80] Ian KARLIN, Jeff KEASLER and Rob NEELY. *LULESH 2.0 Updates and Changes*. Tech. rep. LLNL-TR-641973. Livermore, CA, US: Lawrence Livermore National Laboratory, Aug. 2013, pp. 1–9. URL: https://computing.llnl.gov/projects/co-design/lulesh2.0_changes1.pdf.

[81] George KARYPIS and Vipin KUMAR. *Metis Graph Partitioner*. 1997. URL: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[82] George KARYPIS and Vipin KUMAR. 'A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs'. In: *SIAM J. Sci. Comput.* 20.1 (Jan. 1998), pp. 359–392. ISSN: 1064-8275. https://doi.org/10.1137/S1064827595287997.

[83] George KARYPIS and Vipin KUMAR. 'Multilevel k-Way Hypergraph Partitioning'. In: *36th Annual ACM/IEEE Design Automation Conference*. DAC '99. New York, NY, USA: ACM, 1999, pp. 343–348. ISBN: 978-1-58113-109-3. https://doi.org/10.1145/309847.309954.

[84] George Karypis and Vipin Kumar. *hMetis Partitioning Software*. 2007. URL: http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview.

[85] Muneeb Khan, Michael A. Laurenzanoy, Jason Marsy, Erik Hagersten and David Black-Schaffer. 'AREP: Adaptive Resource Efficient Prefetching for Maximizing Multicore Performance'. In: *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*. PACT '15. San Francisco, California, USA: IEEE, 2015, pp. 367–378. ISBN: 978-1-4673-9524-3. https://doi.org/10.1109/PACT.2015.35.

[86] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A. L. Narasimha Reddy, Chris Wilkerson and Zeshan Chishti. 'Path Confidence Based Lookahead Prefetching'. In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, Oct. 2016, pp. 1–12. https://doi.org/10.1109/MICRO.2016.7783763.

[87] Sanjeev Kumar, Christopher J. Hughes and Anthony Nguyen. 'Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors'. In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ISCA '07 (San Diego, CA, USA, June 2007). New York, NY, US: Association for Computing Machinery, 2007, pp. 162–173. ISBN: 978-1-59593-706-3. https://doi.org/10.1145/1250662.1250683.

[88] James Laudon and Daniel Lenoski. 'The SGI Origin: A ccNUMA Highly Scalable Server'. In: *Proceedings of the 24th Annual International Symposium on Computer Architecture*. ISCA '97 (Denver, Colorado, US, 1997). New York, NY, US: Association for Computing Machinery, 1997, pp. 241–251. ISBN: 978-0-89791-901-2. https://doi.org/10.1145/264107.264206.

[89] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh and Sally A. McKee. 'Methods of Inference and Learning for Performance Modeling of Parallel Applications'. In: *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and*

Practice of Parallel Programming. PPoPP '07. San Jose, California, USA: ACM, 2007, pp. 249–258. ISBN: 978-1-59593-602-8. https://doi.org/10.1145/1229428.1229479.

[90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz and Monica S. Lam. 'The Stanford Dash Multiprocessor'. In: *Computer* 25.3 (Mar. 1992), pp. 63–79. ISSN: 0018-9162. https://doi.org/10.1109/2.121510.

[91] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen and Norman P. Jouppi. 'McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures'. In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 42. New York, NY, USA: Association for Computing Machinery, Dec. 2009, pp. 469–480. ISBN: 978-1-60558-798-1. https://doi.org/10.1145/1669112.1669172.

[92] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman and Norman P. Jouppi. 'CACTI-P: Architecture-Level Modeling for SRAM-Based Structures with Advanced Leakage Reduction Techniques'. In: *Proceedings of the International Conference on Computer-Aided Design*. ICCAD '11. San Jose, California: IEEE Press, Nov. 2011, pp. 694–701. ISBN: 978-1-4577-1398-9. https://doi.org/10.1109/ICCAD.2011.6105405.

[93] Tan Li, Yufei Ren, Dantong Yu and Shudong Jin. 'Analysis of NUMA Effects in Modern Multicore Systems for the Design of High-Performance Data Transfer Applications'. In: *Future Generation Computer Systems* 74 (Sept. 2017), pp. 41–50. ISSN: 0167-739X. https://doi.org/10.1016/j.future.2017.04.001.

[94] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu and Hucheng Zhou. 'Machine Learning-Based Prefetch Optimization for Data Center Applications'. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. Portland, Oregon, USA: ACM, 2009. ISBN: 978-1-60558-744-8. https://doi.org/10.1145/1654059.1654116.

[95] Peng Liu, Jiyang Yu and Michael C. Huang. 'Thread-Aware Adaptive Prefetcher on Multicore Systems: Improving the Performance for Multithreaded Workloads'. In: *ACM Transactions on Architecture and Code Optimization* 13.1 (Mar. 2016), 13:1–13:25. ISSN: 1544-3566. https://doi.org/10.1145/2890505.

[96] Gabriel H. Loh. 'Cache Prefetching from Non-Uniform Memories'. US8621157B2. Dec. 2013. URL: https://patents.google.com/patent/US8621157B2/en (visited on 09/07/2020).

[97] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon and Éder F. Zulian. 'The gem5 Simulator: Version 20.0+'. In: (2020). arXiv: 2007.03152 [cs.AR].

[98] Zoltan Majo and Thomas R. Gross. 'Matching Memory Access Patterns and Data Placement for NUMA Systems'. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO '12. San Jose, California, USA: ACM, 2012, pp. 230–241. ISBN: 978-1-4503-1206-6. https://doi.org/10.1145/2259016.2259046.

[99] Zoltan Majo and Thomas R. Gross. '(Mis)Understanding the NUMA Memory System Performance of Multithreaded Workloads'. In: *Proceedings of the 2013 IEEE International Symposium on Workload Characterization*. IISWC '13 (Portland, OR, US, Sept. 2013). Piscataway, NJ, US: IEEE Press, 2013, pp. 11–22. ISBN: 978-1-4799-0555-3. https://doi.org/10.1109/IISWC.2013.6704666.

[100] Madhavan Manivannan, Vassilis Papaefstathiou, Miquel Pericàs and Per Stenström. 'RADAR: Runtime-Assisted Dead Region Management for Last-Level Caches'. In: *2016 IEEE International Symposium on High Performance Computer Architecture*. HPCA 2016 (Barcelona, ES, Mar. 2016). Piscataway, NJ, US: IEEE Press, 2016, pp. 644–656. ISBN: 978-1-4673-9211-2. https://doi.org/10.1109/HPCA.2016.7446101.

[101] Madhavan Manivannan, Miquel Pericás, Vassilis Papaefstathiou and Per Stenström. 'Global Dead-Block Management for Task-Parallel Programs'. In: *ACM Transactions on Architure and Code Optimization* 15.3 (2018). ISSN: 1544-3566. https://doi.org/10.1145/3234337.

[102] Larry McAvoy and Carl Staelin. 'Lmbench: Portable Tools for Performance Analysis'. In: *USENIX 1996 Annual Technical Conference*. USENIX, 1996, pp. 279–294. URL: https://www.usenix.org/legacy/publications/library/proceedings/sd96/mcvoy.html.

[103] John D. McCalpin. 'Memory Bandwidth and Machine Balance in Current High Performance Computers'. In: *IEEE Comput. Soc. Tech. Comm. Comput. Archit. TCCA Newsl.* (1995), pp. 19–25. URL: http://www.cs.virginia.edu/stream/.

[104] Puya Memarzia, Suprio Ray and Virendra C Bhavsar. 'Toward Efficient In-memory Data Analytics on NUMA Systems'. In: *arXiv e-prints* (2019). arXiv: 1908.01860v2 [cs.DB].

[105] Pierre Michaud. 'Best-Offset Hardware Prefetching'. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Mar. 2016, pp. 469–480. https://doi.org/10.1109/HPCA.2016.7446087.

[106] Sparsh Mittal. 'A Survey of Recent Prefetching Techniques for Processor Caches'. In: *ACM Computing Surveys* 49.2 (Aug. 2016). ISSN: 0360-0300. https://doi.org/10.1145/2907071.

[107] Kyle J. Nesbit and James E. Smith. 'Data Cache Prefetching Using a Global History Buffer'. In: *Proceedings of the 10th International Symposium on High Performance Computer Architecture*. 10th International Symposium on High Performance Computer Architecture. HPCA '04 (Madrid, ES, Feb. 2004). Piscataway, NJ, US: IEEE Press, 2004, pp. 96–96. https://doi.org/10.1109/HPCA.2004.10030.

[108] Linus Torvalds, ed. *NUMA binding description*. 2016. URL: https://www.kernel.org/doc/Documentation/devicetree/bindings/numa.txt (visited on 07/08/2020).

[109] Pablo de Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov and William Jalby. 'CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization'. In: *ACM Trans. Archit. Code Optim.* 12.1 (Apr. 2015). ISSN: 1544-3566. https://doi.org/10.1145/2724717.

[110] Pablo de Oliveira Castro, Yuriy Kashnikov, Chadi Akel, Mihail Popov and William Jalby. 'Fine-Grained Benchmark Subsetting for System Selection'. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '14. Orlando, Florida, USA: ACM, 2014, pp. 132–142. ISBN: 978-1-4503-2670-4. https://doi.org/10.1145/2581122.2544144.

[111] Rabab al-Omairy, Guillermo Miranda, Hatem Ltaief, Rosa M. Badia, Xavier Martorell, Jesús Labarta and David Keyes. 'Dense Matrix Computations on NUMA Architectures with Distance-Aware Work Stealing'. In: *Supercomput. Front. Innov.* 2.1 (Jan. 2015), pp. 49–72. ISSN: 2313-8734. https://doi.org/10.14529/jsfi150103.

[112] *OpenBLAS Library*. 2016. URL: http://www.openblas.net/.

[113] OpenMP Committee. *OpenMP 4.0 Complete Specifications*. OpenMP Committee Technical Report. OpenMP Architecture Review Board, July 2013. URL: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[114] Cristobal Ortega, Lluc Alvarez, Marc Casas, Ramon Bertran, Alper Buyuktosunoglu, Alexandre E. Eichenberger, Pradip Bose and Miquel Moretó. 'Intelligent Adaptation of Hardware Knobs for Improving Performance and Power Consumption'. In: *IEEE Transactions on Computers* 70.1 (2021), pp. 1–16. https://doi.org/10.1109/TC.2020.2980230.

[115] Cristobal Ortega, Miquel Moretó, Marc Casas, Ramon Bertran, Alper Buyuktosunoglu, Alexandre E. Eichenberger and Pradip Bose. 'libPRISM: An Intelligent Adaptation of Prefetch and SMT Levels'. In: *Proceedings of the International Conference on Supercomputing*. ICS '17 (Chicago, IL, US, June 2017). New York, NY, US: Association for Computing Machinery, 2017. isbn: 978-1-4503-5020-4. https://doi.org/10.1145/3079079.3079101.

[116] Vassilis Papaefstathiou, Manolis G.H. Katevenis, Dimitrios S. Nikolopoulos and Dionisios Pnevmatikatos. 'Prefetching and Cache Management Using Task Lifetimes'. In: *Proceedings of the 2013 ACM International Conference on Supercomputing*. ICS '13 (Eugene, OR, US, June 2013). New York, NY, US: Association for Computing Machinery, 2013, pp. 325–334. isbn: 978-1-4503-2130-3. https://doi.org/10.1145/2464996.2465443.

[117] Jean-Charles Papin, Christophe Denoual, Laurent Colombet and Raymond Namyst. 'SPAWN: An Iterative, Potentials-Based, Dynamic Scheduling and Partitioning Tool'. In: *SC '15 - RESPA Workshop*. Nov. 2015. url: https://hal.inria.fr/hal-01223897.

[118] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot and Édouard Duchesnay. 'Scikit-learn: Machine Learning in Python'. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. arXiv: 1201.0490 [cs.LG].

[119] Leeor Peled, Uri Weiser and Yoav Etsion. 'A Neural Network Prefetcher for Arbitrary Memory Access Patterns'. In: *ACM Transactions on Architecture and Code Optimization* 16.4 (Oct. 2019), 37:1–37:27. ISSN: 1544-3566. https://doi.org/10.1145/3345000.

[120] François Pellegrini. 'Static Mapping by Dual Recursive Bipartitioning of Process Architecture Graphs'. In: *Scalable High Performance Computing Conference*. SHPCC 1994. IEEE, May 1994, pp. 486–493. ISBN: 978-0-8186-5680-4. https://doi.org/10.1109/SHPCC.1994.296682.

[121] François Pellegrini. *SCOTCH*. 2012. URL: https://www.labri.fr/perso/pelegrin/scotch/.

[122] François Pellegrini. *Scotch and libScotch 6.0 User's Guide*. 2014. URL: http://gforge.inria.fr/docman/view.php/248/8260/scotch_user6.0.pdf.

[123] Mihail Popov, Chadi Akel, Yohan Chatelain, William Jalby and Pablo de Oliveira Castro. 'Piecewise holistic autotuning of parallel programs with CERE'. In: *Concurrency and Computation: Practice and Experience* 29.15 (2017). https://doi.org/10.1002/cpe.4190.

[124] Mihail Popov, Chadi Akel, Florent Conti, William Jalby and Pablo de Oliveira Castro. 'PCERE: Fine-Grained Parallel Benchmark Decomposition for Scalability Prediction'. In: *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*. IPDPS '15. Hyderabad, India: IEEE, 2015, pp. 1151–1160. ISBN: 978-1-4799-8649-1. https://doi.org/10.1109/IPDPS.2015.19.

[125] Mihail Popov, Alexandra Jimborean and David Black-Schaffer. 'Efficient Thread/Page/Parallelism Autotuning for NUMA Systems'. In: *Proceedings of the ACM International Conference on Supercomputing*. ICS '19 (Phoenix, Arizona, US, June 2019). New York, NY, US: Association for Computing Machinery, 2019, pp. 342–353. ISBN: 978-1-4503-6079-1. https://doi.org/10.1145/3330345.3330376.

[126] Fatih Porikli. 'Integral Histogram: A Fast Way to Extract Histograms in Cartesian Spaces'. In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Vol. 1. CVPR 2005. IEEE, 2005,

pp. 829–836. ISBN: 978-0-7695-2372-9. https://doi.org/10.1109/CVPR.2005.188.

[127] Maria PREDARI and Aurélien ESNARD. 'A k-Way Greedy Graph Partitioning with Initial Fixed Vertices for Parallel Applications'. In: *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. PDP 2016. IEEE, Feb. 2016, pp. 280–287. https://doi.org/10.1109/PDP.2016.109.

[128] Petar RADOJKOVIĆ, Vladimir ČAKAREVIĆ, Miquel MORETÓ, Javier VERDÚ, Alex PAJUELO, Francisco J. CAZORLA, Mario NEMIROVSKY and Mateo VALERO. 'Optimal Task Assignment in Multithreaded Processors: A Statistical Approach'. In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVII (London, England, UK, Mar. 2012). New York, NY, US: Association for Computing Machinery, 2012, pp. 235–248. ISBN: 978-1-4503-0759-8. https://doi.org/10.1145/2150976.2151002.

[129] Petar RADOJKOVIĆ, Paul M. CARPENTER, Miquel MORETÓ, Vladimir ČAKAREVIĆ, Javier VERDÚ, Alex PAJUELO, Francisco J. CAZORLA, Mario NEMIROVSKY and Mateo VALERO. 'Thread Assignment in Multicore/Multithreaded Processors: A Statistical Approach'. In: *IEEE Transactions on Computers* 65.1 (Jan. 2016), pp. 256–269. ISSN: 0018-9340. https://doi.org/10.1109/TC.2015.2417533.

[130] Edwin D. REILLY, Anthony RALSTON and David HEMMENDINGER, eds. *Encyclopedia of Computer Science*. 4th ed. Chichester, UK: John Wiley and Sons, Ltd., 2003. ISBN: 978-0-470-86412-8.

[131] James REINDERS. *Intel Threading Building Blocks*. 1st ed. Sebastopol, CA, US: O'Reilly Associates, Inc., 2007. ISBN: 978-0-596-51480-8.

[132] Alejandro RICO, Felipe CABARCAS, Carlos VILLAVIEJA, Milan PAVLOVIC, Augusto VEGA, Yoav ETSION, Alex RAMIREZ and Mateo VALERO. 'On the Simulation of Large-Scale Architectures Using Multiple Application Abstraction Levels'. In: *ACM Transactions on Architecture and Code Optimization* 8.4 (Jan. 2012), 36:1–36:20. ISSN: 1544-3566. https://doi.org/10.1145/2086696.2086715.

[133] Alejandro Rico, Isaac Sánchez Barrera, Jose A. Joao, Joshua Randall, Marc Casas and Miquel Moretó. 'On the Benefits of Tasking with OpenMP'. In: *OpenMP: Conquering the Full Hardware Spectrum*. 11th International Workshop on OpenMP. IWOMP 2019 (Auckland, New Zealand, Sept. 2019). Ed. by Xing Fan, Bronis R. de Supinski, Oliver Sinnen and Nasser Giacaman. Lecture Notes in Computer Science 11718. Cham, CH: Springer International Publishing, 2019, pp. 217–230. ISBN: 978-3-030-28596-8. https://doi.org/10.1007/978-3-030-28596-8_15.

[134] Isaac Sánchez Barrera, David Black-Schaffer, Marc Casas, Miquel Moretó, Anastasiia Stupnikova and Mihail Popov. 'Modeling and Optimizing NUMA Effects and Prefetching with Machine Learning'. In: *Proceedings of the 34th ACM International Conference on Supercomputing*. ICS '20. New York, NY, USA: Association for Computing Machinery, June 2020, pp. 1–13. ISBN: 978-1-4503-7983-0. https://doi.org/10.1145/3392717.3392765.

[135] Isaac Sánchez Barrera, Marc Casas and Miquel Moretó. 'NUMA-Aware Hardware Prefetching'. 2021. Submitted.

[136] Isaac Sánchez Barrera, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, Mateo Valero and Marc Casas. 'Reducing Data Movement on Large Shared Memory Systems by Exploiting Computation Dependencies'. In: *Proceedings of the 2018 International Conference on Supercomputing*. ICS '18 (Beijing, CN, June 2018). New York, NY, US: Association for Computing Machinery, 2018, pp. 207–217. ISBN: 978-1-4503-5783-8. https://doi.org/10.1145/3205289.3205310.

[137] Curt Schimmel. *Cache Coherency*. In: *Encyclopedia of Computer Science*. Ed. by Edwin D. Reilly, Anthony Ralston and David Hemmendinger. 4th ed. Chichester, UK: John Wiley and Sons, Ltd., 2003, pp. 176–180. ISBN: 978-0-470-86412-8.

[138] *SGI® Altix® UV 100 System User's Guide*. Version 001. Silicon Graphics International Corp. 2010. xviii + 88. URL: https://irix7.com/techpubs/007-5662-001.pdf (visited on 23/04/2021).

[139] Daniel L. Slotnick. 'The Conception and Development of Parallel Processors: A Personal Memoir'. In: *Annals of the History of Computing* 4.1 (1982), pp. 20–30. https://doi.org/10.1109/MAHC.1982.10003.

[140] Alan Jay Smith. 'Cache Memories'. In: *ACM Computing Surveys* 14.3 (Sept. 1982), pp. 473–530. issn: 0360-0300. https://doi.org/10.1145/356887.356892.

[141] *Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. Standard, IEEE 1003.1c-1995.

[142] Xubin Tan, Jaume Bosch, Daniel Jiménez-González, Carlos Álvarez-Martínez, Eduard Ayguadé and Mateo Valero. 'Performance analysis of a hardware accelerator of dependence management for task-based dataflow programming models'. In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS 2016 (Uppsala, SE, Apr. 2016). Piscataway, NJ, US: IEEE Press, 2016, pp. 225–234. isbn: 978-1-5090-1953-3. https://doi.org/10.1109/ISPASS.2016.7482097.

[143] Xubin Tan, Jaume Bosch, Miquel Vidal, Carlos Álvarez, Daniel Jiménez-González, Eduard Ayguadé and Mateo Valero. 'General Purpose Task-Dependence Management Hardware for Task-Based Dataflow Programming Models'. In: *2017 IEEE International Parallel and Distributed Processing Symposium*. IPDPS 2017 (Orlando, FL, US, May–June 2017). Piscataway, NJ, US: IEEE Press, 2017, pp. 244–253. isbn: 978-1-5386-3914-6. https://doi.org/10.1109/IPDPS.2017.48.

[144] Masahiro Tanaka and Osamu Tatebe. 'Workflow Scheduling to Minimize Data Movement Using Multi-Constraint Graph Partitioning'. In: *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGrid 2012 (Ottawa, ON, CA, May 2012). Piscataway, NJ, US: IEEE Press, 2012, pp. 65–72. https://doi.org/10.1109/CCGrid.2012.134.

[145] Xavier Teruel, Xavier Martorell, Alejandro Duran, Roger Ferrer and Eduard Ayguadé. 'Support for OpenMP Tasks in Nanos V4'. In: *Conference of the Center for Advanced Studies on Collaborative Research*. CASCON '07. Riverton, NJ, USA: IBM Corp., 2007, pp. 256–259. https://doi.org/10.1145/1321211.1321241.

[146] The MPI Forum. 'MPI: A Message Passing Interface'. In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing '93 (Portland, Oregon, US). New York, NY, US: Association for Computing Machinery, 1993, pp. 878–883. ISBN: 978-0-8186-4340-8. https://doi.org/10.1145/169627.169855.

[147] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 'Hardware Monitors for Dynamic Page Migration'. In: *Journal of Parallel and Distributed Computing* 68.9 (Sept. 2008), pp. 1186–1200. ISSN: 0743-7315. https://doi.org/10.1016/j.jpdc.2008.05.006.

[148] François Trahay, Manuel Selva, Lionel Morel and Kevin Marquet. 'NumaMMA: NUMA MeMory Analyzer'. In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP 2018 (Eugene, OR, US, Aug. 2018). New York, NY, US: Association for Computing Machinery, 2018. ISBN: 978-1-4503-6510-9. https://doi.org/10.1145/3225058.3225094.

[149] Jan Treibig, Georg Hager and Gerhard Wellein. 'LIKWID: A Lightweight Performance-Oriented Tool Suite for X86 Multicore Environments'. In: *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*. ICPPW '10. San Diego, California, USA: IEEE, 2010, pp. 207–216. ISBN: 978-0-7695-4157-0. https://doi.org/10.1109/ICPPW.2010.38.

[150] Mateo Valero, Miquel Moretó, Marc Casas, Eduard Ayguadé and Jesús Labarta. 'Runtime-Aware Architectures: A First Approach'. In: *Supercomputing Frontiers and Innovations* 1.1 (Sept. 2014), pp. 28–43. ISSN: 2313-8734. https://doi.org/10.14529/jsfi140102.

[151] Raul Vidal, Marc Casas, Miquel Moretó, Dimitrios Chasapis, Roger Ferrer, Xavier Martorell, Eduard Ayguadé, Jesús Labarta and Mateo Valero. 'Evaluating the Impact of OpenMP 4.0 Extensions on

Relevant Parallel Workloads'. In: *OpenMP: Heterogenous Execution and Data Movements. International Workshop on OpenMP*. Lecture Notes in Computer Science. Cham: Springer, Oct. 2015, pp. 60–72. ISBN: 978-3-319-24594-2 978-3-319-24595-9. https://doi.org/10.1007/978-3-319-24595-9_5.

[152] Philippe VIROULEAU, François BROQUEDIS, Thierry GAUTIER and Fabrice RASTELLO. 'Using Data Dependencies to Improve Task-Based Scheduling Strategies on NUMA Architectures'. In: *Euro-Par 2016: Parallel Processing*. 22nd International Conference on Parallel and Distributed Computing. Euro-Par 2016 (Grenoble, FR, Aug. 2016). Lecture Notes in Computer Science 9833. Cham, CH: Springer International Publishing, Aug. 2016, pp. 531–544. ISBN: 978-3-319-43659-3. https://doi.org/10.1007/978-3-319-43659-3_39.

[153] Krishnaswamy VISWANATHAN. *Disclosure of Hardware Prefetcher Control on Some Intel® Processors*. Intel Corp. 24th Sept. 2014. URL: https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html.

[154] Qian WANG, Xianyi ZHANG, Yunquan ZHANG and Qing YI. 'AUGEM: Automatically Generate High Performance Dense Linear Algebra Kernels on X86 CPUs'. In: *International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. New York, NY, USA: ACM, 2013, 25:1–25:12. ISBN: 978-1-4503-2378-9. https://doi.org/10.1145/2503210.2503219.

[155] Wei WANG, Jack W. DAVIDSON and Mary Lou SOFFA. 'Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale NUMA machines'. In: *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture*. HPCA '16. Barcelona, Spain: IEEE, 2016, pp. 419–431. ISBN: 978-1-4673-9211-2. https://doi.org/10.1109/HPCA.2016.7446083.

[156] Zheng WANG and Michael O'BOYLE. 'Machine Learning in Compiler Optimization'. In: *Proceedings of the IEEE* 106.11 (2018), pp. 1879–

1901. ISSN: 0018-9219. https://doi.org/10.1109/JPROC.2018.2817118.

[157] Zheng WANG and Michael F.P. O'BOYLE. 'Mapping Parallelism to Multi-Cores: A Machine Learning Based Approach'. In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '09. Raleigh, North Carolina, USA: ACM, 2009, pp. 75–84. ISBN: 978-1-60558-397-6. https://doi.org/10.1145/1504176.1504189.

[158] Maurice V. WILKES. 'Slave Memories and Dynamic Storage Allocation'. In: *IEEE Transactions on Electronic Computers*. EC 14.2 (Apr. 1965), pp. 270–271. ISSN: 0367-7508. https://doi.org/10.1109/PGEC.1965.264263.

[159] Carole-Jean WU and Margaret MARTONOSI. 'Characterization and Dynamic Mitigation of Intra-Application Cache Interference'. In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. ISPASS '11. Austin, Texas, USA: IEEE, 2011, pp. 2–11. ISBN: 978-1-61284-367-4. https://doi.org/10.1109/ISPASS.2011.5762710.

[160] William A. WULF and Sally A. MCKEE. 'Hitting the Memory Wall: Implications of the Obvious'. In: *SIGARCH Computer Architecture News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964. https://doi.org/10.1145/216585.216588.

[161] Sam Likun XI, Hans JACOBSON, Pradip BOSE, Gu-Yeon WEI and David BROOKS. 'Quantifying Sources of Error in McPAT and Potential Impacts on Architectural Studies'. In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2015, pp. 577–589. https://doi.org/10.1109/HPCA.2015.7056064.

[162] Yonghong YAN, Jisheng ZHAO, Yi GUO and Vivek SARKAR. 'Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement'. In: *Languages and Compilers for Parallel Computing*. Berlin, Heidelberg: Springer, Oct. 2009, pp. 172–187. ISBN: 978-3-642-13373-2 978-3-642-13374-9. https://doi.org/10.1007/978-3-642-13374-9_12.

[163] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish and Srinivas Devadas. 'IMP: Indirect Memory Prefetcher'. In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. New York, NY, USA: Association for Computing Machinery, Dec. 2015, pp. 178–190. isbn: 978-1-4503-4034-2. https://doi.org/10.1145/2830772.2830807.