# Architectural Support for Software-Guided Energy Reduction of Manycore Communication

Vassilis D. Papaefstathiou

December 2013

University of Crete
School of Sciences and Engineering
Computer Science Department

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy

**Doctoral Dissertation Committee:**
Manolis G.H. Katevenis (Advisor)
Dimitrios S. Nikolopoulos (Co-Advisor)
Angelos Bilas
Panagiota Fatourou
Stefanos Kaxiras
Dionisios N. Pnevmatikatos
Per Stenström

This work was performed at the Computer Architecture and VLSI Systems (CARV) Laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH), and was financially supported by a FORTH-ICS scholarship, including funding by the European Union 7th Framework Programme [FP7/2007-2013], under the ENCORE (FP7-ICT-248647), TeXT (FP7-ICT-261580), and HiPEAC-3 (FP7-ICT-287759) projects.

University of Crete
Department of Computer Science

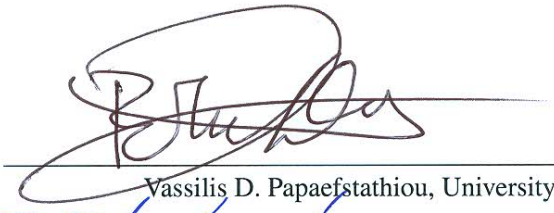
**Architectural Support for Software-Guided Energy Reduction of Manycore Communication**

Dissertation submitted by

**Vassilis D. Papaefstathiou**

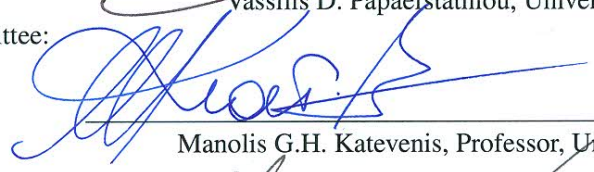
in partial fulfillment of the requirements for
the Ph.D. degree in Computer Science.

Author:

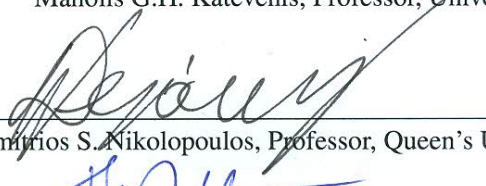


Vassilis D. Papaefstathiou, University of Crete

Committee:



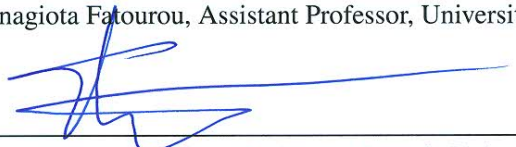
Manolis G.H. Katevenis, Professor, University of Crete



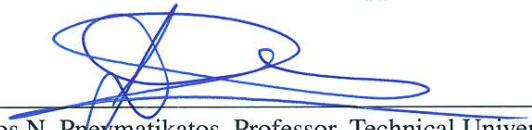
Dimitrios S. Nikolopoulos, Professor, Queen's University of Belfast

Angelos Bilas, Professor, University of Crete

Panagiota Fatourou, Assistant Professor, University of Crete



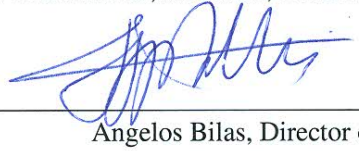
Stefanos Kaxiras, Professor, Uppsala University



Dionisios N. Pnevmatikatos, Professor, Technical University of Crete

Per Stenström, Professor, Chalmers University of Technology

Approved by:



Angelos Bilas, Director of Graduate Studies
Heraklion, December 2013

Αφιερωμένο στους γονείς μου και
στις ιερές σκιές των παππούδων και γιαγιάδων μου

*Dedicated to my parents and*
*the holy shadows of my grandparents*

# Architectural Support for Software-Guided Energy Reduction of Manycore Communication

Vassilis D. Papaefstathiou

Advisors: Manolis G.H. Katevenis and Dimitrios S. Nikolopoulos

## *Abstract*

At the beginning of the 21st century, the processor industry made a fundamental shift towards multicore architectures, in order to address the diminishing returns in single-thread performance with increasing transistor counts, and in order to overcome the severe power problems of clock frequency scaling. Semiconductor technology trends indicate that now the era of power- and energy-constrained manycore architectures has come. Technology projections show that the energy consumed by data movement and communication will dominate the corresponding budget of future computing systems; thus, unnecessary data movements will subtract significant energy margin from computations.

The most popular communication model for multi-core and many-core architectures is shared-memory. Threads or processes that run concurrently on different cores communicate and exchange data by accessing the same global memory locations. However, accesses to off-chip memory are slow and, thus, processor designs utilize a hierarchy of faster on-chip memories to improve the speed of memory operations. Memory hierarchies today are based on two dominant schemes: *(i)* multi-level coherent caches, and *(ii)* software-managed local memories (scratchpads). Caches manage the memory hierarchy transparently, using hardware replacement policies, and communication happens implicitly, with cache-coherence protocols that provoke data transfers between caches. Scratchpad memories are controlled by the programmer or the runtime software, and communication happens explicitly, through programmable DMA engines that perform the data transfers.

This thesis proposes architectural support in the memory hierarchy to enable the software to control data locality; we design programmable hardware primitives that allow runtime software to orchestrate communication and reduce the associated energy consumption.

We demonstrate a hybrid cache/scratchpad memory hierarchy that provides unified hardware support for both implicit communication, via cache-coherence, and explicit communication, via fast virtualized inter-processor communication hardware primitives. We also introduce the Epoch-based Cache Management (ECM), which allows software to assign priorities to cache-lines, in order to guide the cache replacement policy, and, in effect, to manage locality. Moreover, we design the Explicit Bulk Prefetcher (EBP), a programmable prefetch engine that allows software to accurately prefetch data ahead of time, in order to hide memory

latency and improve cache locality. Furthermore, we propose a set of hardware primitives for Software Guided Coherence (SGC) in non-cache-coherent systems, in order to allow runtime software to orchestrate the fetching of the most up-to-date version of data from the appropriate cache(s) and maintain coherence at the software object granularity.

We evaluate our proposed hardware primitives by comparing them against directory-based cache-coherence with hardware prefetching. Our experimental results for explicit communication show that we can improve performance by 10% to 40%, and at the same time reduce the energy consumption of on-chip communication by 35% to 70% owing to significant reduction in on-chip traffic, by factors of 2 to 4. Moreover, we exploit a task-based programming system to guide hardware, and show that our proposed hardware primitives in cache-coherent systems (ECM, EBP) improve performance by an average of 20%, inject 25% less on-chip traffic on average, and reduce the energy consumption in the components of the memory hierarchy by an average of 28%. Our hardware support for non-cache-coherent systems (ECM, SGC) improves performance by an average of 14%, injects 41% less on-chip traffic on average, and reduces the energy consumption in the components of the memory hierarchy by an average of 44%.

Αρχιτεκτονική Υποστήριξη για Μείωση Κατανάλωσης
Ενέργειας στην Επικοινωνία Πολυπύρηνων Επεξεργαστών
υπό την Καθοδήγηση Λογισμικού

Βασίλειος Δ. Παπαευσταθίου

Επόπτες: Μανόλης Γ.Η. Κατεβαίνης και Δημήτριος Σ. Νικολόπουλος

# Περίληψη

Στις αρχές του $21^{ου}$ αιώνα, η βιομηχανία των επεξεργαστών έκανε μια θεμελιώδη στροφή προς αρχιτεκτονικές πολλαπλών πυρήνων, για να αντιμετωπίσει την φθίνουσα απόδοση των επιπλέον τρανζίστορ στην επίδοση των μονοεπεξεργαστών, και να ξεπεράσει τα προβλήματα κατανάλωσης ισχύος λόγω της αύξησης των συχνοτήτων ρολογιού. Οι τάσεις της βιομηχανίας των ημιαγωγών δείχνουν ότι έχει πλέον έρθει η εποχή των ενεργειακά περιορισμένων πολυπύρηνων επεξεργαστών. Τεχνολογικές προβλέψεις δείχνουν ότι η ενέργεια που καταναλώνεται για κίνηση δεδομένων και επικοινωνία θα κυριαρχήσει στον ενεργειακό προϋπολογισμό των μελλοντικών υπολογιστικών συστημάτων, επομένως, κάθε περιττή μετακίνηση δεδομένων θα μειώνει την διαθέσιμη ενέργεια για υπολογισμούς.

Το πιο δημοφιλές μοντέλο επικοινωνίας για πολυπύρηνες αρχιτεκτονικές είναι η κοινόχρηστη μνήμη. Οι διεργασίες και τα επεξεργαστικά νήματα που εκτελούνται ταυτόχρονα σε διαφορετικούς πυρήνες επικοινωνούν και ανταλλάσσουν δεδομένα προσπελαύνοντας τις ίδιες καθολικές τοποθεσίες μνήμης. Ωστόσο, οι προσβάσεις σε μνήμη εκτός του τσιπ είναι αργές και ως εκ τούτου, οι επεξεργαστές χρησιμοποιούν μια ιεραρχία από ταχύτερες μνήμες εντός του τσιπ για να βελτιώσουν την ταχύτητα των προσπελάσεων μνήμης. Οι ιεραρχίες μνήμης σήμερα βασίζονται σε δύο κυρίαρχα σχέδια: (α) πολυ-επίπεδες κρυφές μνήμες με πρωτόκολλα συνοχής, και (β) τοπικές μνήμες διαχειριζόμενες από το λογισμικό. Οι κρυφές μνήμες διαχειρίζονται την ιεραρχία μνήμης διαφανώς, το υλικό εφαρμόζει πολιτικές αντικατάστασης, και η επικοινωνία γίνεται εμμέσως, μέσω των πρωτοκόλλων συνοχής που χειρίζονται τη μεταφορά δεδομένων μεταξύ των κρυφών μνημών. Οι τοπικές μνήμες ελέγχονται από τον προγραμματιστή ή το λογισμικό εκτέλεσης και η επικοινωνία γίνεται ρητώς, μέσω προγραμματιζόμενων μηχανών άμεσης προσπέλασης μνήμης που εκτελούν τις μεταφορές δεδομένων.

Η διατριβή αυτή προτείνει αρχιτεκτονική υποστήριξη στην ιεραρχία μνήμης για να επιτρέψει στο λογισμικό να διαχειριστεί την τοπικότητα των δεδομένων που χρησιμοποιεί. Σχεδιάζουμε προγραμματιζόμενα στοιχεία υλικού που επιτρέπουν στο λογισμικό εκτέλεσης να ενορχηστρώσει την επικοινωνία και να μειώσει τη σχετιζόμενη κατανάλωση ενέργειας.

Παρουσιάζουμε μια υβριδική ιεραρχία μνήμης που λειτουργεί ταυτοχρόνως σαν κρυφή μνήμη, και σαν τοπική μνήμη διαχειριζόμενη από το λογισμικό. Παρέχουμε ενιαία υποστήριξη στο υλικό τόσο για έμμεση επικοινωνία, μέσω πρωτοκόλλων συνοχής, όσο και για ρητή επικοινωνία μέσω γρήγορων στοιχείων υλικού που υποστηρίζουν εικονικοποίηση. Εισάγουμε επίσης την διαχείριση κρυφής μνήμης μέσω 'εποχών', που επιτρέπουν στο λογισμικό να αναθέσει προτεραιότητες στα δεδομένα της κρυφής μνήμης, να καθοδηγήσει την πολιτική αντικατάστασης, και ουσιαστικά να διαχειριστεί την τοπικότητα των δεδομένων. Επιπλέον, σχεδιάζουμε μια προγραμματιζόμενη μηχανή ρητής προσκόμισης που επιτρέπει στο λογισμικό να μεταφέρει εγκαίρως, και με ακρίβεια, τα δεδομένα που θα χρειαστεί, με σκοπό την μείωση των καθυστερήσεων στις προσπελάσεις μνήμης, και την βελτίωση της τοπικότητας στην κρυφή μνήμη. Επιπροσθέτως, προτείνουμε στοιχεία υλικού που επιτρέπουν στο λογισμικό να διαχειριστεί την συνοχή των κρυφών μνημών σε συστήματα που δεν υλοποιούν πρωτόκολλα συνοχής, έτσι ώστε να επιτρέψουμε στο λογισμικό εκτέλεσης να ενορχηστρώσει την μεταφορά της πιο πρόσφατης έκδοσης των δεδομένων από τις κατάλληλες κρυφές μνήμες και να διατηρήσει την συνοχή στο επίπεδο των αντικειμένων του λογισμικού.

Αξιολογούμε τα προτεινόμενα στοιχεία υλικού συγκρίνοντάς τα με υλικό που υλοποιεί πρωτόκολλα συνοχής κρυφών μνημών βασιζόμενα σε καταλόγους, και περιλαμβάνει μηχανές αυτόματης προσκόμισης. Τα πειραματικά μας αποτελέσματα για ρητή επικοινωνία δείχνουν ότι μπορούμε να βελτιώσουμε την επίδοση κατά 10% έως 40%, και ταυτόχρονα να μειώσουμε την κατανάλωση ενέργειας στην επικοινωνία εντός του τσιπ κατά 35% έως 70% λόγω της σημαντικής μείωσης στην κυκλοφορία δεδομένων εντός του τσιπ, κατά παράγοντες 2 έως 4. Επιπλέον, χρησιμοποιούμε ένα σύστημα προγραμματισμού που βασίζεται σε έργα για να καθοδηγήσει το υλικό, και δείχνουμε ότι τα προτεινόμενα στοιχεία υλικού σε συστήματα με πρωτόκολλα συνοχής κρυφών μνημών βελτιώνουν την επίδοση κατά μέσο όρο 20%, μειώνουν την κυκλοφορία εντός του τσιπ κατά μέσο όρο 25%, και μειώνουν την κατανάλωση ενέργειας στην ιεραρχία μνήμης κατά μέσο όρο 28%. Σε συστήματα που δεν υλοποιούν πρωτόκολλα συνοχής κρυφών μνημών, η αρχιτεκτονική υποστήριξη που προτείνουμε βελτιώνει την επίδοση κατά μέσο όρο 14%, μειώνει την κυκλοφορία εντός του τσιπ κατά μέσο όρο 41%, και μειώνει την κατανάλωση ενέργειας στην ιεραρχία μνήμης κατά μέσο όρο 44%.

# Acknowledgments

This thesis was performed at the Computer Architecture and VLSI Systems Laboratory (CARV) of the Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH), and was financially supported by a FORTH-ICS scholarship, including funding by the European Union 7$^{th}$ Framework Programme [FP7/2007-2013], under the ENCORE (FP7-ICT-248647), TeXT (FP7-ICT-261580), and HiPEAC-3 (FP7-ICT-287759) projects. I deeply thank FORTH-ICS for being my "academic home" during the last *ten* years (since quite earlier than the beginning of my PhD studies), offering me the opportunity to work on numerous challenging research topics, and collaborate with bright researchers.

I am especially grateful to my advisor Prof. Manolis Katevenis for his thorough guidance and strong support all these years; his advices have been tremendously influential in my thinking and writing. I am also grateful to my co-advisor Prof. Dimitris Nikolopoulos for offering valuable advice and pointing to very interesting research directions. I would also like to thank Prof. Dionisios Pnevmatikatos and Prof. Angelos Bilas for always asking the "right" and thought-provoking questions. I also thank Prof. Panagiota Fatourou, Prof. Stefanos Kaxiras, and Prof. Per Stenström for their feedback and comments on this thesis.

I would like to thank all the members of the CARV Laboratory, and especially the following persons who deserve a honorable mention. Foremost, I thank my friend George Kalokerinos for our wonderful collaboration, his unique way of threatening to throw curses on me has been extremely motivational! Spyros Lyberis, Stamatis Kavadias, and George Nikiforos have been ideal colleagues, we did great things together, we had numerous brainstorming sessions, and they really helped me move forward this work. Manolis Marazakis and Michael Ligerakis were always willing to discuss and offer help. Polyvios Pratikakis and Foivos Zakkak provided valuable support in compiler issues.

I owe gratitude to all my friends; although many have spread across several cities and countries, our friendship remains strong. I would like to thank, in random order: Chariton, Dimitris (Golden-boy), Dimitris (Darth), Takis, Panos (Giogakis), Manos, Vassilis, Lilly, Themis, Panos (Castillo), Kostas, Dimitris (Bakdim), Mimis, Nikos, Kostas (Italos), Giorgos – each one of you knows why. Special thanks to Evangelia for standing by me and supporting me all these years.

Last but not least, I would like to thank my brother Meletis, my sister Angeliki, and my parents Dimitris and Vassiliki for their love and support. All I have accomplished I owe it to my parents and I dedicate this work to them.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

During the last decades of the 20$^{th}$ century, performance improvement in computing systems was driven to a considerable extend by advances in semiconductor technology and *Moore's Law* [1], which predicts that the number of transistors per chip doubles every 18 months. Additionally, *Dennard Scaling* [2] allowed the power density of chips to remain nearly constant across process generations owing to smaller transistors, improved transistor speed, and supply voltage scaling. Computer architects utilized the increasing transistor counts and came up with innovative architectural designs that offered exponential scaling in single-core processor performance. Superscalar processors with out-of-order and speculative execution and with large caches exploited instruction-level parallelism (ILP), and deep pipelines permitted significantly higher clock frequencies.

At the beginning of the 21$^{st}$ century, the processor designers were confronted with diminishing returns from ILP mining, while higher clock frequencies encountered serious power consumption limits. At that point, the industry made a fundamental shift towards processors with multiple cores and fixed clock frequencies, in order to continue improving the aggregate chip performance while staying within the power envelope. Figure 1.1 illustrates historical microprocessor scaling trends and clearly shows the inflection point where processors with multiple cores ap-

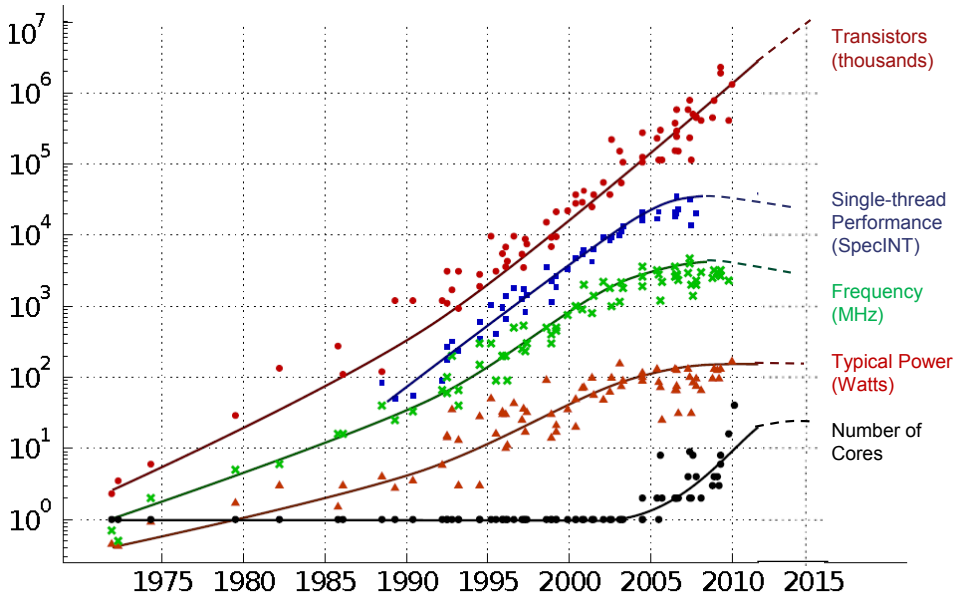



Figure 1.1: Microprocessor scaling trends of the last 35 years. *Chuck Moore, AMD [3] - Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten - Dotted line extrapolations by C. Moore.*

peared when designs hit the ILP and power wall.

The advent of multicore processors and the upcoming manycore era pose significant challenges for the computing industry. The hardware industry is facing serious power constraints in chip designs, and the software industry is looking after a solution to the "parallel programming crisis".

The exploitation of processors with multiple cores requires application developers to write parallel code and delve into the details of the underlying architecture. Parallel programming is traditionally considered hard and the programmers need to put significant effort in order to write efficient programs and debug complex concurrency bugs. Achieving performance that scales with the number of cores requires tangling with tedious and time-consuming tasks such as scheduling, synchronization, and locality optimizations. The research community and software engineering industry are currently in a race to find appropriate programming models and abstractions to ease programmability, and improve performance efficiency.

For the semiconductor industry, the ITRS technology roadmap [4] indicates the end of Dennard scaling, since transistor speed and voltage supply do not scale proportionally to dimensions anymore. Although transistor density per chip will continue to double in successive process generations (following Moore's Law), the

energy efficiency of transistors will not follow the same rate, thus every future processor chip will be power-limited [5]. Recent studies also forecast that due to power constraints, future processor chips will not be able to operate all existing cores simultaneously and will require a large fraction of the cores to be switched off [6]. The *dark silicon* in future chips may be as much as 50% [7, 8].

## 1.1 Motivation: Cost of Communication

Processor architectures will change dramatically in the next decade as, the power constraints impose hard limits. Future chips will further increase the on-chip parallelism and offer hundreds of cores, while applications and algorithms need to adapt and harness the computing resources as architectures evolve. The scientific community is very active and calls for innovations and major breakthroughs to overcome the new technological obstacles [9]. Research organizations have recently published several reports that identify new challenges, predict technological trends, and provide research directions [10, 11]. There is general consensus that future computing performance growth faces a new reality [12]:

> "*Energy efficiency is the new fundamental limiter of processor performance, way beyond numbers of processors"*

A recent study from the Exascale Computing Initiative of US Department of Energy (DoE) indicates that the energy consumed by data movement and communication will dominate the energy budget of future computing systems [10, 13]. Figure 1.2 presents projections on the energy cost of communication relative to the cost of FLOP for 2010 and 2018 systems. The largest increase in energy consumption occurs when data move off-chip, and the associated cost is an order of magnitude higher than the cost of a FLOP. Moreover, the cost of moving data on-chip is commensurate to the distance traveled. The projections for the 2018-2020 technology show that the cost of FLOP will scale reasonably well and advances in memory technologies and interfaces [14] can reduce the energy cost of off-chip memory accesses. On the contrary, the cost of on-chip communication and data movement is not improving at similar rates in future technologies, when counting these costs for a fixed physical distance (e.g. 5 millimeters). Industrial research groups also confirm the latter trends. A paper recently published by Intel's Exascale Technology Group [15] indicates that signaling power is expected to scale much worse than logic power, causing on-chip and off-chip communication to be-

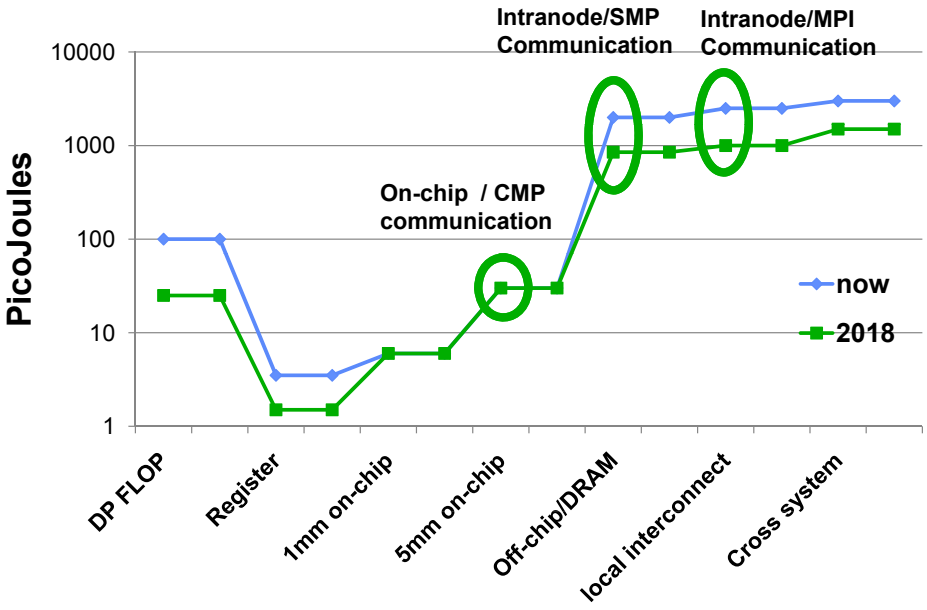



Figure 1.2: Projections on the energy cost of communication relative to the cost of floating-point operation (FLOP) for 2010 and 2018 systems. *John Shalf et al., NERSC DoE [13] - ©Springer-Verlag 2010.*

come a larger fraction of overall power. Nvidia's architecture research group also states that fetching operands costs more energy than computing on them [5, 16]. It becomes apparent that the cost of communication is so pronounced, that unnecessary data movements subtract significant energy budget from computations.

To address this challenge requires achieving two primary goals: *(a)* minimize data movement by exploiting on-chip locality, and *(b)* removing unnecessary overhead from communication. The future architectures can no longer sustain the luxury of software-invisible innovation [11]. The computer architecture community needs to rethink the computing stack and co-design architectures, programming environments, and applications, with energy efficiency in mind. The application algorithms should perform more work per unit of data movement, the programming environments should exploit locality and further optimize data movement, and the architecture should expose hooks for the software to control the memory hierarchy, i.e. provide efficient and low overhead communication primitives.

Memory hierarchies today are based on two dominant schemes: *(a)* multi-level coherent caches [17], and *(b)* software-managed local memories (scratch-pads) [18,19]. Caches manage the memory hierarchy transparently, using hardware replacement policies and communication happens *implicitly*, via cache-coherence

protocols that provoke and control data movements between caches. Scratchpad memories are controlled by the programmer, or the runtime system, and communication happens *explicitly*, through programmable DMA engines that perform the data movements.

Recent studies on parallel programming models and runtimes for software-managed memories [20–22] allow the use of explicit communication with minimal burden for the programmers. These programming models require only the identification of the input and output datasets of parallel tasks. Moreover, evolutions of these programming models allow the underlying runtime systems to track inter-task dependencies and explore task-based dataflow parallelism [23–28].

Architectures with scratchpad memories allow software to reason about locality, control data movement, and hide the memory latency through DMAs. However, such architectures lend themselves better to workloads with known datasets. The use of irregular data, e.g. linked-lists, requires careful data marshaling (scatter-gather lists). On the other hand, current cache designs have a more software friendly best-effort behavior and can support diverse workloads, but they are transparent and optimize only for the common case. Software cannot reason about the presence of specific data in the cache, the underlying data transfers, and communication. Cache replacement policies and hardware cache-coherence protocols manage memory and data movement transparently.

We believe that there is enough design space to explore between the two types of memory architectures and communication styles, so that we can exploit the best of both worlds to improve performance and reduce energy consumption.

## 1.2 This Thesis

This thesis proposes and evaluates architectural support for the power-constrained manycore systems of the future. We focus on the memory hierarchy design, which is where communication occurs. Our work gives answers to the following two key questions:

Q1. How should the memory hierarchy be designed, to offer the flexibility of a cache but still allow software to control locality and data movement?

Q2. What hardware primitives should the architecture provide to minimize the communication overhead and reduce the associated energy consumption?

We answer question 1 by designing a *Hybrid Cache/Scratchpad Memory Hierarchy* where software can opt to manage locality explicitly when datasets are known, or resort to the cache when datasets are hard to manage. To achieve the same goal in cache-based memory hierarchies, we introduce the *Epoch-based Cache Management (ECM)* scheme that allows software to guide cache replacement decisions, hence, control data locality and reason about data movement.

We answer question 2 by designing *Explicit Communication Hardware Primitives* that allow software to direct data movement between scratchpad memories without relying on coherence protocols, thus removing the associated control overhead and reducing energy. To achieve the same effect in cache-based memory hierarchies, we propose hardware primitives for *Software Guided Coherence (SGC)* that allow software to orchestrate data movement, fetch the most up-to-date version of data from the appropriate cache(s), and maintain coherence at the software object granularity.

### Contributions

Our architectural support consists of two major parts and the associated contributions are listed below:

A. We design a hybrid cache/scratchpad memory hierarchy and provide unified hardware support for both implicit and explicit communication within the same address space[1]. This part makes the following contributions:

   A1. We provide run-time configurability of the local memory resources and allow them to operate either as cache or scratchpad memory or a dynamic mix of the two. We also merge the cache controller and the network interface into a unified design to economize on circuits, and provide fast and low-overhead hardware primitives for explicit interprocessor communication.

   A2. We quantify the performance and energy benefits of explicit communication by comparing against directory-based hardware cache coherence with hardware prefetching.

[1] This work was performed jointly with Stamatis Kavadias, George Kalokerinos and George Nikiforos. Vassilis Papaefstathiou and Stamatis Kavadias contributed equally to the definition of the explicit communication hardware primitives. Stamatis Kavadias was the sole contributor of the explicit synchronization primitives. Vassilis Papaefstathiou was the sole contributor of the performance evaluation of the explicit communication primitives presented here. George Kalokerinos and George Nikiforos were major contributors to the development of the FPGA prototype. All collaborators contributed equally to the verification and evaluation effort of the FPGA prototype.

A3. We implement an FPGA prototype of the proposed architecture and measure the hardware cost and latency of the hardware primitives.

A4. We present the paper design of Coherent RDMA to permit copies between scratchpad and cacheable memory regions.

B. We propose hardware/software co-design for cache-based memory hierarchies, assuming a task-based programming system similar to OmpSs [23]. We design hardware primitives to allow runtime software to transfer task arguments and guide the cache replacement policy. Moreover, we provide hardware support to allow software to manage coherence in systems with non-coherent caches. This part makes the following contributions:

B1. We propose the *Explicit Bulk Prefetcher (EBP)*, a programmable prefetch engine that allows software to accurately prefetch data ahead of time and improve cache locality in task-based programs.

B2. We propose *Epoch-based Cache Management (ECM)*, a generic lightweight mechanism to guide cache replacement decisions, assign local cache resources to tasks, and isolate the effects of prefetching.

B3. We propose hardware primitives for *Software Guided Coherence (SGC)* in non-cache-coherent systems, to allow the runtime software to orchestrate data movement, fetch the most up-to-date version of task arguments from the appropriate cache(s), and maintain coherence at task granularity.

B4. We evaluate these hardware primitives (EBP, ECM, SGC) using a task-based runtime and a set of benchmark applications in systems with and without cache-coherence. We measure performance and energy consumption, and compare against directory-based cache-coherence with hardware prefetching.

Parts of work A have been published in the IEEE Micro Magazine (IEEE Micro) 2010 [29], Transactions on High-Performance Embedded Architectures and Compilers (Transactions on HiPEAC) 2010 [30], and IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS) 2009 [31].

Parts of work B have been published and presented in the ACM International Conference on Supercomputing (ICS) 2013 [32]. An additional publication is currently under preparation for submission to a journal.

## 1.3 Outline

The rest of this thesis is organized as follows. Chapter 2 describes the hardware communication mechanisms found in manycore architectures, overviews task-based parallel programming models, and reviews related work. Chapter 3 presents our architectural support that allows software to guide data transfers (communication) through the memory hierarchy and describes the design of related hardware primitives. First, we design a hybrid cache/scratchpad memory hierarchy with explicit communication primitives. Then, we present a hardware/software co-design with task-based programming models for cache-based memory hierarchies, in systems with and without cache-coherence. Chapter 4 describes the experimental methodology used for the evaluation of our proposed architectural support, presents our simulation infrastructure, and describes our design of a minimal task-based runtime system. Chapter 5 presents the performance evaluation of our architectural support and the proposed hardware communication primitives by comparing them against directory-based cache-coherence with hardware prefetching. Chapter 6 summarizes our work, discusses the lessons learned, and presents future work. Appendix A presents the design of an FPGA prototype that implements our hybrid cache/scratchpad memory hierarchy and evaluates hardware cost and performance. Appendix B describes the paper design of Coherent RDMA.

# 2

# Background and Related Work

This chapter presents the basic concepts and relevant work on topics closely related to the field of our research. In Section 2.1 we describe the dominant hardware communication mechanisms found in manycore architectures. In Section 2.2 we make an overview of the most appealing task-based programming models for manycore architectures and focus on the emerging task-based dataflow programming model. Finally, in Section 2.3 we review related work.

## 2.1 Communication in Manycore Architectures

The most popular communication model for multi-core and many-core architectures is shared-memory. Threads or processes that run concurrently on different cores communicate and exchange data by accessing the same global memory locations. However, accesses to off-chip memory are slow and, thus, processor designs utilize a hierarchy of faster on-chip memories to improve the speed of memory accesses. Memory hierarchies today are based on two dominant schemes: *(i)* multi-level coherent caches, and *(ii)* software-managed local memories (scratchpads). Caches manage the memory hierarchy transparently, using hardware replacement policies, and communication happens *implicitly*, with cache-coherence protocols

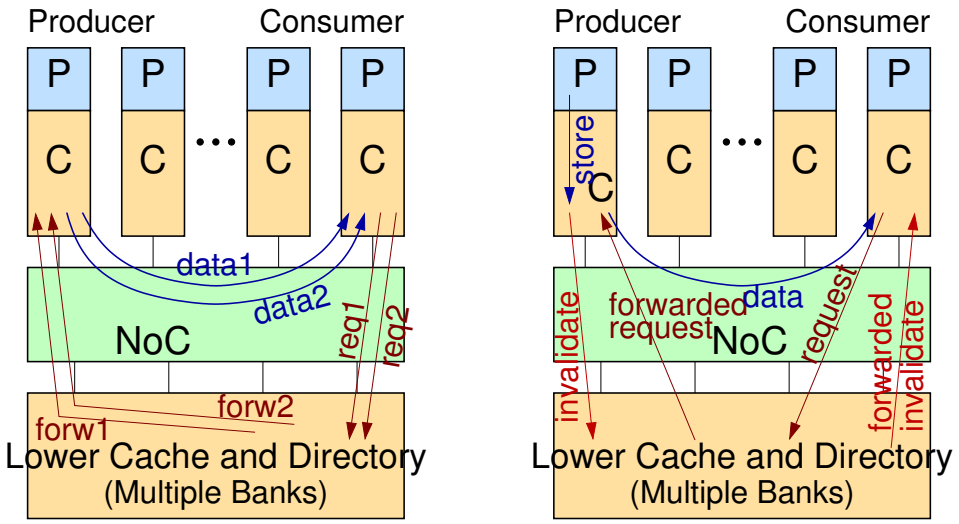



Figure 2.1: Implicit Communication with cache-coherence, pull type (left) push type (right)

that provoke data transfers between caches. Scratchpad memories are controlled by the programmer or the runtime software and communication happens *explicitly*, through programmable DMA engines that perform the data transfers.

The actual communication occurs every time a thread reads a word that has last been modified by another thread. We call producer the modifying thread, and consumer the reading thread. Although the terms producer and consumer have been traditionally associated with stream processing, we use the terms in a true dependence-based definition that is completely general and applies to all cases of shared-memory programming.

Depending on which of the two sides is the initiator of the data exchange, we characterize the Communication as *Push* or *Pull*. Pull communication, where the consumer initiates data transfers, is the default shared-memory communication. We present below communication via cache-coherence in Section 2.1.1 and explicit communication in Section 2.1.2

### 2.1.1 Cache-based Implicit Communication

When the producer updates its local copy and the consumer had an old copy, the latter will get invalidated. When the consumer later discovers that it needs the (new) data, it suffers the latency of a miss or a remote read. This exchange is depicted in the left side of Figure 2.1: A miss is sent to the directory, which in

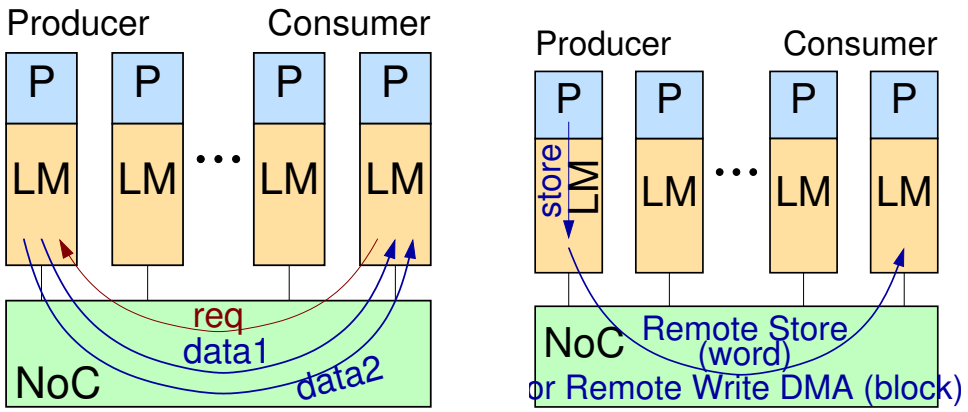



Figure 2.2: Explicit Communication with RDMA, pull type (left) push type (right)

turn forwards the request to the producer that has the updated copy. Then the requested blocks are sent to the consumer either through the directory, or – in an optimized protocol – directly from the producer to the consumer. The total number of messages is three, two control (small) and one data message. To transfer many blocks of data, additional miss requests can proceed in overlap with the earlier ones.

The right side of Figure 2.1 depicts the push type communication, where the producer initiates data transfers. When the producer produces and writes the next value, it first has to obtain exclusive access to the cache block. An upgrade request is sent to the directory, which invalidates the consumer's copy, and allows the producer to complete its write. A subsequent read on the consumer's side will result in a miss that is sent to the directory. In an optimized protocol, the directory controller will issue a "forward" request to the producer, which will then transfer its data directly to the consumer. The total number of messages is four control (small) messages plus the data transfer message, and again, the transfer of multiple blocks of data can proceed in a pipelined fashion, overlapped with the earlier transfers.

### 2.1.2 Scratchpad-based Explicit Communication

The left part of Figure 2.2 shows the pull scenario with explicit communication and RDMA operations. The consumer has knowledge of the producer's addresses, and issues an RDMA request directly to it. Then a stream of data is sent as a reply and placed in the consumer's memory. This transfer costs one control (small) message plus a data message, while the amortized cost for large volumes of communicated data is just the data transfer messages themselves.

The right side of Figure 2.2 shows the push scenario with explicit communication and RDMA operations. The producer has knowledge of the consumer's addresses, and issues a remote-write (for a single word of data) or a RDMA request (for a larger block of data) directly to it. Then a stream of data is directly sent and placed in the consumer's memory. The transfer cost in this case is just the data message(s), and large transfers can proceed at network speed in a pipelined fashion.

### 2.1.3 Comparison of Implicit vs. Explicit Communication

We have measured the messages required for communication in previous sections and we compare the cost of explicit communication versus implicit communication using coherent caches, and found a reduction in the number of network packets by factors on the order of *two to five times* for the most common communication patterns. While the comparison using message counts is not directly related to performance due to overlapping, there are concrete advantages in the explicit communication architecture: we expect it to scale better, since it does not require a coherence protocol and it offers reduced memory latency to critical data. We also expect that it will consume less energy, as it needs to switch fewer packets for the same data transfer. The network energy consumption is related not only to the packet volumes, but also to the total number of packets. The reason is that at the control plane each packet – however small – is processed fully for every hop in the network. We present a quantitative evaluation with application benchmarks in Chapter 5.

## 2.2 Task-Based Programming Models

This thesis considers a family of the task-based programming models, where a program is split into tasks. Tasks are asynchronous function calls performing chunks of work that run to completion. We describe below the most popular task-based programming models and libraries.

#### OpenMP

OpenMP allows programmers to easily convert a serial program into parallel with *pragma* annotations in the code. Moreover, OpenMP version 3.0 defines directives to support task parallelism [33] and version 4.0 extends the latter support by allow-

ing dependencies between tasks to be specified. Tasks derive from statements, code blocks, or functions that are annotated with the *task* pragma. The *taskwait* directive offers synchronization between tasks and the *barrier* directive allows waiting until all outstanding tasks complete. OpenMP pragmas are converted into calls to the underlying runtime library at compile-time and are supported by all major compiler toolchains.

#### Cilk

Cilk/Cilk++ [34] is one of the most popular task-based programming models. The language offers a *spawn* statement that allows independent tasks, which may run in parallel, to be specified. The *sync* statement allows a parent task to wait until all of its child tasks have completed. Tasks are enqueued in per-worker *deques* and the scheduler dynamically decides at run-time which tasks are executed. The Cilk scheduler applies randomized work-stealing to utilize all worker threads and balance load. Each worker executes tasks from the top of its local deque or "steals" from the bottom of other workers' deques. Blumofe et al. [35] showed that Cilk programs execute in almost optimal time and take optimal space, within a constant factor, when compared to the serial elisions of the respective programs.

#### Thread Building Blocks - TBB

Threading Building Blocks (TBB) [36] is a C++ framework that was developed by Intel Corporation to ease parallel programming on multicore architectures. The TBB library is based on templates and offers a rich set of constructs to support the most common parallel programming patterns. The programmer specifies "tasks", rather than threads, and the underlying run-time library maps and schedules these tasks dynamically to worker threads. Besides the typical fork-join task dependence pattern, TBB supports acyclic graphs of dependent tasks using reference counts; dependent tasks are spawned when their reference count becomes zero. The TBB scheduler is based on task-stealing and is similar to Cilk. The scheduling strategy is depth-first work and breadth-first stealing.

#### Task Dataflow Programming

The task-based dataflow programming models aim to simplify parallel programming by discovering task dependencies at runtime and dynamically extracting task parallelism. Such models require the programmer (or the compiler) to identify

```
1  // the tasks required for Cholesky decomposition
2  void dgemm_blk(N, BS, double A[N][N],
3                        double B[N][N], double C[N][N]);
4  void dsyrk_blk(N, BS, double A[N][N], double C[N][N]);
5  void dpotrf_blk(N, BS, double A[N][N]);
6  void dtrsm_blk(N, BS, double T[N][N], double B[N][N]);
7  ...
8  double data[N][N];
9  ...
10 for (int j=0 ; j < N ; j+=BS) {
11   for (int k=0 ; k < j ; k+=BS) {
12     for (int i=j+1 ; i < N ; k+=BS) {
13       double * A = &data[i][k];
14       double * B = &data[j][k];
15       double * C = &data[i][j];
16
17       #pragma xss task input(A{0:BS}{0:BS}) \
18                        input(B{0:BS}{0:BS}) \
19                        inout(C{0:BS}{0:BS})
20       dgemm_blk(N,BS,A,B,C);
21     }
22   }
23
24   for (int i=0 ; i < N ; i+=BS) {
25     double * A = &data[j][i];
26     double * C = &data[j][j];
27     #pragma xss task input(A{0:BS}{0:BS}) \
28                      inout(C{0:BS}{0:BS})
29     dsyrk_blk(N,BS,A,C);
30   }
31
32   #pragma xss task inout(data{0:BS}{0:BS})
33   dpotrf_blk(N,BS, &data[j][j]);
34
35   for (int i=j+1 ; i < N ; i+=BS) {
36     double * T = &data[j][j];
37     double * B = &data[i][j];
38     #pragma xss task input(T{0:BS}{0:BS}) \
39                      inout(B{0:BS}{0:BS})
40     dtrsm_blk(N,BS,T,B);
41   }
42 }
43
44 #pragma xss waitall
45 ...
```

Figure 2.3: Cholesky decomposition using the task dataflow programming model

tasks (functions) that may run in parallel, annotate the memory footprint of their arguments (addresses), and declare the side-effect of each task argument in memory (read/write). Representative examples of such programming models include OmpSs/SMPSs [23, 24, 37], BDDT [38], Legion [39], Serialization Sets [25] and other proposals that follow similar concepts and techniques [28, 40, 41].

The underlying runtime libraries use the memory footprints and the side-effects of each task argument to identify task dependencies and build dependency graphs as directed acyclic graphs (DAGs) at runtime. Independent tasks are immediately scheduled for execution on the available cores, while dependent tasks are kept in internal data structures and queues, waiting until all of their dependencies are satisfied. Tasks in this model may execute out-of-order using the scheduling techniques followed in processors, while respecting task dependencies in the same way that processors respect register dependencies, i.e. true dependencies (RAW), anti-dependencies (WAR) and output dependencies (WAW). Deterministic execution and the preservation of task dependencies are guaranteed by the order in which the main application thread issues tasks, similarly to the order that instructions are issued in a sequential program.

In this work, we consider the programming model proposed in [24], which is based on *C pragmas* and follows syntax similar to the popular OpenMP task pragmas [33]. We also consider the extensions to this programming model that provide support for multi-dimensional memory regions in task arguments [37]. The side-effects of task arguments may be declared as one of: *(i) input:* for read-only arguments, *(ii) output:* for write-only arguments[1], and *(iii) inout:* for arguments that are both read and written. A code example of Cholesky decomposition with the task dataflow programming model is shown Figure 2.3.

## 2.3 Related Work

This section presents related work for the architectural support we propose in Chapter 3. Our main focus is on literature relevant to our hybrid cache/scratchpad memory hierarchy, and the hardware/software co-design for task-based dataflow programming models in cache-based memory hierarchies.

***Configurable Local Memory***

Configuration of memory blocks has been studied before in the Smart Memo-

[1] Arguments that are guaranteed to be written before they are read may also be declared as output.

ries [42] project, but from a VLSI perspective. They demonstrate that using their custom "mats", i.e. memory arrays and reconfigurable logic in the address and data paths, they are able to form a big variety of memory organizations: single-ported, direct-mapped structures, set-associative, multi-banked, local scratchpad memories or vector/stream register files. The TRIPS prototype [43] also implements memory array reconfiguration, but in very coarse granularity. They organize arrays into memory tiles (MTs), which include an on-chip network (OCN) router. Each MT may be configured as an L2 cache bank or as a scratchpad memory, by sending configuration commands across the OCN to a given MT.

Cache-scratchpad configurability has been proposed elsewhere and a few of its variants are supported in some embedded processors. Ranganathan et al. [44] proposed associativity-based partitioning and overlapped wide-tag partitioning of caches for software-managed partitions (among other uses). The cache subsystem in ARM and PowerPC architectures allow locking of cache contents. Intel's Xscale also allows per line locking for virtual address regions, which are either backed by main memory, or not. The design of our hybrid cache/scratchpad memory generalizes the use of line state for configurable communication initiation and synchronization, in addition to locking lines in the cache.

***Interprocessor Communication Hardware***

Network interface (NI) placement in the memory hierarchy has been explored in the past. In 90's, the Alewife multiprocessor [45] explored an NI design on the L1 cache bus to exploit its efficiency for both coherent shared memory and message passing traffic. At about the same time, the Flash multiprocessor [46] was designed with the NI on the memory bus for the same purposes. Cost effectiveness of NI placement was evaluated by assessing the efficiency of interprocessor communication (IPC) mechanisms. Mukherjee et al. [47] demonstrated highly efficient messaging IPC with a processor caching buffers of a coherent NI, placed on the memory bus. Streamline [48], an L2 cache-based message passing mechanism, is reported as the best performing in applications with regular communication patterns among a large collection of implicit and explicit mechanisms in [49]. Moreover, NI Address Translation was extensively studied in the past to allow user-level access, overcoming operating system overheads [50], and leveraging DMA directly from the applications [46].

Syncretic adaptive memory (SAM) [51] integrates a stream register file (SRF) with a cache, exploiting compiler mapping of generalized streams. SAM targets

a stream processing environment and does not provide support for direct communication between cores. Exploiting caches for streaming data in general purpose systems was considered in [52] via hardware support and in [53] via the compiler. Our design integrates equivalent and more scalable mechanisms inside caches than those of [52], providing virtualized RDMA support for efficient bulk transfers.

Leverich et al. [54] provide a detailed comparison of cache versus partitioned cache-scratchpad on-chip memory systems for CMPs, and find that hardware prefetching and cache optimizations eliminate the advantages of the mixed environment. However, they consider communication between on-chip cores and off-chip main memory. By contrast, our evaluation for on-chip core-to-core communication (between scratchpad memories), shows that explicit communication offers significant traffic and energy reduction.

***Hardware and Software Prefetching***

There is a vast amount of previous work on hardware prefetchers that try to predict memory access patterns and prefetch data without any software guidance such as [55, 56]. However, we propose a hardware-software approach for prefetching that is based on the fully accurate knowledge about task memory footprints, known a-priori by the runtime software (before tasks start executing).

Guided region prefetching (GRP) [57] is a related scheme that augments load instructions with compiler-generated hints to improve the accuracy of a hardware prefetcher. GRP requires sophisticated compiler analysis, the hints are not always accurate, and the prefetches are triggered by L2 misses during the execution of code. Our approach differs from GRP, since we use fully accurate memory footprints and the data are fetched early before task execution with the potential to hide all L2 misses.

Most pertinent to this work is Streamware [53], and the related architectural support [52] that target stream processing. They propose a software programmable Stream-Load-Store (SLS) hardware unit that resembles EBP, and is used by the runtime software. However, they do not tackle the problem of cache pollution and interference due to prefetching, as we propose with the use of ECM. ARM's Preload Engine (PLE) [58] that is also similar to EBP does not address cache pollution either.

***Software Guided Cache Replacement***

"KILL" [59] and "Evict-me" [60] are two related schemes that try to improve cache replacement decisions and reduce pollution in the presence of prefetching. Both

of them are based on sophisticated compiler analysis and instruction hints (using ISA modification) that are used in conjunction with the replacement policy. These approaches try to identify which data will not be used in the future and mark this data appropriately. In the task-based execution context, this approach could only be useful for the data of the current task, while there can be no information about the behavior of the next task, the data of which can be prefetched. In addition, when a task completes, such schemes would require massive marking (or eviction) of each task's dataset. The latter could be an erroneous behavior if some of the data is reused by the next task. ECM on the other hand, easily handles all these cases and only requires software to advance the local epoch. Moreover, the epoch quotas offer an additional criterion to throttle prefetching.

***Prefetch Aware Cache Replacement***

PACMan [61] is a relevant prefetch-aware cache management hardware scheme that builds on top of RRIP [62]. PACMan tries to reduce cache pollution and prefetch interference by handling demand and prefetch requests separately. To achieve this effect, PACMan modifies the cache insertion and hit promotion policies. Although this scheme might perform well for intra-task prefetching (when data for the current task is prefetched), it cannot handle inter-task prefetching. ECM addresses prefetching across tasks and helps EBP to throttle prefetches in order to reduce traffic and pollution.

***Software Managed Coherence***

Some recent research prototypes adopt the use of non-cache-coherent memory hierarchies and rely on software to manage coherence. Intel's Single-chip Cloud Computer (SCC) [63], a 48-core manycore prototype chip, does not support hardware cache-coherence, but offers specialized on-chip "message buffers (MPB)" (8KB per core), and cache flushing instructions to allow software to maintain coherence. The software sends data by copying them to the MBP (put) and receives data by copying data from remote MBPs (get); sending more than 8KB data, requires segmentation. Our support for Software Guided Coherence (SGC) does not require software to copy any data, and allows arbitrarily large data transfers.

Intel's Runnemede [15] architecture does not also provide hardware cache-coherence but offers several cache-management instructions to maintain coherence. The architecture provides the following instructions at cache-line granularity: prefetch, invalidate (remove without writing back dirty data), and update (from backing store). Our SGC primitives, have lower software overhead because they

can operate with larger than cache-line granularities, and allow for cache-to-cache transfers without always involving the backing store, thus save more energy.

Fensch [64] proposes an OS-based alternative to hardware cache-coherence. They propose mapping lines to physical caches at page-level with OS support (first touch) and offer hardware support with a TLB-like structure that identifies remote cache accesses using virtual addresses. This scheme does not require any hardware cache-coherence protocol, and also supports some controlled migration and replication of data. On the other hand, our SGC scheme supports finer granularities, as fine as a cache-line, allows bulk prefetching of cache-lines, does not perform remote accesses repeatedly (saves traffic and energy), and avoids the OS overhead.

The Rigel [65] architecture also manages cache-coherence under software control. The processor is organized in clusters, with 8 cores per cluster, and features a shared "cluster cache" – L1 cache. Another level of banked "global cache" allows sharing between the clusters. Coherence between the cluster caches and the global cache banks is enforced by software, using either eager or lazy synchronization. Two types of memory operations are defined: "local" (in the cluster cache) and "global" that bypass the cluster cache. The architecture offers cache management instructions to flush or invalidate the cluster cache, at the granularity of cache-line and the entire cache. Our support for SGC does not rely on remote accesses (cache bypass) or flushing, which spend energy on data movements, but performs direct cache-to-cache transfers to minimize traffic and reduce the associated energy.

Moreover, there are relevant proposals that assume software properties such as "data-race freedom" to improve directory-based cache-coherence in terms of traffic, latency, and storage. The VIPS coherence protocol [66] removes the directory and the invalidations, by applying a write-back policy for private data and a write-through policy for shared data. DeNovo [67] is based on Deterministic Parallel Java [41] to identify data regions and performs region self-invalidation using compiler hints. The DeNovo protocol does not keep sharers in the directory, however, writers have to "register" in the shared cache level and some form of indirection is required. Cohesion [68] is hybrid hardware/software memory model that permits coherence for memory regions to be managed by either hardware protocols, or software. This design allows regions to move from the hardware coherence domain to the software coherence domain without copying. Our SGC scheme completely removes hardware cache-coherence protocols and directories, keeps everything in software, and offers only minimal hardware primitives to facilitate direct data movement in the memory hierarchy.

# 3 Architectural Support

This chapter proposes architectural support that allows software to manage the memory hierarchy, control locality, and guide data movement. We present low overhead hardware communication primitives that can be exploited by software to reduce the energy of communication in manycore systems.

First, we design a hybrid cache/scratchpad memory hierarchy with unified support for implicit and explicit communication using common hardware primitives, Section 3.1. Then, we present a hardware/software co-design with task-based programming models for cache-based memory hierarchies. Our hardware support allows software to manage cache locality and orchestrate data movement, in systems with and without cache-coherence, Section 3.2.

## 3.1 Hybrid Cache/Scratchpad Memory Hierarchy

This section proposes architectural support to unify the hardware mechanisms for implicit and explicit communication. We provide run-time configurability for the local memory resources and allow them to operate as a mix of cache and scratchpad memory. We merge the cache controller and network interface functions and provide fast hardware primitives for explicit inter-processor communication and

synchronization that can be used at user-level.

Section 3.1.1 overviews the architecture of the proposed integrated memory hierarchy and network interface, along with the supported synchronization primitives. Section 3.1.2 presents our approach for run-time configurable local memory that allows portions of the local cache to operate as scratchpad memory. Section 3.1.3 explains our support for virtualized user-level DMA and how the cache controller and the network interface integrate. Section 3.1.4 introduces hardware primitives for advanced inter-processor communication and synchronization. Details about the FPGA prototype that implements our proposed hardware support appear in Appendix A. Moreover, we study the case for Coherent RDMA and present our paper design in Appendix B.

### 3.1.1 Architecture Overview

Memory hierarchies of modern multicore computing systems are based on one of the two dominant schemes – multi-level caches, or directly addressable local scratchpad memories. Caches transparently decide on the placement of data, and use *coherence* to support communication, which is especially helpful in the case of *implicit* communication, i.e. the input dataset and/or the producer of data are not known in advance. However, caches lack determinism and make it hard for the software to explicitly control and optimize data transfers and locality in the cases when it can intelligently do so. Furthermore, coherent caches scale poorly to over hundreds of processors. Scratchpad memories (or software-managed memories) are popular in embedded [69] and special purpose systems [18, 19, 70], because they offer predictable performance – suitable for real-time applications – and also offer scalable performance by allowing explicit control and optimization of data placement and transfers. *Explicit* communication uses *remote direct memory accesses (RDMA)*; it is efficient, and it becomes possible in the cases when the producer knows who the consumers will be, or when the consumer knows its input dataset ahead of time. Recent advances in parallel programming and runtime systems [20, 22, 71] allow the use of explicit communication with minimal burden to the programmers, who merely have to identify the input and output data sets of their tasks.

Our goal is to provide *unified* hardware support for *both implicit and explicit* communication within the same address space as shown in Figure 3.1. To achieve low latency, we integrate our mechanisms close to the processor - in the upper cache levels, unlike traditional RDMA that is implemented at the I/O bus level. We

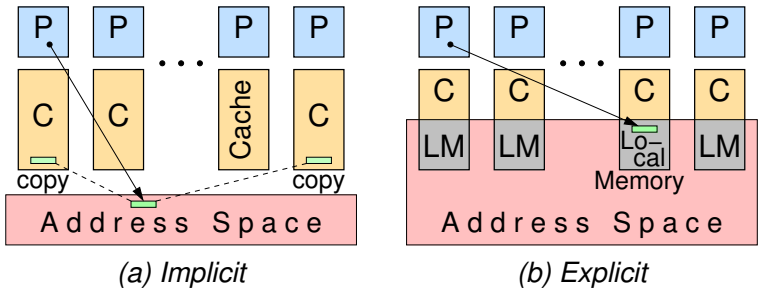



Figure 3.1: Implicit vs. Explicit Communication

provide *configurability* of the local SRAM blocks that lie next to each core, so that they operate either as cache or scratchpad memory, or as a dynamic mix of the two. Configurability is at *run-time* allowing different programs with different memory requirements to run on the same core, or even different stages of a program to adapt the underlying memory to their needs. We also strive to merge the hardware required by the cache and the scratchpad memory into one *integrated* Network Interface (NI) and Cache Controller (CC), in order to economize on circuits.

We propose a simple, yet efficient, solution for cache/scratchpad memory configuration at run-time and a common NI that serves both cache and scratchpad communication requirements. The NI receives DMA commands and delivers completion notification in designated portions of the scratchpad memory. This allows the OS and runtime systems to allocate as many NI command buffers as desired per protection domain, thus effectively virtualizing the NI, while providing user-level access to its functions so as to drastically reduce latency. We improve SRAM utilization compared to traditional NIs (that used dedicated memories) by sharing the SRAM blocks between the processor and the NI, and we sustain high-throughput operation by organizing these SRAM blocks as a wide interleaved memory. Scratchpad space can be allocated inside the L1 or L2 caches and consequently the NI is brought very close to the processor, reducing latency. Our NI also offers fast messages, queues, and counters, as *synchronization* primitives, to support advanced interprocessor communication mechanisms.

Our proposed architecture targets chip multiprocessor systems with tens or hundreds of processor cores: each core has at least two levels of private caches and communicates with shared memory using Global Virtual Addresses [72]. The next sections describe run-time configuration of the local SRAM blocks as cache and/or scratchpad. We explain how scratchpad memory can be used to support virtualized NI command buffers, and present our hardware synchronization prim-

itives. Details about the FPGA prototype that implements our proposed hardware support appear in Appendix A.

### 3.1.2 Run-time Configurable Scratchpad

Scratchpad memory space in our scheme is declared as a contiguous address range and corresponds to some cache lines that are pinned (locked) in a specific way of the cache, i.e. cache line replacement is not allowed to evict (replace) them.

Scratchpad areas can be allocated inside either L1 or the L2 caches. Most applications seem to require relatively large scratchpad sizes, so the L2 array is a more natural choice. Moreover, L2 caches offer higher degree of associativity, hence more interleaved banks. Although L2 latency is higher than L1, the performance loss due to this increased latency is partly compensated in two ways: *(i)* The L2 and scratchpad supports pipelined random accesses (read or writes) at a rate of 1 per clock cycle; *(ii)* configurable parts of the scratchpad space can be cacheable in the (write-through) L1 caches[1].

Owing to the use of progressive address translation [72], caches and scratchpad operate with virtual addresses, and the TLB only needs to be consulted when messages are transferred through the NI and the NoC to another node. In lieu of the processor-TLB, our architecture has a small table called Address Region Table (ART). As shown in Figure 3.2, ART provides a few bits that determine whether an address region contains cacheable or directly addressed (scratchpad) data. This is important when remote scratchpad regions are addressed, so that the hardware accesses them remotely, rather than locally caching them. It also obviates tag bit comparison to verify that a memory access actually hits into a scratchpad line; hence, tag bits of scratchpad areas are freed, and can be used for other purposes, such as implementing communication semantics for RDMA commands, counters, and queues that will be described shortly. Regions marked as local scratchpad in the ART occupy a set of blocks in the data portion of an L2 memory "way" block, such that low-order bits (the cache index) are compatible with the scratchpad address. The region can be freely allocated into any of the cache "ways", with ART identifying the "way" used. Each of the blocks in the region is marked as non-evictable in its state bits. This marking allows the distinction of memory access

[1] Write-back policy can also be used, provided that coherence between L1 and L2 is maintained. However, the write-through policy simplifies coherence without any performance loss. The inclusion property assumed here, is more intuitive than exclusion that would require moving locked lines between the cache levels.

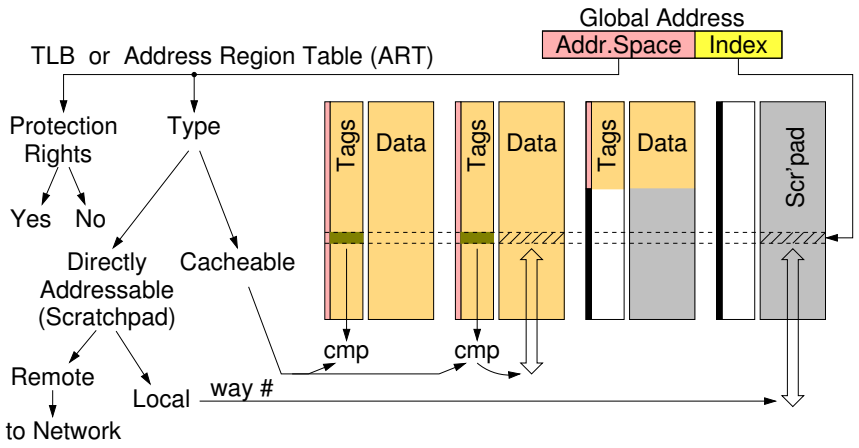



Figure 3.2: Memory access flow of the hybrid cache/scratchpad memory.

semantics at cache block granularity, and is used to ignore the actual tag-matching of the hit logic, as well as to prevent replacements. This mechanism allows for run-time configurable partitioning of the on-chip SRAM blocks between cache and scratchpad use, thus adapting to the needs of the application that is being run at each point in time.

### 3.1.3 Virtualized User-Level DMA

NI command buffers are DMA control areas that are allocated upon user software demand and reside in normal scratchpad regions. These buffers share the same ART entry with normal scratchpad and the distinction is made using a special bit (cache-line state), set upon allocation. Any user program can have dedicated NI command buffers (DMA registers) in its scratchpad region; this allows a low-cost virtualized DMA engine where every process/thread can have its own resources. To ensure protection of the virtualized resources, we also utilize permission bits in the ART and demand the OS/runtime system to update the ART appropriately on context switches. Moreover, the inherent support for dynamic number of DMAs at run-time promotes scalability and allows the processes to adapt their resources on the program's communication patterns that might differ among different stages of a program.

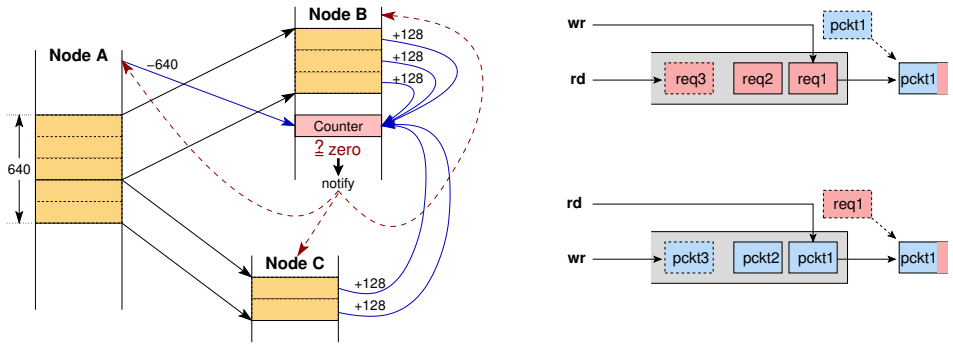
DMAs are issued as a series of store instructions – to provide the arguments: *(i)* opcode, *(ii)* size, *(iii)* source address, and *(iv)* destination address – destined to words within command buffers, that gradually fill DMA command descriptors, possibly out-of-order. The NI uses a command protocol to detect command com-

pletion and inform the DMA engine that a new command is present. All new and pending commands are kept in a *Network Job List* that is served by the NI according to its scheduling policy. When serving DMAs, the NI generates packets along with their customized lightweight headers. RDMA packets belong to one of the two primitive categories: *Write* or *Read*. The NI carefully segments the DMAs into smaller packets when they exceed the maximum network packet size. The cache controller uses the *Network Job List* to request write-backs upon replacements and fills upon misses: the same mechanisms serve DMA transfers as well as cache operations.

### 3.1.4 Additional Interprocessor Communication Primitives

We provide additional NI features that offer additional flexibility to the programmer in order to achieve more efficient communication between processors. We implement *Remote Stores* with write combining, to scratchpad regions of remote processors, in order to optimize remote access latency [73]; the ART can identify scratchpad ranges as remote. NI command buffers, described above, can also be used for fast *Messages*, allowing atomic, multi-word transfers. Message data are provided directly by the processor and no source address is needed. In addition, an explicit acknowledgment address can be specified to support software notification of transfer completion; acknowledgment addresses are allowed to be "null" to deactivate the mechanism. Multi-segment RDMA completion notification requires additional hardware support as described below.

We implement *Counters* with atomic add-on-store capability, also hosted in scratchpad space, as a primitive to support completion notification for an unordered sequence of operations, such as multiple RDMA transfer completion, barriers, and other synchronization operations. Counters are initialized with a value (e.g total transfer size in bytes) via local or remote stores and trigger single-word writes to notification addresses when they expire (reach zero). For RDMA transfer completion, software can specify an explicit acknowledgment address targeting a counter, which will gather all partial acknowledgments for DMA segments, as illustrated in Figure 3.3a. In the scenario shown, a single RDMA transfers 640 bytes. The destination region is mapped in the scratchpad of two separate nodes (nodes B and C). When all acknowledgments arrive at the counter, as well as the initialization value of -640, the counter triggers three notifications towards preconfigured addresses on nodes A, B and C. Counters is the only support required by the network interface for adaptive/multipath routing NoC optimizations, since RDMA transfer comple-




(a) Multi-packet RDMA completion

(b) Multiple-reader Queues

Figure 3.3: Illustrating our proposed interprocessor communication primitives.

tion notifications will also work correctly with out-of-order packet arrivals. The only requirement for correct operation of counters is that the NoC never generates duplicate packets.

Finally, we provide *Remote Queues* as an appropriate level of abstraction for multiprocessor synchronization [74]. Queues are hosted inside scratchpad regions and their configuration (size, pointers and item granularity) can be programmed in the tags of special control lines. *Single Reader Queues* are provided to support efficient many-to-one control information exchange, with receiver polling to a single location. Moreover, *Multiple Reader Queues* (mr-Qs) are provided as a means for many-to-many synchronization, by allowing asynchronous write (enqueue) and read (dequeue) operations from any number of processors. As shown in Figure 3.3b, read requests arriving at an empty mr-Q are recorded, waiting until corresponding writes arrive, thus effectively *matching* read and write requests in time. Upon successful matching, a response packet is generated; matching is dual, i.e. either writes or reads might wait to be matched. Multiple reader queues can also be used for locks or to accelerate task/job dispatching.

## 3.2 Cache-based Memory Hierarchy

This section presents a hardware/software co-design with task-based programming models for cache-based memory hierarchies. We propose architectural support that allows software to manage cache locality and orchestrate data movement, in systems with and without cache-coherence. The runtime software can exploit our hardware support to improve performance and reduce the energy consumption.

Section 3.2.1 overviews the inherent characteristics of task-based dataflow programming models, makes observations on their behavior in the memory system, and discusses the opportunities for architectural support and hardware/software co-design. Section 3.2.2 proposes the *Explicit Bulk Prefetcher (EBP)*, a programmable prefetch engine that allows software to accurately prefetch data ahead of time and improve cache locality in task-based programs. Section 3.2.3 introduces *Epoch-based Cache Management (ECM)*, a generic lightweight mechanism to guide cache replacement decisions, assign local cache resources to tasks, and isolate the effects of prefetching. Section 3.2.4 explains the case for *Software Guided Coherence (SGC)* in non-cache-coherent systems and presents hardware primitives that allow runtime software to maintain coherence at task granularity. Finally, section 3.2.5 discusses the intended use of the proposed hardware primitives by task-based runtime software.

### 3.2.1 Opportunities for Software Guidance

One common approach to parallel programming is to decompose a program into a set of tasks and distribute them among the processing elements. Many task-based programming models have been proposed in the literature [23, 33, 34, 36] and promise to ease programming effort by abstracting out the elements of parallel programming that are traditionally considered hard and time consuming, such as scheduling, synchronization, and locality optimizations. This work considers a class of emerging task-based dataflow programming models where the memory footprint of each task is declared by the programmer and the runtime software automatically detects task dependencies based on these footprints and schedules independent tasks concurrently [23–25, 39, 40]. In this context, we focus on the memory behavior of fine-grain tasks and their impact on cache locality, which greatly affects performance and energy consumption. The use of fine-grain tasks can unleash large amounts of parallelism and has the potential to allow many-core computing resources to be utilized.

Each task has its own unique memory footprint and the associated data will eventually be transferred into the portion of the underlying memory hierarchy that is closest to the core that executes this task, i.e. L1 and/or L2 cache. Given that tasks are separate units of work, each with its own memory footprint, reuse of cache contents among tasks is a difficult problem that locality-aware schedulers are trying to alleviate [75]. However, task data reuse is not always possible, and depends on the distance between producer and consumer tasks, which is an intrinsic

characteristic of each application. Moreover, several types of applications do not benefit from a single type of scheduler and their behavior may be incompatible with specific scheduling algorithms. Often times, applications benefit from load-balancing and work-stealing, which makes locality a conflicting goal; the most extreme case of locality scheduling dictates that all tasks execute sequentially in a single core.

The task-based dataflow programming models use the memory footprints of tasks to build dependency graphs (DAGs) and maintain significant amount of information that is used to dynamically drive runtime decisions. Essentially, the runtime system discovers producer-consumer relations among tasks, maintains such knowledge internally and uses it to schedule and execute the tasks in the correct order. Owing to this, the most recent copy of a task argument will reside in the memory hierarchy (L1 cache or higher cache/memory levels) of the core that last executed the "producer" task. Effectively, the runtime system keeps, in software, knowledge and state equivalent to hardware directories in cache-coherent systems. However, the underlying memory architecture is still agnostic of what runtimes are trying to achieve. Therefore, hardware decisions are based on rather simplistic assumptions. Our thesis is that the runtime software maintains important semantic knowledge regarding the execution sequence and memory footprints of tasks, which can be shared with the underlying hardware to achieve an effective hardware-software synergy. To this end, we propose architectural support and runtime co-design to improve cache locality, optimize execution, and reduce the energy footprint of task-based workloads.

We present the *Explicit Bulk Prefetcher (EBP)*, a programmable prefetch engine that can be utilized by the runtime software to prefetch task data. EBP is reminiscent of an RDMA (remote direct memory access) engine, but it is designed for cache-based architectures, integrates with the local cache hierarchy of each core, and offers a low overhead memory-mapped interface that can be used at *user-level*. EBP enables runtime software to prefetch task data in bulk before each task executes and to perform common optimizations such as double-buffering. This form of software-directed prefetching can overcome some of the challenging issues with hardware-only prefetchers such as *timeliness*, *accuracy*, and *access pattern prediction*.

Although prefetching has the potential to improve cache locality and hide memory latency, its effectiveness is affected by the ability of the cache to keep the prefetched data. When applying double-buffering optimizations, prefetching can

pollute the cache and evict useful data. To address the latter issues and shortcomings, we propose *Epoch-based Cache Management (ECM)*, a mechanism that allows software to guide the cache replacement policy, expose its knowledge of tasks to the cache hierarchy, assign cache resources to them, and isolate the effects of prefetching. ECM is based on the notion of *Epoch*, which can be defined by software as the lifetime of a task, i.e. the time period during which a task executes. ECM offers a memory-mapped interface that allows software to advance epochs, i.e. signal the beginning of new tasks, and assign quotas to epochs, i.e. declare the space a task is allowed to allocate in the cache. All data accessed (or prefetched) by a task is associated with an epoch number in the cache. ECM guides the cache replacement policy, by allowing it to distinguish between data belonging to different tasks. The hardware cost of ECM is very small.

The nature of task-based dataflow programming allows the runtime system to identify producer-consumer relationships between tasks and offers the opportunity to arrange the data movement between the cores that execute these tasks. The potential for software-guided data transfers is extremely important for future many-core systems since it does not mandate hardware cache-coherence and thus, allows removing the associated control traffic overhead so as to reduce energy consumption in communication. We propose hardware primitives for *Software Guided Coherence (SGC)* in non-cache-coherent systems, to allow the runtime software to orchestrate fetching the most up-to-date version of the task arguments from the appropriate cache(s) and maintain coherence at task granularity.

### 3.2.2 Explicit Bulk Prefetcher

Task-based programming models with annotated memory footprints allow the runtime to know before-hand which data will be used by a task before that task executes. This observation offers the opportunity to prefetch task data in a timely fashion and improve cache locality. The *Explicit Bulk Prefetcher (EBP)* is a hardware unit that allows software to explicitly prefetch memory ranges that correspond to task arguments. Software can utilize EBP to prefetch data for the next task(s) waiting in the scheduling queue, effectively applying double- or multi- buffering, in order to minimize cache misses and improve task execution.

EBP is a programmable prefetch engine that offers a memory-mapped interface and accepts a set of commands at user-level. We design EBP as a per-core engine that operates on a private coherent L2 cache. We target L2 caches since their large capacity – when compared to L1 caches – makes them more suitable for bulk

prefetching and because processor pipelines and critical paths are less susceptible to changes in the L2 cache. Moreover, we choose a memory-mapped interface instead of a register-mapped interface, in order to avoid ISA changes and make our design less intrusive. EBP is reminiscent of existing RDMA engines [18, 58].

The memory-mapped interface offered by EBP, is designed to allow memory ranges to be specified with virtual addresses, thus offering fast user-level access with low software overhead. In order to translate virtual addresses[2] and ensure protection, EBP requires access to the local TLBs; typically the 2nd level TLB. The use of virtual addresses provides EBP with the capability to prefetch across page boundaries, which is a common limiting factor in hardware-only prefetchers. In addition, EBP can trigger page-table walking hardware early and minimize, or even hide, the effect of TLB misses.

### EBP Request Engine

The EBP engine supports *2D memory ranges* with a constant stride in order to minimize the number of required prefetch operations in common array patterns, such as blocking/tiling. The interface defines the following memory-mapped registers to initiate prefetch operations:

- *Address:* The starting virtual address for a prefetch.
- *Block Size:* The size (bytes) of each block.
- *Block Number:* The number of blocks to prefetch.
- *Block Stride:* A constant stride (measured in bytes) used for the calculation of the next block address.
- *Epoch:* This field is used by ECM as described later.
- *Opcode:* This field marks whether this data will be used as Read-Only or Read-Write. Based on this value, the associated cache-lines are requested with the proper coherence permissions: *Shared* or *Exclusive*.

Upon writing the "Opcode" register, all command fields are atomically enqueued in a "Command FIFO" and each command is served in-order by the internal "Request Engine" (Figure 3.4). The "Request Engine" converts each memory range into multiple cache-line aligned requests, performs address translation, and

[2] We assume that the L2 cache is physically tagged.

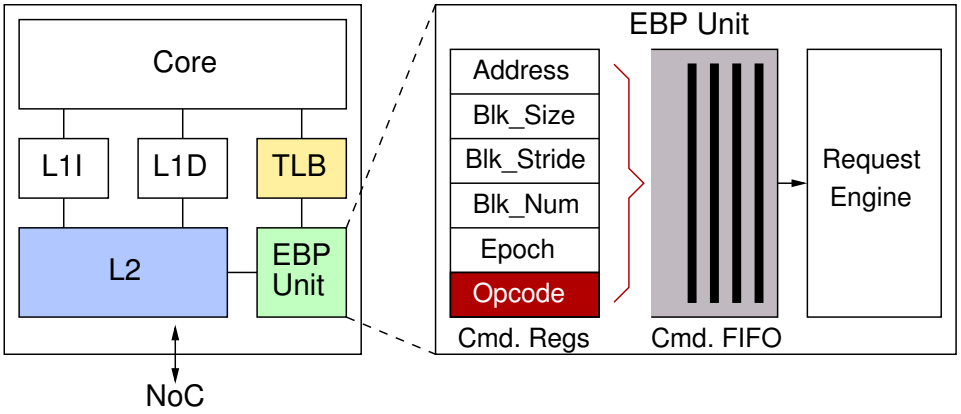



Figure 3.4: An overview of the Explicit Bulk Prefetcher (EBP)

probes the cache. If a cache-line is present in the cache with the appropriate coherence permissions, then the request is skipped. If the cache-line is not present, or present with "limited" permissions, then a new request is sent to the coherence directory to fetch or upgrade the cache-line. In case a cache set is full, an old cache-line is evicted to make space. The intermediate "Command FIFO" supports multiple outstanding prefetch operations (32 in our implementation). The "Request Engine" also supports multiple outstanding cache-line requests, the number of which is however limited by the number of miss status handling registers (MSHRs). We assume that up to 8 outstanding requests can be issued without occupying all MSHRs.

### 3.2.3 Epoch-based Cache Management

Current cache replacement policies base their decisions on the recent history of referenced cache-blocks and try to predict which blocks will be referenced in the near future. They typically assume that the most recent or the most referenced blocks should remain in the cache [62] and try to optimize this behavior. However, the behavior of task-based execution models is substantially different, since after the lifetime of a task, the reference history of many cache-blocks used by the completed task may be useless and can negatively affect performance.

The replacement decisions become even more challenging in the presence of prefetching [61], e.g. when utilizing EBP. The use of EBP has the potential to effectively hide memory latency when the software can initiate it ahead of time, before data is requested by a task, for example when the runtime software uses double-buffering. However, the effectiveness of EBP is also affected by the ability

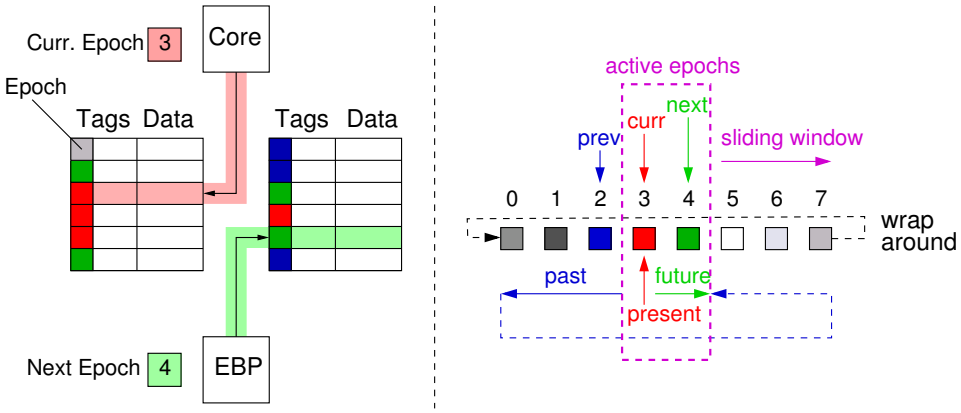



Figure 3.5: An overview of Epochs. Cache-lines accessed by the processor use the current epoch, while accesses from the prefetcher use the next epoch.

of the cache to keep the prefetched data. Prefetching is known to cause cache pollution, therefore double-buffering data for the next task may evict data needed by the current task. Likewise, the current task may evict data prefetched for the next task.

These inefficiencies offer an opportunity to improve performance and energy consumption by making the cache aware of the tasks' lifetimes and datasets. We propose *Epoch-based Cache Management (ECM)*, a hardware mechanism that allows software to expose the knowledge of tasks and their requirements to the cache hierarchy in order to improve replacement decisions, optimize cache locality, and minimize data movement.

### Epoch Essentials

ECM is based on the notion of *Epoch*, which can be defined as the time period during which a task executes, which we refer to as the *task lifetime*. Epochs are tracked locally on each core, in the form of a memory-mapped register visible to the local cache. Every memory access that arrives from the processor, is augmented in hardware with the current epoch and marks the corresponding cache-line. The epoch number is kept in the tag of each cache-line and occupies a few bits, e.g. 3 bits to support 8 epochs. ECM only requires the software to advance the epoch register at the beginning of a new task. The epochs are free to wrap around without any special handling. Figure 3.5 illustrates an overview of epochs.

The epoch number is essentially a short identifier that allows the cache to distinguish between data that belong to different tasks while maintaining a short his-

tory, i.e. data accessed by the last 8 tasks. The replacement policy can use the epoch numbers contained in the tags of each set and the current epoch register, in order to quickly filter old data and decide which cache-lines to victimize when needed. This strategy effectively prioritizes data used by the current task. When all cache-lines in a set belong to the current epoch, the replacement policy operates as it would do without epochs and uses the reference state of cache-lines (e.g. LRU bits).

### Epochs Quotas for Cache Space Allocation

However, when the runtime employs double-buffering optimizations and uses EBP, the cache must also handle another active task context, i.e. the next task and its data. The cache has to ensure that data between these active task contexts do not interfere in a destructive manner. To address the latter issue, we introduce software-controlled *Quotas* for epochs and their corresponding tasks. ECM offers a set of memory-mapped quota registers that enables software to assign a portion of cache space for each "active" epoch. We consider active epochs to be the "current" and "next" epoch, which correspond to the current task and the first waiting task in the processor's task queue. Older epochs do not have quotas. The software assigns quotas, expressed in number of bytes for the active epochs, using the memory footprint of each task, in order to reserve cache space for the task. We implement this scheme in private L2 caches.

The underlying hardware mechanism uses the quotas to construct flexible and lightweight partitions for each epoch. When a quota is assigned, the byte quantity is converted into equivalent number of ways, depending on the size and associativity of the cache. The scheme rounds up the quota to the closest multiple of equivalent cache ways and handles cases of over-booking; the sum of quotas cannot exceed the number of cache ways. ECM enforces the quotas in a best-effort manner and guarantees that each active epoch can allocate *at least* its assigned quota (ways) per set. However, an active epoch is allowed to allocate more than the assigned ways in a set, when another active epoch does not fully utilize its quota. Moreover, an active epoch is also free to use cache-lines that belong to old epochs.

### Cache Replacement With Epochs

The replacement policy counts the allocated ways for the active epochs in a per-set basis and based on the quotas, decides whether an epoch can allocate more space.

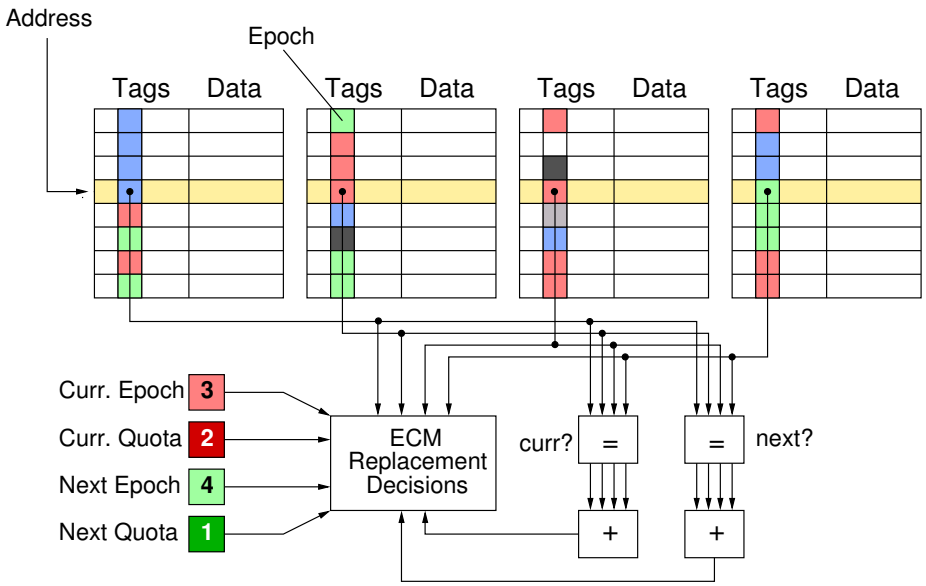



Figure 3.6: Cache replacement using epochs and quotas.

When a cache set is fully utilized with cache-lines that belong to active epochs, the replacement policy selects a victim that belongs to the requesting epoch, by consulting only the reference state bits (e.g. LRU bits) that belong to this epoch. Cache-line allocation for an active epoch is not tied to specific cache ways but is instead dynamically selected. An overview of cache replacement using ECM is illustrated in Figure 3.6.

When EBP is used in conjunction with ECM, each prefetch request is augmented with the "Epoch" field of the prefetch command, to signify whether this request belongs to the current or the next epoch. In addition, EBP probes ECM to discover whether its quota has exceeded, i.e. the corresponding set is full with cache-lines that belong to active epochs. When a set is full, EBP throttles prefetching by skipping requests destined to this specific set, in order to avoid evicting cache-lines that were recently prefetched.

ECM can be easily implemented at low hardware cost, and in fact we have already implemented it along with EBP in an FPGA prototype [76]. Adding 3 epoch bits per tag in a 256KB 8-way set-associative L2 cache has a memory overhead of only 0.5%.

### Generalization of Epochs

We have described before the use of epochs in a task-based programming environment, however, the epochs are a more generic mechanism that allows software to guide cache-replacement decisions and control data locality. Epoch-based cache management can offer locality guarantees equivalent to those found in software-managed memories (scratchpads).

We extend our hardware design to support a set of long-lived active epochs that have higher priority than normal epochs. We add an extra priority bit in the epoch field (e.g. a 4th bit) to support 8 high-priority epochs that are always active (i.e. epochs 8-15) and they are not influenced by epoch wrap-around; we also add the associated epoch quota registers. The replacement policy treats these high-priority epochs as "non-evictable" and never replaces the associated cache-lines in favor of other epochs. Cache-lines of high-priority epochs are only replaced by accesses that belong only to the same high-priority epoch. Cache replacement within a high-priority epoch uses only the reference state bits of this epoch (e.g. LRU bits). To preserve locality guarantees among high-priority epochs, we do not allow high-priority epochs to exceed their quota space. The hardware handles the transient case of resizing high-priority epochs, mainly when epoch quotas shrink, by marking this in the associated quota register and performing extra evictions when needed.

The software can use these 8 high-priority epochs to maintain the equivalent of 8 buffers in scratchpad memory. The software now manages epochs, instead of local memory addresses, and can issue EBP commands to prefetch data for these epochs; like it would do to fetch data into scratchpad memory buffers. A representative case where software can benefit from high-priority epochs is stencil codes (e.g. Jacobi). The parallel contexts in such codes, reuse a large part of an array across iterations, and exchange the boundary elements with their neighbors. During the boundary exchanges, the cache replacement policy may evict data that will be reused in the next step. The code may use high-priority epochs (long-lived) for the accesses of the "core" array part and normal epochs (short-lived) for the accesses (or prefetches) of the boundary elements. Essentially, the use of different types of epochs, allows software to guide cache replacement and expose different levels (and durations) of temporal locality, in order to minimize data movement.

### 3.2.4 Software Guided Coherence

The majority of processor chips with multiple cores implement cache-coherence to ease programmability. The coherence protocols are implemented purely in hardware, track the location of data copies (typically at cache-line granularity) throughout the memory hierarchy, and perform the data movement between cores transparently. As core counts increase and the trends for the coming years predict hundreds of cores per chip, hardware cache-coherence becomes increasingly expensive and inflexible. Designing efficient cache-coherence protocols and verifying [77, 78] them for a "sea of cores" is in doubt. Moreover, recent studies indicate that the cost of inter-processor communication and data movement will become, in future technologies, more expensive than computation in terms of energy and power [11,13,15,16]. Cache-coherence protocols incur significant cost in on-chip network traffic and energy mainly due to superfluous control packets that are exchanged between the directories and caches; we illustrate such behaviors in Section 2.1 and present a quantitative evaluation in Section 5.1.

Some early research prototypes of future manycore architectures have adopted the use of non-coherent caches [15, 79] and increasingly rely on software to maintain coherence. We advocate that non-coherent caches is a viable option for future manycore chips, as long as the architecture offers the appropriate hardware support to allow software to guide hardware data transfers and enforce coherence. Therefore, we propose architectural support for *Software Guided Coherence (SGC)* so that task-based dataflow programming models can exploit their internal knowledge about producer-consumer relationships among tasks, orchestrate data movement between the cores that execute these tasks, and maintain coherence. Our scheme is based on the observation that, the most up-to-date copy of a task argument will reside in the memory hierarchy (L1 cache or higher cache/memory levels) of the core that executed the "producer" task.

#### Support for Non-Coherent Caches

Our hardware support builds upon the *Explicit Bulk Prefetcher (EBP)*, presented in Section 3.2.2, and we augment it with extensions for non-coherent caches (EBP-NC). The most important addition is a "source core" field in the register set of EBP. To maintain coherence, the software should also provide, for each prefetch operation, a core number in the system, i.e. the core that last executed the "producer" task for each task argument. Each EBP-NC engine is capable of delegating/ser-

vicing commands to/from remote engines utilizing separate buffers for local and remote commands. Further details on EBP-NC are described later.

Correct task execution and coherent memory behavior can only be guaranteed if the process of prefetching task arguments has completed before task execution starts; (pre)fetching becomes an obligatory step and older versions of the associated data are overwritten. EBP-NC handles command completion using a set of programmable hardware "counters" that can be associated with commands; we support up-to 32 counters. Each counter measures the packet volume (bytes) transferred for the command(s) associated with it. The complete sequence of packets belonging to a command carries the associated counter number. Moreover, we propose the use of per core "mailboxes", i.e. dedicated hardware FIFOs, as a primitive for low-volume direct inter-processor communication that allows cores to exchange information, e.g. dispatching task descriptors. These hardware primitives are explained in detail in a later section.

### Data Transfers Between Non-Coherent Caches

Upon a transfer request, the local EBP-NC engine delegates the command to the EBP-NC engine of the source core (remote). Instead of breaking the request in cache-lines – the basic transfer unit – the data are requested in bulk, using a single block transfer, in order to save traffic and energy in the underlying network. The remote EBP-NC engine probes the local cache, at cache-line granularity, in order to send back the most up-to-date version of the data. Each destination engine updates the associated hardware counter when each cache-line arrives, in order to trigger transfer completion and notify the software to begin task execution.

However, requesting data from remote caches presents some non-trivial cases that require special handling, such as misses (data not present in the cache) and "dirty" cache-lines. Our EBP-NC engine defines two types of opcodes for data transfers: *(i)* Fetch and *(ii)* Fetch-with-Ownership (Fetch-O). The "Fetch" operation is intended for transferring read-only data, i.e. input task arguments, while the "Fetch-O" operation is intended for data that will be written, i.e. output and inout task arguments. The Fetch-O operation is required to ensure correctness and allows a cache to acquire ownership for specific data (cache-lines) from a previous remote owner/writer, by clearing the "dirty-bit" of the involved cache-lines in remote caches. If a specific cache-line is dirty in more than one cache, then it can cause inconsistent memory behavior if write-backs occur in wrong order.

When an EBP-NC engine serves block transfer requests, it "splits" each request

| **Single Chip with Directly Accessible Shared Memory** | | | | |
|---|---|---|---|---|
| Operation | Source Cache lookup result | | | Actions |
| | miss | hit-clean | hit-dirty | |
| Fetch | • | | | 1) cache forwards to memory<br>2) memory sends data |
| Fetch | | • | | 1) cache sends data |
| Fetch | | | • | 1) cache updates memory<br>2) cache sends data |
| Fetch-O | • | | | 1) cache forwards to memory<br>2) memory sends data |
| Fetch-O | | • | | 1) cache sends data |
| Fetch-O | | | • | 1) cache clears dirty-bit<br>2) cache sends data marked dirty |

Table 3.1: Actions required for software-guided coherence in a single chip that provides all cores with direct access to the shared memory.

into cache-lines and probes the cache at cache-line granularity. During each probe, the engine may find the requested cache-lines in one of the following three possible states: *(i)* not-present (miss), *(ii)* valid (hit-clean), and *(iii)* dirty (hit-dirty). Based on the two types of request opcodes (Fetch or Fetch-O) and the cache-line state, the EBP-NC engine in conjunction with the cache controller follow a set of predefined steps that appear extensively in Table 3.1. The latter table defines the behavior of transfer requests, when transfers take place between caches that share the same main memory and have direct access to it. Several cases that appear in Table 3.1 have common actions and we discuss below the details for each different case.

When the engine experiences a miss (either with Fetch or Fetch-O opcode), as shown in Figure 3.7a, then it forwards the request to the main memory which in turn responds to the cache that originally made the request; there is no need to bring the data to the intermediate serving cache. When the engine experiences a clean-hit (either for Fetch or Fetch-O opcode), as shown in Figure 3.7b, then it just sends the data to the requesting cache. In the latter two cases the incoming cache-lines, arriving at the receiver cache, are up-to-date in main memory, so the receiver cache can silently evict them and possibly fetch them again (upon processor demand) through normal cache misses.

However, when a cache-line is dirty in the cache, then different actions need to

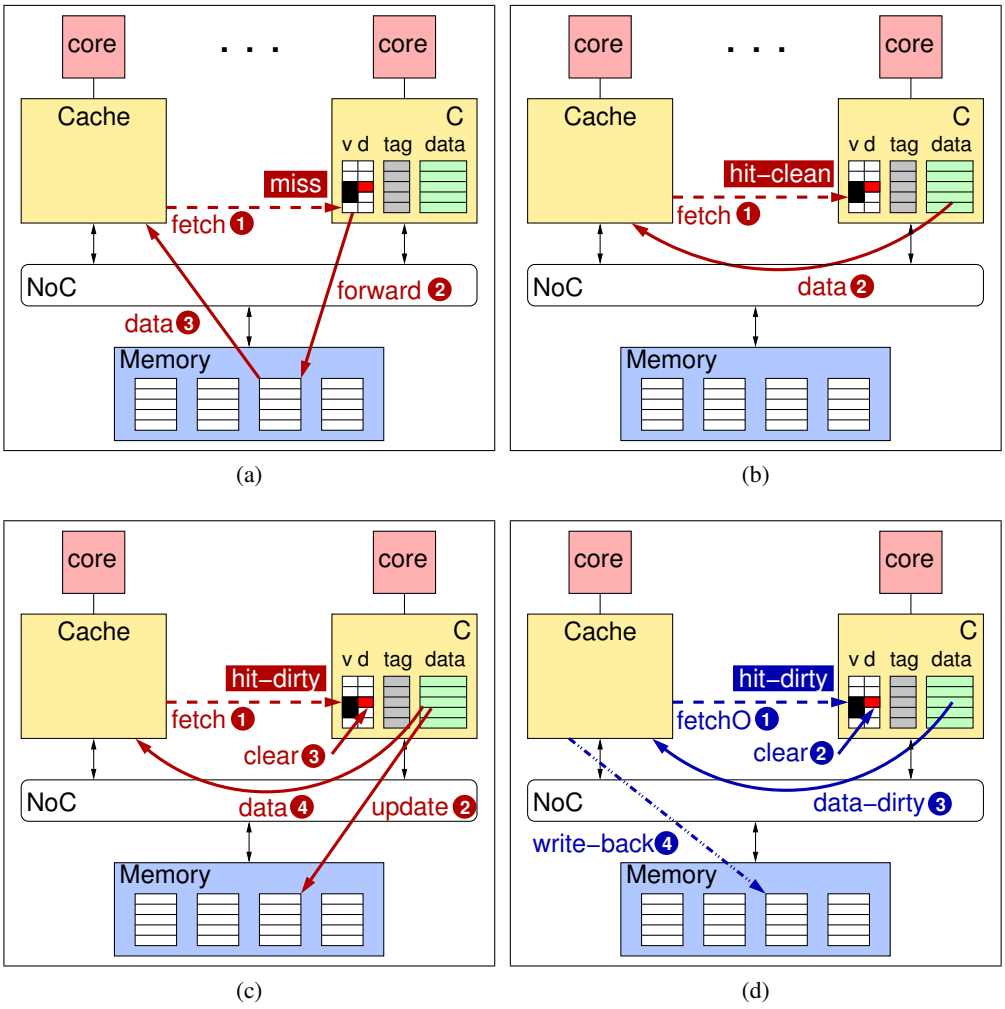



Figure 3.7: Transfers between caches in a single chip under software-guided coherence: All possible cases of a Fetch operation (a,b,c) and a Fetch-O operation (d) that hits on a dirty cache-line

be performed. When a Fetch request finds a cache-line dirty, Figure 3.7c, the source cache has to update main memory, wait for acknowledgment, and then send the data back to the destination cache. The step to update main memory is critical since the destination cache may, at some point, evict that line and request it back through a normal cache miss. Waiting for the acknowledgment is also important since packets may travel out-of-order in the on-chip network. When a Fetch-O request finds a cache-line dirty in the cache, Figure 3.7d, then a data response with the data marked dirty is sent to the destination cache and the dirty-bit is cleared. The

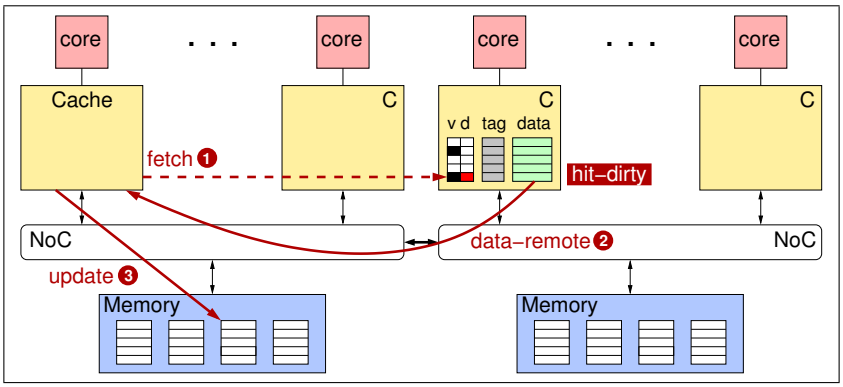
| **Multiple Chips without Directly Accessible Shared Memory** | | | | |
|---|---|---|---|---|
| Operation | Source Cache lookup result | | | Actions |
| | miss | hit-clean | hit-dirty | |
| Fetch | • | | | 1) cache forwards to memory<br>2) memory sends data marked remote<br>3) dst. cache updates memory |
| Fetch | | • | | 1) cache sends data marked remote<br>2) dst. cache updates memory |
| Fetch | | | • | 1) cache sends data marked remote<br>2) dst. cache updates memory |
| Fetch-O | • | | | 1) cache forwards to memory<br>2) memory sends data marked dirty |
| Fetch-O | | • | | 1) cache sends data marked dirty |
| Fetch-O | | | • | 1) cache clears dirty-bit<br>2) cache sends data marked dirty |

Table 3.2: Actions required for software-guided coherence on multiple chips that do not provide all cores with direct access to the total shared memory.

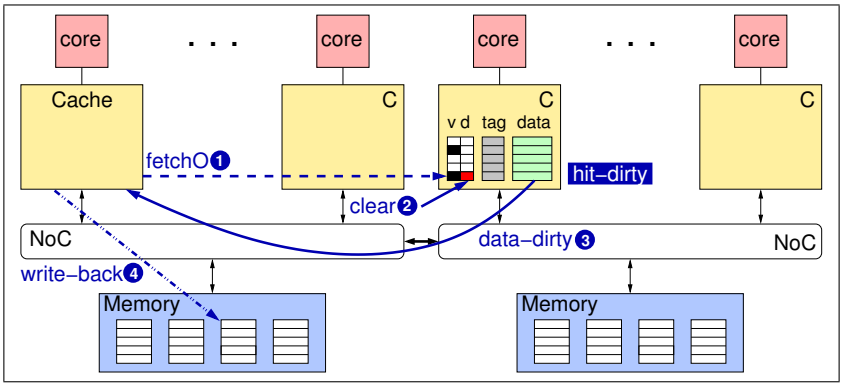
destination cache sets the line as dirty, since it is the new owner, and is responsible to write-back upon eviction at any point in time.

### Support for Multiple Chips

Our primitives for software-guided coherence are scalable and offer support for systems that extend beyond a single chip. We allow the use of these primitives across multiple chips with distributed, physically separate memories. An important requirement for the exploitation of a multi-chip distributed memory system is the existence of a Global Virtual Address Space, *(a)* so that software can refer to any and all objects in the entire system using a single, global identifier (pointer) – its global (virtual) address, and *(b)* communication primitives can operate within this global space. We assume that the runtime and OS establish a Global Virtual Address Space which is visible to all cores in the system (even across chips) and that all tasks belonging to an application operate within this address space. Task arguments should maintain their global virtual addresses throughout the system in order to preserve pointer-based data structures intact and enable dependency resolution without additional overheads. However, the virtual-to-physical address mappings on each chip should be a local decision, internal to each chip, and is




(a)




(b)

Figure 3.8: Transfers between caches on multiple chips under software-guided coherence: (a) Fetch and (b) Fetch-O operations that hit on a dirty cache-line

allowed to differ among chips.

We use the same transfer primitives as in the case of a single chip and list the specific cases in Table 3.2. We describe below how these transfer primitives operate when the main memories of the communicating cores/caches are physically separate and direct access between them is not provided.

The behavior of the Fetch operation is almost independent of the cache-line state (miss, clean, dirty) in the remote cache. In all these cases, data responses to caches outside the same chip (memory territory) are marked as “remote” and the receiver cache should update its local main memory in order to be able to access it through normal cache misses in the future; the receiver should use the local virtual-to-physical mappings for the update. A representative case where a Fetch operation hits on a dirty cache-line is presented in Figure 3.8a.

The Fetch-O operation across caches with separate main memories is also independent of the cache-line state (miss, clean, dirty) in the remote cache. Given the request for ownership (intention to write), all data responses to caches outside the same chip (memory territory) are sent as "dirty". The latter choice allows the receiver cache to keep these lines dirty and update the local main memory only upon evictions; the receiver should use the local virtual-to-physical mappings for the update. When a Fetch-O operation finds a cache-line dirty in the remote node, it clears the dirty bit, as shown in Figure 3.8b.

### Hardware Primitives

This section presents the details of EBP-NC and the associated hardware primitives, i.e. counters and mailboxes, required to support communication in systems with non-coherent caches. An overview of EBP-NC is presented in Figure 3.9.

***Extensions to Explicit Bulk Prefetcher***

We extend the EBP register set and define the EBP-NC interface with the following memory-mapped registers to support systems with non-coherent caches:

- *Source Address:* The starting virtual address that will be used by the source core.
- *Source Core:* The source core number that will serve the operation.
- *Destination Address:* The destination virtual address that will be used for the destination core.
- *Destination Core:* The destination core number that will receive the data.
- *Acknowledgment Counter:* The acknowledgment counter that will track the completion of the operation.
- *Acknowledgment Core:* The core that will receive the acknowledgment for the completion of the operation.
- *Block Size:* The size (bytes) of each block (as in EBP).
- *Block Number:* The number of blocks to prefetch (as in EBP).
- *Block Stride:* A constant stride (measured in bytes) used for the calculation of the next block address (as in EBP).

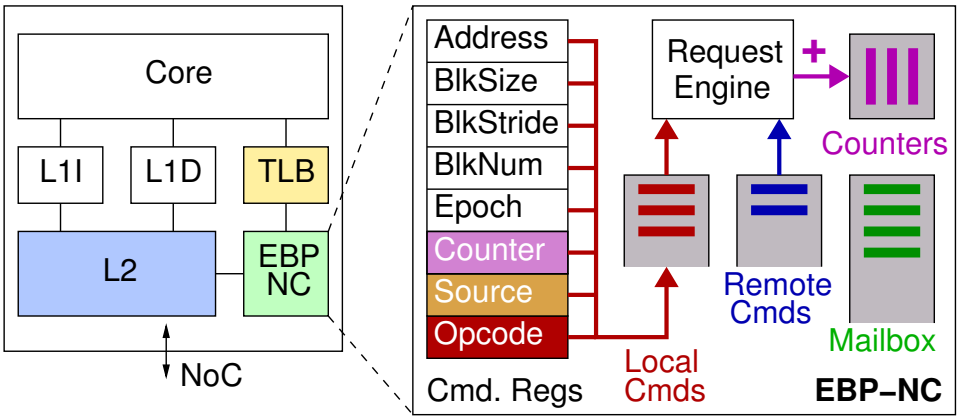



Figure 3.9: Hardware extensions of the Explicit Bulk Prefetcher to support systems with non-coherent caches (EBP-NC).

- *Epoch:* This field is used by ECM (as in EBP).
- *Opcode:* Three opcodes are defined: *(i) Fetch*, *(ii) Fetch-with-Ownership*, and *(iii) Message*. This field marks the access mode of the data and effectively controls the actions in the source cache (as described in Section 3.2.4). The "Message" opcode transfers one cache-line from the source address to the "Mailbox" of the destination core.

The EBP-NC engine requires software to specify the cores that participate in each transfer (core numbers), so that transfers are directed and serviced from the memory hierarchy (caches) of the last writer/producer that modified the data. Effectively, we request the last writer (source core) to provide its coherent view of the data. The destination core is typically the core that initiates the operation and requests the data. However, instead of assuming the destination core to be the initiator, we provide the flexibility for a "third-party" core to initiate commands between other cores, e.g. the task-based runtime executing on a specific core may arrange transfers between any pair of cores in order to maintain coherence.

When the source core number differs from the local core number, then the engine forwards the command descriptor to the appropriate remote EBP-NC engine, otherwise, it operates as in EBP. The engine uses a separate command queue for remote operations and supports up-to 64 remote commands. The arbitration between the "local command queue" and the "remote command queue" follows a round-robin policy. Any number of cores may issue remote commands at any point in time, so the engine has to handle the case of overflows in the finite queue

resources (64 entries). Instead of enforcing any resource allocation policies for the remote commands – or require any software synchronization – we choose to handle such scenarios with negative acknowledgments to counters (explained later in this section).

Moreover, instead of having a single address for each command, we choose to have separate source and destination addresses in order to offer more flexibility to the software and allow it to offload memory copies. Both addresses are virtual addresses and the translation to physical is performed at the corresponding core. In addition, the engine permits arbitrary byte alignments between the source and the destination addresses by following a destination alignment policy, i.e. generates cache-line aligned responses using the destination address. In the context of task-based dataflow programming models, copies can be exploited to implement memory renaming for task arguments. Renaming task arguments, i.e. changing their memory addresses, is equivalent to register-renaming in out-of-order processors and allows the runtime to eliminate output dependencies (WAW) and anti-dependencies (WAR) between tasks. Memory renaming can avoid unnecessary serialization due to memory reuse and has the potential to expose more task parallelism.

***Counters***

Our scheme for coherence among tasks in non-cache-coherent environments requires that all EBP-NC commands associated with a task have completed before that task becomes eligible for execution. To address this issue, we use some special software-programmable hardware counters and add an "acknowledgment counter" field in the EBP-NC command interface. Our design supports up-to 32 signed counters (32-bit) and offers a memory-mapped interface to allow software access with plain load/store instructions. Each EBP-NC command can be associated with an acknowledgment counter and every response data packet arriving in the destination cache contains the acknowledgment counter number. Upon writing the data to the cache, the destination engine atomically increments the associated counter with the payload size (bytes) of the packet. The software may check the current counter value and based of the total transfer size of each command ($block\_size \times block\_number$), it can decide when a command has completed. Our design assumes that the underlying network does not generate duplicate packets.

However, tracking and checking counters per task argument entails significant software overhead and under certain circumstances the software may exhaust this

finite resource (32 counters in our design). To minimize software overhead and counter usage, we permit multiple commands to use the same hardware counter with the intention to collect the aggregate volume of task arguments that belong to a task, thus the software may use and check only one counter. A counter may be initialized to zero, prior to the EBP-NC command, in order to wait for the appropriate value, or initialized to the negative value of the expected transfer size (the counters are signed) in order to wait for zero.

To further optimize waiting on a counter, each counter offers a "blocking" access mode using an alternate address. Loads to the alternate address return only when the associated counter value becomes zero, or when a negative acknowledgment arrives; negative acknowledgments indicate failure to access a remote resource. The blocking mode can substitute polling on a counter, with the potential to save energy when the software waits for transfer completion. Each counter also features a status register where the software may check, in non-blocking mode, if the counter triggered zero or an error occurred, i.e. a negative acknowledgment has been received.

The EBP-NC interface defines an "acknowledgment core" field to indicate which core will receive acknowledgment to the specified counter (acknowledgment counter) when a packet is delivered; this information is contained in each response packet. If a packet is delivered to a core different from the acknowledgment core, then a special "ack" packet is sent to update a remote counter with the packet payload size. This extra field offers the flexibility for a remote core to be notified about packet deliveries. This feature is useful when a "third-party" core arranges transfers between remote cores.

***Mailboxes***

Communication between cores in non-cache-coherent environments requires setting up communication buffers, i.e. memory addresses known in advance. A sender core may exploit the EBP-NC primitives to perform cache-to-cache transfers to the receiver core. However, the software running at a receiver should regularly check the communication buffers for new data. Any core may communicate with any other core in the system, so every receiver should allocate per-sender communication buffers and check all of them regularly. To avoid the memory overhead of communication buffers and the software overhead of polling, we use "Mailboxes".

Mailboxes are small memory-mapped hardware FIFO buffers, 4 KBytes in our design, that allow low-volume communication between cores. Every core has a

local mailbox that receives *atomically* incoming data from the network. The software can utilize the EBP-NC "Message" opcode to directly transfer data from the local cache to a remote mailbox. The "Message" command in EBP-NC fetches one cache-line (64-bytes), from the local cache using the source address and transmits it to the destination core's mailbox through the network. The network packets carrying messages to mailboxes have a special opcode.

Any number of cores may send data to remote mailboxes at any point in time, so the case of overflows in the finite mailbox buffer (4 KBytes - 64 cache-lines) has to be handled. Instead of enforcing any resource allocation policies for the mailboxes – or require any software synchronization – we use negative acknowledgments to counters, allowing software to retry.

Each mailbox is memory-mapped to the local core and features a control register to indicate the current size of the buffer. The software can check for new data in the mailbox by checking the current size on the control register. Mailbox data can only be accessed in FIFO order (dequeue) using a single address. Every load operation dequeues 8-bytes from the FIFO. The format of the data arriving to the mailbox is defined by the software. The task-based runtime running on a master core can utilize mailboxes to dispatch task descriptors to workers, and the worker cores can signal task completion to a master. Mailboxes optimize waiting for new data (e.g. task descriptors) featuring "blocking" access when the mailbox is empty. The blocking mode can substitute polling to the control register, with the potential to save energy when the software waits new data (e.g. workers waiting for tasks).

### 3.2.5 Software Use

This section sketches the use of our proposed hardware primitives in task-based runtime software. We assume that a typical task-based runtime consists of a "Task Manager" that allows applications to submit tasks in the form of task descriptors. Task descriptors contain at least: *(i)* a pointer to the task code and *(ii)* the argument list for the task. Each task argument contains: *(i)* a memory pointer (or a scalar value), *(ii)* argument size (memory range), and *(iii)* argument type (input, output, inout). The task manager uses the task arguments' address ranges to detect dependencies and schedule the independent tasks. Scheduling usually involves putting tasks in intermediate "ready queues" where worker threads can dequeue and execute tasks. When workers complete tasks, they notify the task manager so that dependent tasks satisfy their dependencies and eventually become eligible for execution.

```
1  TaskManager * task_manager;
2  TaskQueue * task_queue;
3  int worker_id;
4  bool active;
5
6  void worker_loop() {
7    Task * task = NULL;
8    // basic worker loop
9    while( active ) {
10     task = fetch();
11     if ( task != NULL ) {
12       execute(task);
13       release(task);
14     }
15   }
16 }
17
18 Task * fetch() {
19   Task * new_task = NULL;
20   // fetch a new task
21   if ( ! task_queue->empty() )
22     new_task = task_queue->dequeue();
23   return new_task;
24 }
25
26 void execute(Task * task) {
27   // task execution
28   task->function_pointer( task->args );
29 }
30
31 void release(Task * task) {
32   // notify the runtime about the task completion
33   task_manager->completed_task(worker_id, task);
34 }
```

Figure 3.10: Skeleton code for worker threads in a task-based runtime.

Figure 3.10 presents a rudimentary skeleton code in C++ for the part of the task-based runtime that is used by the worker threads. The workers operate on a standard loop (worker_loop) and perform the following steps:

1. *fetch()*: fetch a task from the task queue,

2. *execute()*: execute the task using the function pointer and the arguments,

3. *release()*: notify the task manager about the completion of a task.

### Double-buffering Tasks

The execution of tasks by the workers is one of the most performance critical parts of the runtime and the application itself. We propose EBP (Section 3.2.2) in order to allow the workers to prefetch task data and avoid as many cache misses as possible during task execution. The runtime can implement double-buffering of tasks, i.e. initiate prefetching for the next task data as early as possible, before the current task execution. This would provide EBP with (hopefully) adequate time to complete data transfers before the current task completes, and hide the associated transfer latency.

Based on the skeleton code shown in Figure 3.10, we present an implementation of double-buffering in Figure 3.11. We provide new implementations for the functions *fetch()* and *execute()*. At first, during the fetch step, the runtime attempts to dequeue two tasks from the task queue (the current and next task), if it succeeds to do so (enough tasks in the queue), the next time it will dequeue just one task (the next). At steady state, the worker will always have the next task in the "pipeline" and can initiate prefetching for it.

Prefetching is performed before task execution in the execute step. At first, the worker will prefetch data for two tasks and the next time, at steady state, it will prefetch just one (the next). The prefetching step involves iterating through all task arguments, and issuing an EBP command (EBP_PREFETCH) for each of them. The EBP_PREFETCH function uses the address, the size, and the type found in each task argument to initiate an EBP command through the memory-mapped registers. Command initiation requires 5 store instructions to the EBP registers.

The software overhead for double-buffering is mostly commensurate to the number of task arguments. Assuming a modest number of task arguments (e.g. 5), the overhead to initiate prefetch commands is in the order of tens of clock-cycles. Such a number of clock cycles is directly comparable to a long latency cache miss served by off-chip DRAM. However, employing this technique has the potential to save many cache misses as presented in our evaluation in Chapter 5.

### Cache Management of Tasks' Data

Applying double- or multi- buffering of tasks' arguments incurs the danger of cache pollution and interference between the tasks' data in the cache. We propose ECM (Section 3.2.3) to make the cache aware about task contexts and the lifetimes of their associated data. With ECM, the software can allocate local cache resources

```
1  Task * next_task = NULL;
2  bool curr_task_prefetched = false;
3  bool next_task_prefetched = false;
4
5  Task * fetch() {
6    Task * curr_task = NULL;
7    if ( next_task == NULL ) {
8      if ( ! task_queue->empty() )  // fetch the current task
9        curr_task = task_queue->dequeue();
10     if ( ! task_queue->empty() )  // fetch the next task
11       next_task = task_queue->dequeue();
12
13     curr_task_prefetched = false;
14     next_task_prefetched = false;
15   }
16   else {
17     curr_task = next_task;        // next task becomes current
18
19     if ( ! task_queue->empty() )  // fetch new next task
20       next_task = task_queue->dequeue();
21     else
22       next_task = NULL;
23
24     curr_task_prefetched = next_task_prefetched;
25     next_task_prefetched = false;
26   }
27
28   return curr_task;
29 }
30
31 void execute(Task * task) {
32   // prefetch task arguments for current and next task
33   if ( ! curr_task_prefetched ) {
34     for( int i=0 ; i < task->args_num ; i++ )
35       EBP_PREFETCH( task->args[i] );
36   }
37   if ( next_task != NULL ) {
38     for( int i=0 ; i < next_task->args_num ; i++ )
39       EBP_PREFETCH( next_task->args[i] );
40     next_task_prefetched = true;
41   }
42
43   // finally task execution
44   task->function_pointer( task->args );
45 }
```

Figure 3.11: Skeleton code for double-buffering tasks.

```
1  void execute(Task * task) {
2    // advance to new epoch
3    ECM_EPOCH_ADVANCE();
4
5    // prefetch task arguments for current and next task
6    if ( ! curr_task_prefetched ) {
7      // set current epoch quota
8      ECM_EPOCH_QUOTA( task->args_size , ECM_EPOCH_CURRENT );
9      for( int i=0 ; i < task->args_num ; i++ )
10       EBP_PREFETCH_EPOCH( task->args[i] , ECM_EPOCH_CURRENT );
11   }
12   if ( next_task != NULL ) {
13     // set next epoch quota
14     ECM_EPOCH_QUOTA( next_task->args_size , ECM_EPOCH_NEXT );
15     for( int i=0 ; i < next_task->args_num ; i++ )
16       EBP_PREFETCH_EPOCH( next_task->args[i] , ECM_EPOCH_NEXT );
17     next_task_prefetched = true;
18   }
19
20   // finally task execution
21   task->function_pointer( task->args );
22 }
```

Figure 3.12: Skeleton code for double-buffering tasks with epochs.

to tasks, guide the replacement policy, and isolate the effects of prefetching.

Based on the skeleton code of Figure 3.10 and the additions of Figure 3.11, we illustrate in Figure 3.12 the use of ECM. We provide a new implementation of the *execute()* function that uses the ECM features. Before the actual task execution, the code advances the local hardware epoch (ECM_EPOCH_ADVANCE). This simple step signals a new task lifetime and effectively indicates that the data belonging to older tasks (marked in the cache with an older than current epoch) are not critical anymore and are candidates for replacement.

Afterwards, the runtime allocates space (declares quotas) for the tasks that is going to prefetch and indicates the associated epoch (ECM_EPOCH_QUOTA). This piece of code assumes a field in every task descriptor that contains the total size of task arguments (args_size); this field is calculated by the "Task Manager" during the internal dependence analysis step when it iterates the task arguments to find task dependencies. The step to issue EBP commands for each task argument is almost identical to before and the only difference is the epoch field (EBP_PREFETCH_EPOCH); commands now require an extra store instruction per argument. The epoch field indicates whether an EBP command uses the

current or the next epoch for its data and thus, adheres to the associated quota. Prefetch commands now require 6 store instructions compared to 5 when epochs are not used. It becomes apparent from the description above and the code excerpt, that the software use of ECM features incurs a small additional overhead on top of double-buffering. The latter minimal changes in the runtime code that exploit ECM, offer significant performance improvement as it appears from our evaluation in Chapter 5.

# 4

# Experimental Methodology

This chapter describes the experimental methodology we follow to evaluate the architectural support we propose in Chapter 3. The hardware primitives are modeled in architectural simulators that support manycore systems and allow us to compare different system configurations. Section 4.1 presents the simulated system for the hybrid cache/scratchpad memory hierarchy and the benchmarks we implement. Section 4.2 describes the simulation infrastructure, the runtime software, and the benchmarks we use to evaluate the hardware primitives for cache-based memory hierarchies.

## 4.1 Hybrid Cache/Scratchpad Memory Hierarchy

This section describes the experimental methodology we follow to evaluate the hybrid cache/scratchpad memory and the explicit communication hardware primitives we presented in Section 3.1. The hybrid cache/scratchpad memory and the associated hardware primitives are modeled in an architectural simulator, as described in Section 4.1.1. Moreover, we use some popular shared-memory benchmarks that exhibit diverse communication patterns and port them to use our proposed architectural support, Section 4.1.2.

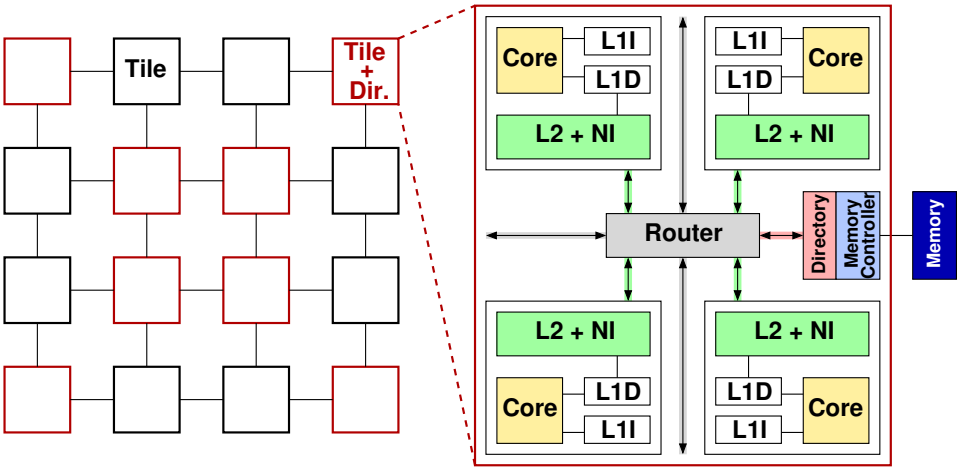



Figure 4.1: A configuration with 64 cores using 4-core tiles connected in a 2D concentrated mesh topology. Each core is coupled with private L1 and L2 caches and our merged cache controller/network interface (NI).

### 4.1.1 Simulator

We use SIMICS 3.0 [80] for full-system simulation and GEMS [81] to implement the timing model for caches, scratchpads, coherence, and RDMA controllers. For accurate NoC modeling we use GARNET [82] and measure NoC energy and power with ORION 2.0 [83].

Cache-coherence is based on a GEMS MOESI directory-based protocol with distributed directories. The system is configured in 4-core tiles where each processor core is coupled with private L1 and L2 caches. Processor tiles are connected to a single NoC node, forming a 2D concentrated mesh topology. The distributed directories, one per memory channel, are placed in a diagonal-X fashion as proposed in [84], and memory blocks are interleaved across memory channels in cache-block granularity. Figure 4.1 illustrates a configuration with 64-cores. We further augment caches with strided hardware prefetching [85] optimized with tagging [86]. We also add our configurable scratchpad memory and RDMA engines inside the private L2 caches and segment large RDMAs into multiple maximum sized network packets. The configuration parameters of our simulated system are listed in Table 4.1.

| Parameter | Setting |
|---|---|
| Cores | 1, 2, 4, 8, 16, 32 or 64, in-order RISC at 2 GHz |
| L1 I/D Caches | 64KB, 2-way associative, 64-byte block,<br>1 port, 1 clock cycle latency |
| L2 Caches | private 256KB, 16-way associative, 64-byte blocks,<br>2-port, 7 clock cycle latency, 32 MSHR,<br>unified, coherent, non-inclusive |
| Coherence Protocol | MOESI distributed directory, per memory channel,<br>4 virtual networks, 10 clock cycles latency,<br>up-to 32 directory protocol engines, non-blocking |
| Data Prefetcher | PC-based stride and tagged, 512-entry history table<br>prefetching degree: 1, 2, 4 |
| Scratchpad | SW based dynamic allocation in L2,<br>block granularity, L1-cacheable |
| RDMA Controller | SW based dynamic allocation of command buffers in L2,<br>64KB max transfer |
| Remote Stores | two 64-byte coalescing remote store buffers |
| NoC | Concentrated Mesh at 2 GHz, 4 cores per node,<br>16-byte control packets, 80-byte data packets,<br>8-byte links, 1 cycle link traversal, 5-stage router pipeline<br>4 virtual networks, 4 VCs per virtual network |
| DRAM | 1GB off-chip DRAM, up-to 8 memory channels,<br>1 channel per 8 cores, 80ns access time |

Table 4.1: Configuration parameters for full-system simulation of the hybrid cache/scratchpad memory and the explicit communication hardware primitives.

### 4.1.2 Benchmarks

In order to evaluate explicit communication and synchronization and compare them with implicit communication via cache-coherence, we choose four benchmark kernels that exhibit diverse communication patterns:

- **Smith-Waterman:** A widely used bioinformatics algorithm that performs local protein sequence alignment. The anti-diagonal wavefront parallelization of this kernel exhibits a typical *one-to-one* streaming pattern.

- **2D Jacobi:** A five-point stencil code where each node communicates with all of its neighboring nodes exhibiting a *nearest-neighbor* pattern.

- **Bitonic Sort:** A popular sorting kernel where multiple different node pairs exchange data depending on the sorting phase and form the *butterfly* pattern.

- **FFT:** We use the Splash-2 [87] FFT kernel to exercise *all-to-all* communication.

We optimize separately the shared-memory and the explicit communication versions of the benchmarks, in order to make fair comparisons. All shared-memory implementations are carefully optimized with blocking and the shared arrays are padded appropriately to avoid *false sharing*. MCS [88] locks and barriers are used for synchronization. Porting the shared-memory implementations of the benchmarks to use explicit communication, i.e. RDMAs, Remote Stores and Counters, and benefit from direct *scratchpad-to-scratchpad* transfers was not a trivial task and required us to fully understand the data exchange patterns. The explicit communication versions of the benchmarks make minimal use cache-coherence, mainly to exchange scratchpad buffer pointers during the initialization phases.

## 4.2 Cache-based Memory Hierarchy

This section describes the experimental methodology we follow to evaluate the hardware primitives for cache-based memory hierarchies we presented in Section 3.2. The hardware primitives are modeled in an architectural simulator that supports manycore systems with and without cache-coherence. We also co-design and implement task-based runtime software that exploits the new hardware primitives.

Section 4.2.1 presents our custom simulation infrastructure, *FORTHSim*, that builds on top of popular architectural simulators and integrates tools for power estimation. Section 4.2.2 describes the design of *TaskFlow*, a minimal task-based dataflow runtime system that implements the basic task constructs and follows the OmpSs/SMPSs tasking syntax. Moreover, we convert six popular benchmarks to the task-based programming model, Section 4.2.3.

### 4.2.1 Simulation Infrastructure

*FORTHSim* is a cycle-level execution-driven simulator for multi-core architectures written in C++[1]. The purpose of the simulator is to evaluate the HW primitives

[1] We opted to develop our custom simulator, since GEMS [81] does not support the newer versions of SIMICS [80] and the simulation speed for configurations with large core counts was prohibitive. The newer GEM5 [89] simulator that uses M5 [90] instead of SIMICS was in "alpha" mode when we started this work.

proposed for non-cache-coherent systems and compare them against pure hardware cache-coherence under common hardware assumptions.

### FORTHSim overview

The implementation of FORTHSim builds upon the popular GEMS simulator [81] and integrates some well respected tools and models for architectural evaluation such as: the PIN dynamic binary instrumentation tool [91], the Garnet NoC models [82], Orion2 NoC power estimator [83], and the DRAMSim2 multi-channel memory controller [92]. FORTHSim uses Cacti 6.5 [93] to estimate the energy for memory structures such as the caches and coherence directories. The simulator runs unmodified x86 binaries at application level; operating system activity is not simulated. A high level overview of the simulation infrastructure is depicted in Figure 4.2.

The simulator follows an *execute-first* methodology, that is: each instruction is first executed at native hardware speeds and then it is simulated. A per-thread *online trace* from the native execution is generated in memory and is fed to the simulated architecture which models the processor cores, the detailed memory system, the NoC and the DRAM. Given that OS is not simulated, the simulation engine implements some OS functions critical for the simulation, such as: thread-to-core assignment and virtual-to-physical memory mappings.

The simulator is divided into a front-end and a back-end part. The front-end consists of a custom PIN tool that performs dynamic binary instrumentation at instruction-level to capture the instructions executed by the application. Each x86 macro instruction is converted to a number of RISC-like instructions (equivalent to uops), however these instructions are more abstract than those found in typical ISAs. The simulator ISA defines the following major categories of operations: *(i)* integer, *(ii)* floating-point, *(iii)* branch, *(iv)* load, *(v)* store, and *(vi)* atomic. The back-end consists of an independent thread that runs in parallel with the front-end and implements the simulation engine, the processor models, interfaces with GEMS Ruby and controls the simulation event-queue.

The front-end and the back-end operate independently and communicate via a set of per-thread producer-consumer instruction queues (lock-free). Each instrumented application thread in the front-end enqueues all executed instructions, while the core assigned to each thread in the back-end dequeues and models each instruction. The length of the instruction queues, which is configurable, controls the slack of the native execution relative to the simulated execution. An instruction

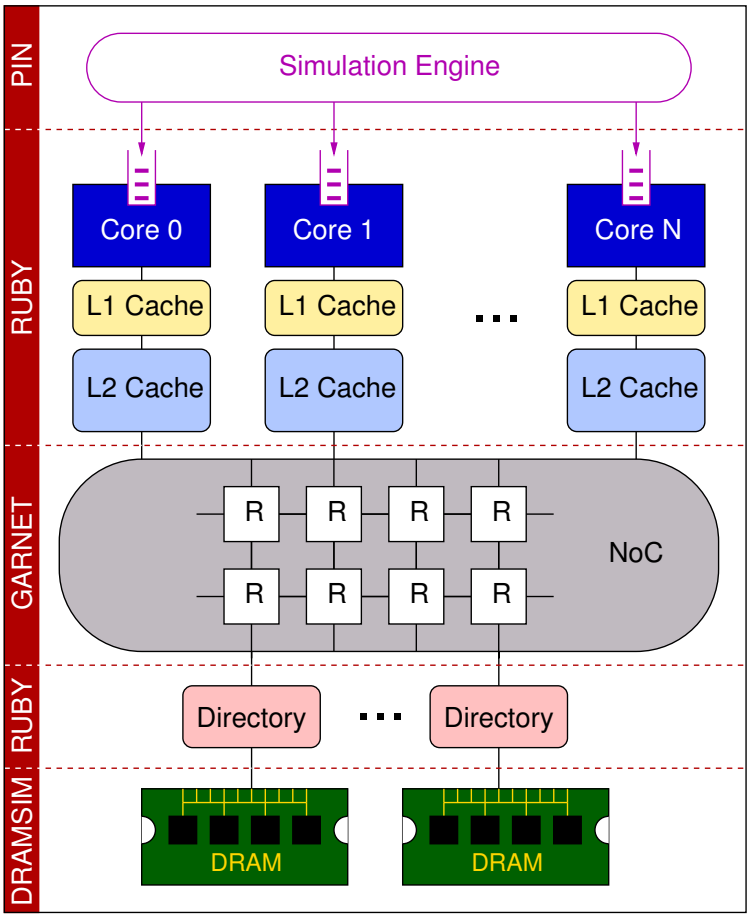



Figure 4.2: FORTHSim high-level overview: the major components appear on the left bar.

queue length of one entry forces the native execution and the simulated execution to run in lock-step; queue lengths of 16-128 entries offer high simulation accuracy while allowing for normal simulation speeds. When an instruction queue becomes full, the native application thread yields the native core to allow for other application threads to execute instructions and fill their instruction queues. The back-end simulation progresses on a cycle-by-cycle basis, when all the active threads have at least one instruction in their respective queue. Effectively, the simulated execution may control how fast the native execution advances in a per-thread granularity and thus affect the actual application behavior. Under certain scenarios, e.g. locks, task-stealing, the flow of the native execution will change to reflect the simulated behavior.

The simulator exploits advanced PIN features and implements function rewriting in order to allow the application that runs natively to communicate and integrate tightly with the simulator. For instance, with function rewriting, the function gettimeofday() returns the simulator system time instead of the native system time.

### FORTHSim components and configuration

The simulator consists of the following major components:

- *Simulation Engine :* A custom PIN tool that collects all application instructions and controls the progress of simulation events in a cycle-by-cycle basis.
- *Ruby Cache and Directory Models:* The Ruby memory system simulator is part of GEMS and allows the definition of detailed memory systems and coherence protocols using a domain specific language called SLICC. Ruby offers tens of configuration parameters, such as: cache organization and sizes, number of cache-levels, coherence protocols etc.
- *Garnet NoC models:* Garnet is a detailed NoC simulator that integrates with Ruby and allows the definition of arbitrary NoC topologies, router architectures, link bandwidths and latencies etc. Moreover, it integrates with the Orion2 NoC power estimator.
- *DRAMSim2 Memory and Controller Models:* DRAMSim2 is a detailed memory controller model that allows for accurate DRAM simulation and power estimation. It supports multi-channel memory controllers, incorporates DRAM timing models for most DDR2 and DDR3 chips, and offers several configuration parameters such as: burst lengths, page management policies, addressing schemes etc.

### Evaluated System

We model a tiled manycore architecture with directory-based cache-coherence and distributed directories (per memory controller). We also model the same architecture without hardware cache-coherence for our software-guided coherence scheme. Our design uses up to 64 in-order cores with a two-level private cache hierarchy, similar to the Intel Xeon-Phi co-processor architecture [94]. We also add an out-of-order superscalar core for the role of the master processor that runs the main

| Parameter | Setting |
|---|---|
| Master Core | out-of-order superscalar at 2 GHz, 4-wide,<br>128-entry ROB, 96-entry LD/ST queue, |
| Worker Cores | up to 64, in-order at 2 GHz |
| L1 Caches | 32KB, 4-way, 64-byte block,<br>1 port, 1-cycle, LRU, split I/D |
| L2 Caches | private 256KB, 8-way, 64-byte block,<br>2-port, 8-cycles, NRU, 16 MSHRs,<br>unified, coherent, inclusive |
| Coherence | MESI directory per memory controller,<br>4 virtual networks, 10-cycles, non-blocking |
| L2 Prefetcher | PC-based stride, 64 streams, degree of 4 |
| NoC | 2D Mesh at 2 GHz, 2 cores per node,<br>16-byte control packets, 80-byte data packets,<br>16-byte links, 1-cycle link, 5-stage routers,<br>4 virtual networks, 4 VCs per virtual network |
| DRAM | 4 dual-channel memory controllers,<br>16GB DDR3 SDRAM, PC3-15400,<br>8 banks, FR-FCFS scheduling policy |
| EBP | per core, 32 commands, 8 outstanding requests |
| ECM | 8 epochs using 3 bits per L2 tag |
| EBP-NC | per core, 32 local commands, 64 remote commands |
| Counters | per core, 32 counters , signed 32-bit values |
| Mailboxes | 4KB per core, 64 cache-lines |

Table 4.2: Detailed architectural parameters for the simulations that evaluate our architectural support for cache-based memory hierarchies.

application thread and spawns tasks. Power estimation is for 32nm technology. Further details for simulation parameters appear in Table 4.2.

For the evaluation of our proposed architectural support we also implement a per-core PC-based stride prefetcher [85] that prefetches cache-lines into the L2 cache. We also model a state-of-the-art prefetch-aware replacement policy [61] for the purpose of comparison with ECM.

### 4.2.2 Task-based Runtime

We implement *TaskFlow*, a runtime system for the task-based dataflow programming model that supports the basic task spawning and waiting constructs as discussed in Section 2.2. It supports two-dimensional address ranges for declaring task footprints and uses a block-based approach with arbitrary granularity for dynamic dependence analysis [38]. Pragmas are converted into calls to the underlying

runtime library using a source-to-source compiler[2] that is based on CIL [95].

The runtime can utilize an arbitrary number of worker threads but only a single master thread can spawn tasks. To issue tasks, the compiler generated code builds task descriptors containing:

*(i)* the function pointer of each task.

*(ii)* the base memory pointer for each argument.

*(iii)* the dimensions of each argument.

*(iv)* the argument type describing the memory side-effects (input, output, inout).

Upon task spawning, the runtime analyzes inter-task dependencies using internal metadata for the application memory and decides whether tasks are eligible for immediate execution. Ready (independent) tasks are scheduled to worker threads, while dependent tasks are kept in internal data structures until all their dependencies are satisfied. Eventually, all tasks become eligible for scheduling. The runtime fully supports out-of-order execution of tasks. The implementation utilizes private per-worker FIFO queues to schedule tasks and follows a *round-robin lowest occupancy first* scheduling policy to achieve load balancing in the presence of unbalanced tasks.

When the runtime executes on architectures that implement EBP and ECM, it utilizes our hardware support to optimize task-based execution. It prefetches task data in a double-buffered fashion using EBP and leverages ECM to manage the local cache resources. The runtime advances ECM epochs on task boundaries and assigns epoch quotas to tasks based on their memory footprint.

For non-cache-coherent systems we implement a slightly modified version of the runtime. In this version, the runtime keeps the history of tasks and the actual core that executed each task, so that it can provide SGC with the location of the last producer/writer of task arguments. Task spawning uses EBP-NC to push the task descriptors to worker caches and notifies them with messages in their mailboxes. The workers notify the master core about task completions by sending messages to its mailbox.

### 4.2.3 Task-based Benchmarks

We use six widely used benchmark applications, after converting them to the task-based dataflow programming model using *C pragmas*. Most benchmarks originate

[2]Foivos Zakkak developed the support in the SCOOP compiler for our runtime API.

from the SMPSs distribution and are sketched in [37]. All benchmarks accept a block size as input parameter to allow controlling the memory footprint of tasks and, in effect, the task granularity. The benchmarks are the following:

- **Matrix Multiplication:** An implementation of dense matrix multiplication using routines from the BLAS [96] package. We run the benchmark with $1000 \times 1000$ double precision matrices.
- **Jacobi:** The iterative 5-point stencil linear equation solver. We run Jacobi with $1000 \times 1000$ double precision matrices.
- **FFT:** The 2D Fast Fourier Transform uses the FFTW [97] package for 1D FFT computations and performs transposition and twiddling in-place. We run FFT with 1M points.
- **Bitonic Sort:** A comparison-based sorting kernel. The implementation uses Quicksort to create an initial bitonic sequence and then performs a logarithmic number of merge phases. We sort 1M long integers.
- **Cholesky:** The Cholesky decomposition of a positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. The implementation uses routines from the BLAS [96] package. We run Cholesky with a $1280 \times 1280$ double precision matrix.
- **Sparse LU:** The LU factorization of a sparse matrix[3] as the product of a lower triangular matrix and an upper triangular matrix. We run Sparse LU with a $1280 \times 1280$ double precision matrix.

[3] Some sub-blocks of the matrix may be null, i.e. contain zeros values.

# 5 Evaluation

This chapter evaluates the architectural support we propose in Chapter 3 using the experimental methodology we present in Chapter 4. We simulate parallel workloads in manycore systems with up to 64 cores and compare our architectural support against directory-based cache-coherence with hardware prefetching. We measure and analyze multiple aspects of our hardware support and examine the impact in performance, memory traffic, and energy consumption. Section 5.1 evaluates our hybrid cache/scratchpad memory hierarchy and the associated explicit communication hardware primitives. Section 5.2 evaluates our architectural support for cache-based memory hierarchies which is co-designed with a task-based dataflow programming system that provides software guidance.

## 5.1 Hybrid Cache/Scratchpad Memory Hierarchy

This section evaluates the hybrid cache/scratchpad memory hierarchy and the explicit communication primitives we propose in Section 3.1 using the experimental methodology we present in Section 4.1. We simulate the execution of popular parallel workloads in manycore systems with up to 64 cores and compare explicit communication against implicit communication via directory-based cache-coherence

with or without hardware prefetching.

Section 5.1.1 presents the performance benefits of explicit communication in a set of popular benchmarks. Section 5.1.2 investigates the impact of our explicit communication architectural support in the on-chip network traffic. Section 5.1.3 measures energy consumption in the on-chip network and shows significant savings in energy and power.

In the evaluation that follows we run all benchmarks with datasets that fit in the on-chip cache/scratchpad memories in order to isolate and quantify the effects of implicit and explicit communication models in on-chip communication. We keep the total application dataset sizes fixed and increase the number of cores in order to study the potential of the communication models for fine-grain on-chip communication patterns.

### 5.1.1 Benefits from Explicit Communication and Synchronization

For each benchmark we run experiments with up to 64 cores to quantify the effects of explicit communication and synchronization in execution time. In Figure 5.1 we present a comparison of the achieved speedups for the following configurations: *(i)* plain hardware managed caches, *(ii)* hardware managed caches with strided hardware prefetching, *(iii)* scratchpad memory with RDMAs, Remote Stores and synchronization primitives, *(iv)* a perfect memory system where every memory access costs a single clock cycle. For the measurements of the prefetching version we experiment with various prefetch degrees and present only the most efficient configuration per benchmark[1].

The parallelism in Smith-Waterman varies from step-to-step since the number of blocks that can be processed in parallel are only those located in the anti-diagonal wavefront. The maximum available parallelism depends on the input sequences length, in our case the peak is 64 and was limited by the on-chip memory size. The graph illustrates that the RDMA version achieves speedups very close to the perfect case, while on the other hand the hardware prefetcher has negative effect on performance and achieves speedup lower than the plain cache configuration. The reason for the bizarre behavior of the prefetcher is that it prefetches data too early, i.e. the producer core has not finished computing the next block, and this leads to a catastrophic sharing pattern; the producer/writer "loses" the exclusive ownership of cache-blocks the time it writes them. On 64 cores, the RDMA

[1]We also used large OS page sizes (up-to 4MB) in order to overcome the prefetcher limitation that does not permit prefetching across page boundaries.

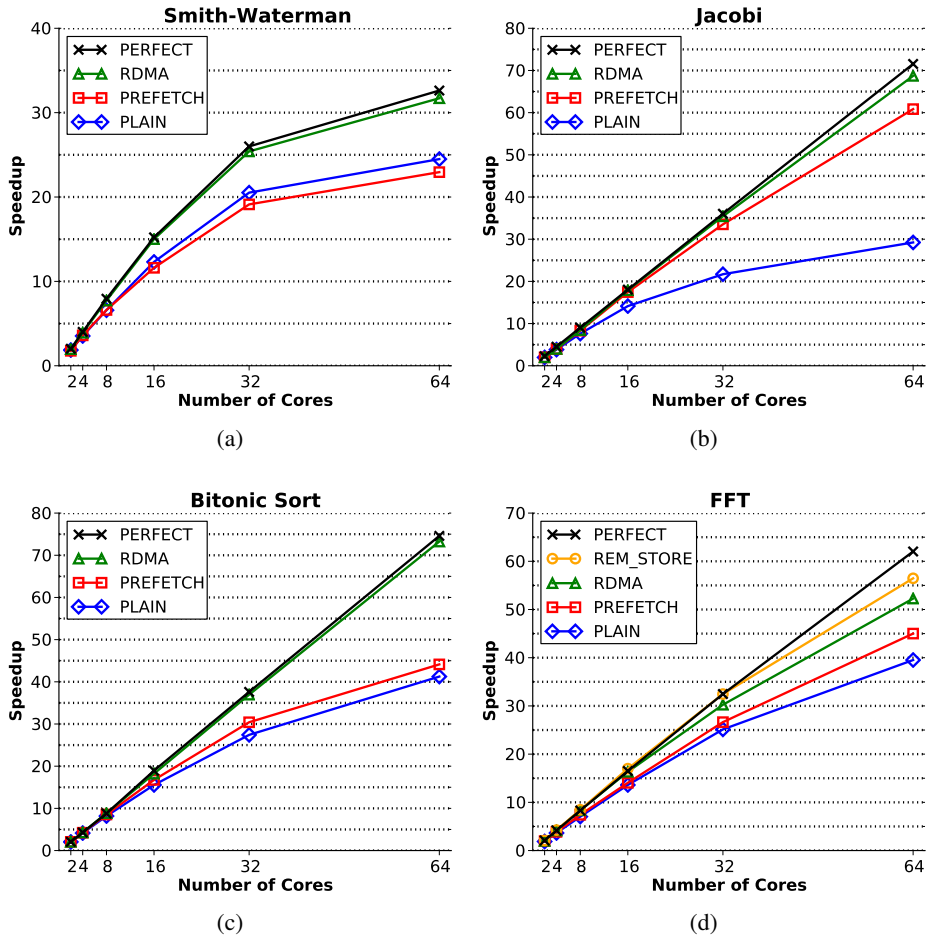



Figure 5.1: Speedup vs core count: Comparing implicit communication with caches and prefetchers versus explicit communication with scratchpad memories and RDMA.

version is *30% and 38% faster* than a plain cache and a cache with prefetcher respectively.

The speedups[2] measured for Jacobi demonstrate that the RDMA version follows closely the perfect case, i.e. the latency of communication can be perfectly hidden since communication happens only between neighboring NoC nodes. For the hardware prefetching configuration, performance declines on more than 32 cores and cannot follow the perfect case. Using 64 cores for the same problem size, parallelism becomes finer, and communication increase offers the RDMA ver-

[2]Superlinear speedups are an effect of the increased total L1 cache size.

sion a larger advantage that reaches *13% faster* execution than the prefetcher-based version. The reason for this behavior is the excess coherence traffic injected by prefetching: although the prefetcher's efficiency is very high – 96% of the prefetched data are actually used – the associated traffic creates contention in the directories and increases the cache miss latencies. Increasing the prefetching degree further in order to hide the additional latency makes things worse as the traffic is further increased. Moreover, the communication is not restricted to neighboring nodes, as the indirection through the distributed directories forces the cache requests to cross more NoC nodes and create additional congestion.

Communication in Bitonic increases with the number of participating processors, thus for small core counts the local sorting phases dominate in the execution time. However, for 64 cores the RDMA version outreaches the prefetcher-based version and results in *40% faster* execution time, ideally following the perfect case. On large core counts communication increases, the pattern changes from step-to-step and the amount of exchanged data becomes finer, so the prefetcher cannot predict and prefetch data accurately in time. Additionally barrier synchronization time increases on many cores and extra communication is required. On the other hand, our completion notification mechanisms, i.e. counters, trigger local notifications when all data are delivered in place by RDMA, thus saving trips through the NoC.

FFT exhibits an *all-to-all* communication pattern, and for large core counts where computation and communication become fine-grain we observe that no version follows very closely the perfect case, however for 64 cores the RDMA version is *16% faster* than the prefetcher-based version. The bottleneck in the RDMA version is the massive initiation of short RDMA transfers that cannot be amortized. Similarly, the startup overhead (learning period) of the prefetcher cannot be amortized. To alleviate the RDMA initiation cost we implement an FFT version with remote stores that incur virtually no cost to communicate with remote nodes. This FFT version is optimized so that the transpose step, the actual communication step, is combined with the FFT computation, through coalesced remote stores. On 64 cores, the remote store version is *9% and 25% faster* than the RDMA and prefetcher-based version respectively.

### 5.1.2 On-chip Traffic

We collect detailed NoC statistics in order to compare the traffic generated by each communication model. We compare the number of control and data packets gen-

erated in each case, and measure the associated transfer volumes in bytes; Figures 5.2 and 5.3 respectively. The bars that present the number of packets and the total NoC volume are normalized to the case of two cores with plain caches in order to gain insight about the increase in communication when many cores collaborate to solve the same problem.

### On-chip Packet Count

The use of explicit communication with RDMA achieves significant reduction in the number of control packets in all benchmarks since it generates negligible control traffic – close to zero (Figure 5.2). There are also additional benefits in the number of data packets and the transferred volume that are further analyzed.

In Smith-Waterman, the RDMA version generates *40% less* data packets, on 64 cores, when compared to plain caches since the shared-memory version of the benchmarks uses an extra dependence array to signal completion of sub-blocks; flag synchronization/polling is used to satisfy the dependences. On the other hand, the RDMA version uses completion counters to trigger local notifications at the receiver, when all inputs are delivered in-place. The version of Smith-Waterman with prefetching has already exhibited the destructive early prefetching pathology and transfers superfluous data; the RDMA version injects *less than one quarter* of the data packets injected in the prefetcher-based version.

In Jacobi and Bitonic, we observe that explicit communication using RDMA generates *about half* the data packets when compared either with plain caches or caches with prefetchers. The reason that caches transmit more data packets here is due to the iterative producer-consumer pattern between processor pairs that reuse the same cache lines and force then to ping-pong between nodes – producer requests exclusive ownership and causes invalidation/forwarding at the consumer.

The RDMA version of the FFT benchmark injects *30% less* data packets: the RDMA version does not need the barrier synchronization required by the shared-memory version, but instead uses RDMA with completion counters to trigger local notifications when all data arrive. The version with remote stores behaves similarly to RDMA for data packets but requires additional control packets to signal the successful delivery of the remote store packets, i.e. remote store acknowledgments.

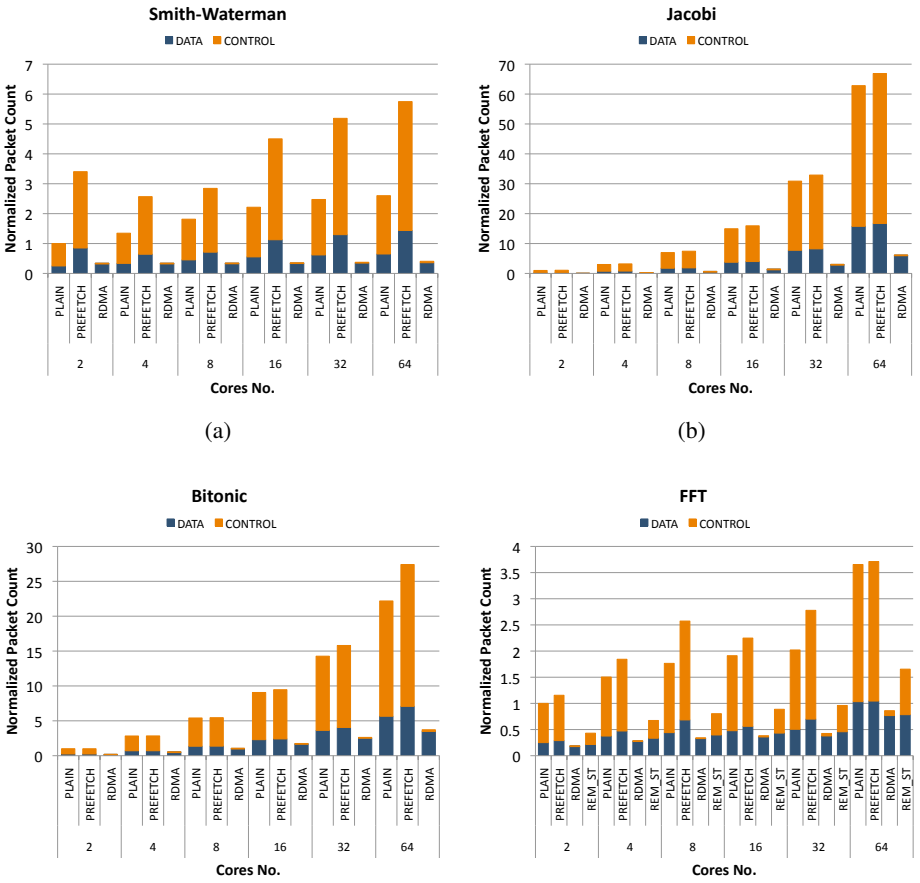



Figure 5.2: NoC packet count, normalized to 2-core plain cache total number of packets.

### On-chip Packet Volume

In terms of total transferred volume through the NoC (Figure 5.3), the RDMA version of Smith-Waterman injects *4 and 10 times less* volume than plain caches and caches with prefetching respectively. The RDMA version of Jacobi reduces NoC volume by a *factor of 4* when compared with the volume transferred by caches. Similarly, the RDMA version of Bitonic generates *2.5 and 3 times less* NoC volume than plain caches and caches with prefetching respectively. For the FFT benchmark, the RDMA and Remote Store versions transfer *2.8 and 1.8 times less* volume than caches.

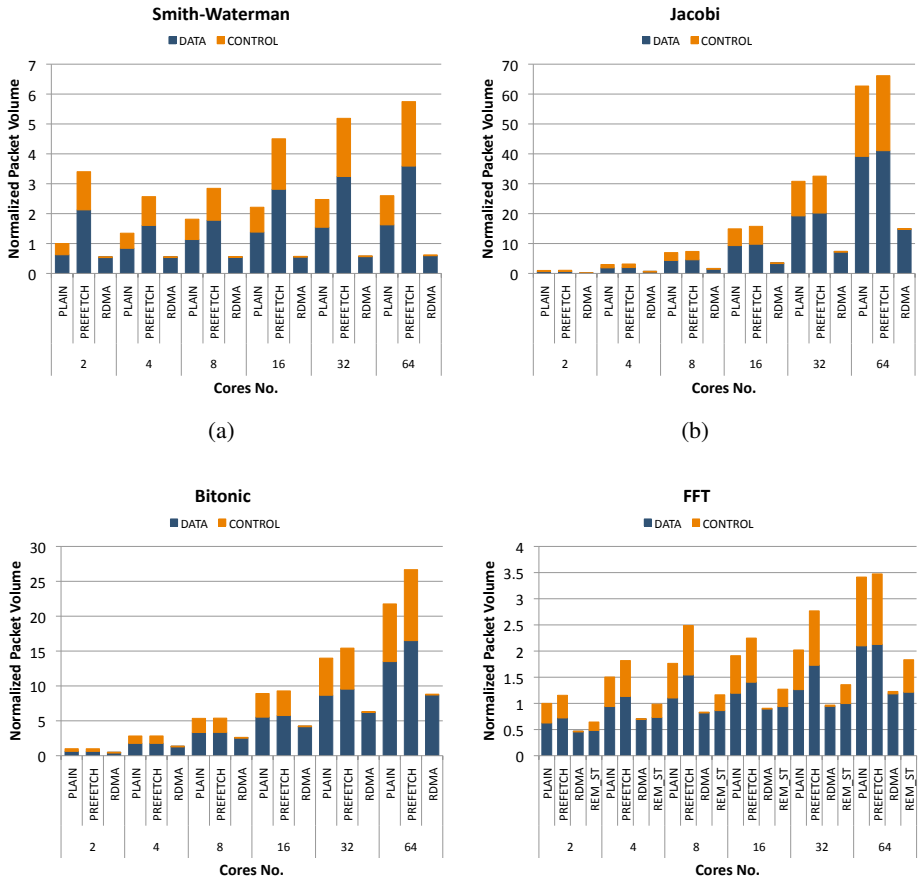



Figure 5.3: NoC packet volume (bytes), normalized to 2-core plain cache total volume.

### 5.1.3 Network Energy, Energy-Delay, and Power

In order to evaluate and compare the impact of explicit communication mechanisms in NoC power, we carefully collect detailed statistics inside the GARNET [82] NoC models and feed the ORION 2.0 [83] NoC power estimation tool. We compare the behavior of each communication mechanism using three metrics: *(i)* Energy consumption, *(ii)* Energy Delay Product (EDP), *(iii)* Power consumption. The results are presented in Figures 5.4, 5.5 and 5.6. The bars are normalized to the case of plain caches in each configuration, i.e. 2, 4, 8, 16, 32 and 64 cores, and we cannot compare energy and power among different configurations since the hardware resources (number of NoC router and links) are different.

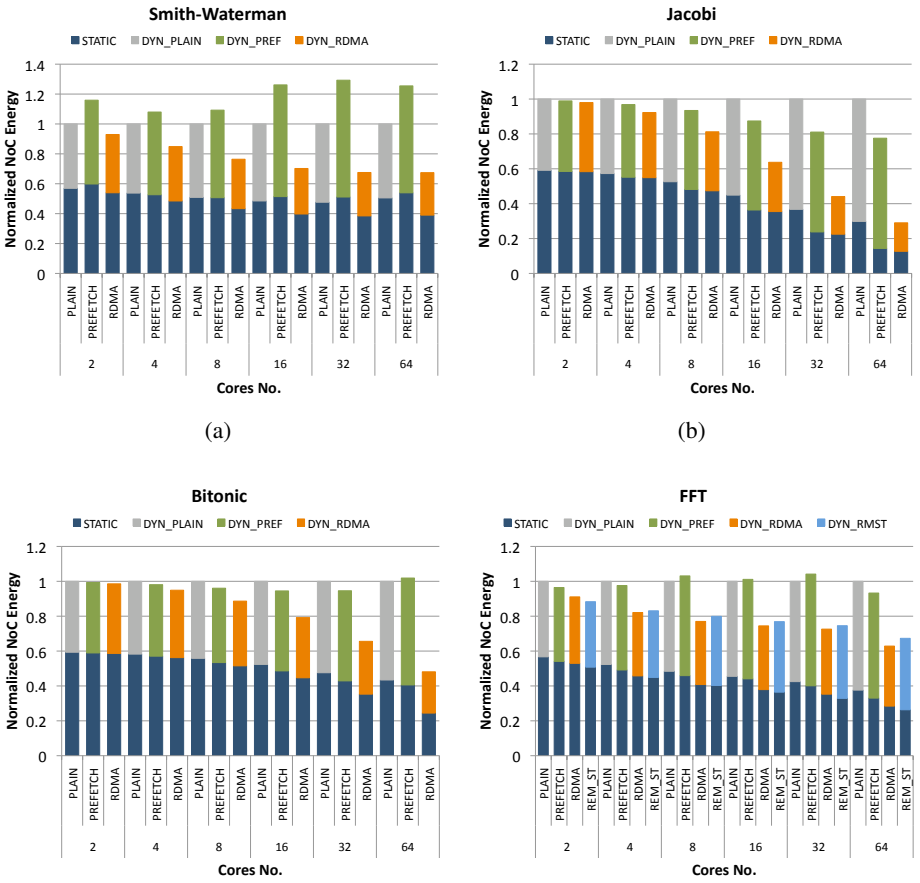



Figure 5.4: NoC Energy, normalized to plain cache energy consumption.

## NoC Energy

As far as the NoC energy is concerned, the results illustrate that for large core counts, explicit communication achieves significant energy reduction on 64 cores that ranges from 35% to 70% when compared to caches with or without prefetching, Figure 5.4. The communication intensity of each application greatly influences the dynamic energy consumption, however explicit communication not only is beneficial in all cases but also offers additional NoC energy benefits when the applications have high communication demands, e.g. Jacobi.

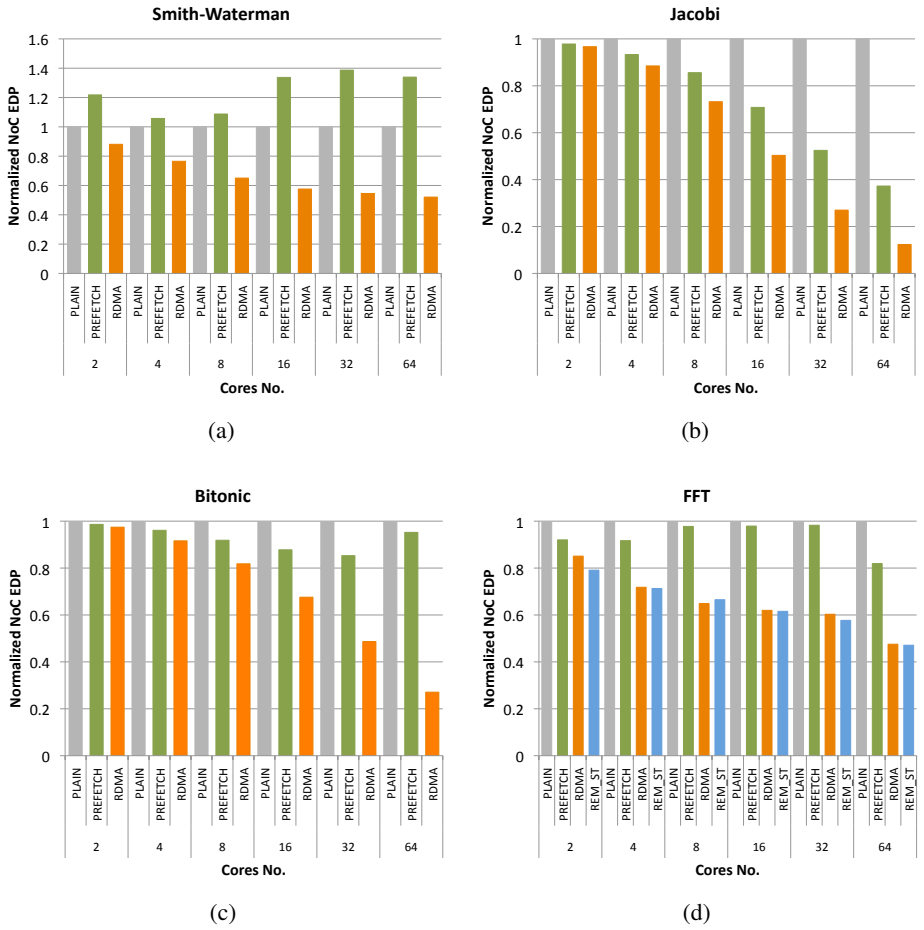



Figure 5.5: NoC Energy Delay Product (EDP), normalized to plain cache EDP.

### NoC Energy-Delay Product

Energy-delay product (EDP) is proposed as useful metric that offers equal "weight" to either energy or performance; lower EDP values are always desirable. Our evaluation demonstrates that explicit communication achieves both lower execution time and lower NoC energy consumption, therefore the EDP metric shows a reduction of 50% to 90% when compared to plain caches and a reduction of 40% to 70% when compared to caches with prefetching, Figure 5.5.

### NoC Power

Analyzing the power consumption in Figure 5.6, shows that the lower energy consumption and EDP of explicit communication is not only an effect of shorter ex-

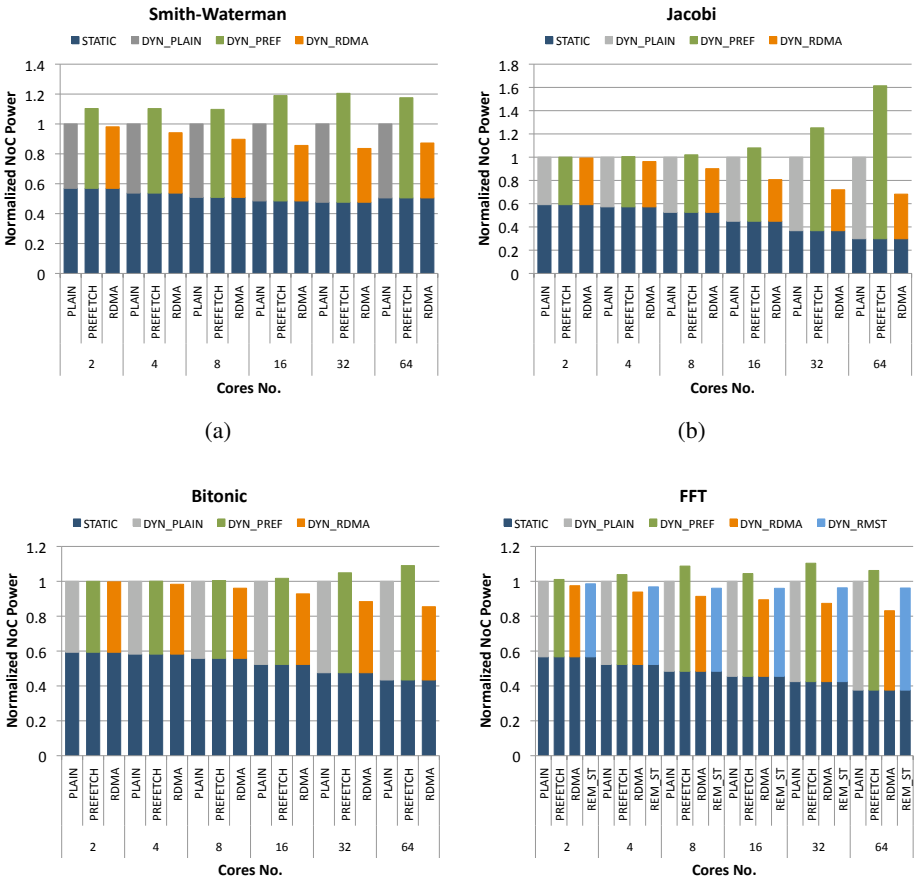



Figure 5.6: NoC Power, normalized to plain cache power consumption.

ecution time, as it appears for the prefetcher version, but also due to the fact that it generates less NoC packets. The latter effect offers a reduction in the total NoC power that ranges from *10% to 30%* when compared with plain caches and *10% to 50%* when compared with caches with prefetching. Prefetching results in increased NoC power consumption when compared to plain caches.

## 5.2 Cache-based Memory Hierarchy

This section evaluates the architectural support for cache-based memory hierarchies we propose in Section 3.2 using the experimental methodology we present in Section 4.2. We simulate the execution of task-based workloads in manycore systems with up to 64 cores and memory hierarchies with and without cache-coherence. We measure and analyze multiple aspects of our hardware support and examine the impact in performance, memory traffic and energy consumption.

Section 5.2.1 presents the performance of task-based benchmarks using the proposed software-directed hardware primitives: *(i)* Explicit Bulk Prefetcher (EBP), *(ii)* Epoch-based Cache Management (ECM) and *(iii)* Software Guided Coherence (SGC), and compares them against a hardware-only prefetcher. Section 5.2.2 analyzes the impact of our hardware support in the on-chip network traffic and the off-chip memory traffic. Section 5.2.3 measures the energy consumption of our hardware primitives and illustrates significant reduction in energy and power. Finally, Section 5.2.4 compares the Epoch-based Cache Management scheme with several cache replacement policies.

### 5.2.1 Performance Analysis

We evaluate the performance of our proposed hardware primitives (EBP and ECM) in a cache-coherent system by contrasting them to a baseline without and with HW-only prefetching. Moreover, we evaluate our architectural support for software-guided coherence (SGC) in an identical system with non-coherent caches and make direct comparisons with the cache-coherent system. SGC implements our proposed hardware support for non-cache-coherent systems (EBP-NC,Counters,Mailboxes).

We run the benchmarks with block sizes that generate fine-grain tasks, each with a memory footprint size approximately equal to the L1 cache size (32KB). Figure 5.7 and Figure 5.8 present the percentage of performance improvement (execution time) over the baseline without prefetching for runs with up to 64 worker cores, for four configurations: *(i)* hardware L2 prefetcher (HWP), *(ii)* EBP, *(iii)* EBP with ECM, and *(iv)* SGC with ECM. All measurements include the software runtime overhead. We also plot the speedup achieved by the baseline without prefetching, normalized to single core serial execution without runtime overhead (ignore pragmas), to show how each benchmark scales in our implementation.

We observe that the configuration with EBP in conjunction with ECM always outperforms HWP. For the benchmarks that scale almost linearly (Matrix multipli-

cation, Jacobi, FFT, and Bitonic) the improvement over HWP for the higher core counts (>=16) ranges from 16% in FFT to 43% in Jacobi, which is the most memory intensive benchmark. Bitonic is 26% faster in 32 cores but appears 10% faster in 64 cores because the master thread saturates and cannot generate enough tasks to feed the workers and allow for tasks double-buffering. Matrix multiplication is consistently better by more than 18%. For benchmarks that do not scale linearly and are more compute intensive (Cholesky, Sparse LU) the improvement ranges from 3% in Sparse LU up to 22% in Cholesky. EBP alone (without ECM) is not always better than HWP. In Jacobi it performs worse than HWP by more than 13% and in FFT it achieves almost the same performance. In the rest of the benchmarks, EBP alone performs either worse than EBP+ECM (8% in Matrix multiplication) or achieves the almost same performance (Bitonic, Cholesky, Sparse LU). On average, for the 64-core configuration, EBP+ECM is *20% faster* than the HW-only prefetcher and *15% faster* than EBP alone.

The non-cache-coherent system with the SGC hardware primitives and ECM (SGC+ECM) introduces small performance degradation when compared to EBP with ECM in the cache-coherent system. The most important reason for this SGC behavior is the requirement to wait until all transfers have completed before tasks start executing. The workers cannot hide the transfer latency when they do not have enough tasks to apply double-buffering. The latter situation appears with "fork-join" task patterns and at the beginning of every application. Moreover, the runtime version for non-cache-coherent systems has more work to perform since it needs to analyze the task arguments and infer the last writer/producer. This extra operation may, under specific circumstances, saturate the master core faster than the standard runtime version and reduce the rate of task spawning. A representative example of faster saturation of the master core is the case of the FFT benchmark. In FFT, the tasks implementing the transpose steps face fragmentation of their task arguments, i.e. internal pieces of an argument have been written by different tasks that run in different cores. In the latter case, the runtime has to provide multiple different last writers for sub-ranges of a task argument.

For the higher core counts (>=16) the performance of FFT with SGC+ECM is from 4% to 18% slower than with EBP+ECM. Jacobi runs up to 9% slower in 64 cores. Bitonic is up to 7% slower and Matrix Multiplication is up to 5% slower. The performance of SGC+ECM in Cholesky and Sparse LU is almost similar to that of EBP+ECM. On average for the 64-core configuration, the performance of SGC+ECM is 6% lower than EBP+ECM. For the 16-core and 32-core setups, the

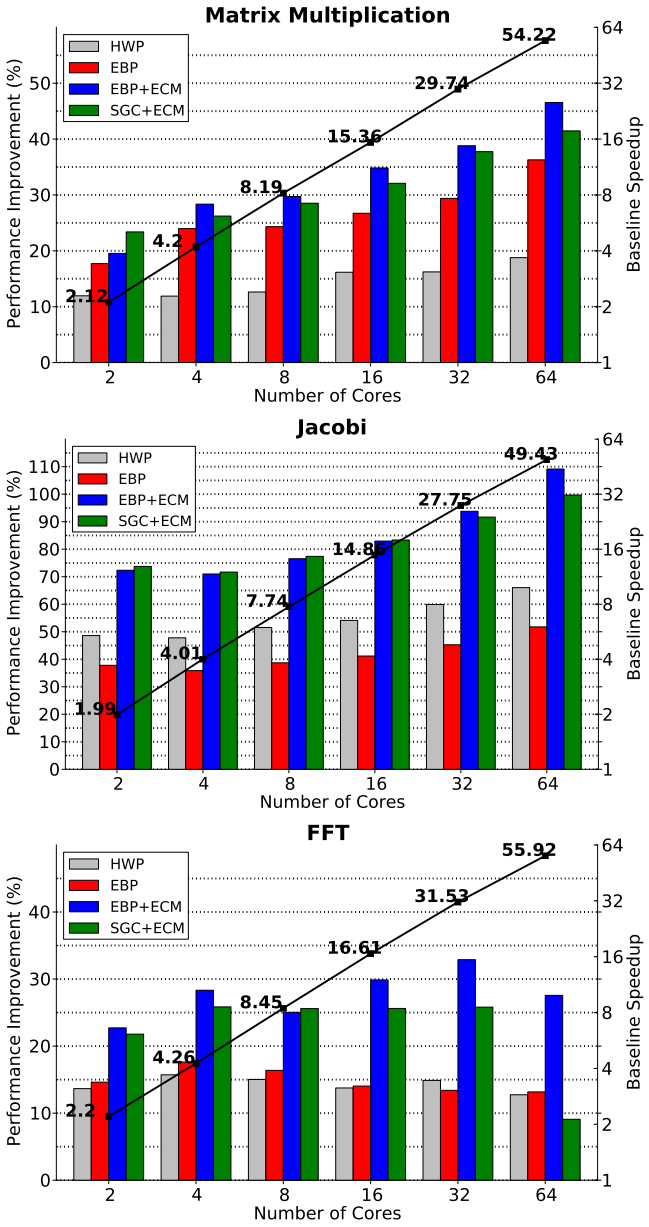



Figure 5.7: Performance improvement over the baseline without prefetching for each of the following configurations: (i) Hardware Prefetcher (HWP), (ii) Explicit Bulk Prefetcher (EBP), (iii) Explicit Bulk Prefetcher with Epoch-based Cache Management (EBP+ECM), (iv) Software Guided Coherence with Epoch-based Cache Management (SGC+ECM). The line plots the baseline speedup normalized to the serial code execution time that ignores the task pragmas (no runtime overhead). Plots are for Matrix Multiplication, Jacobi, and FFT benchmarks.

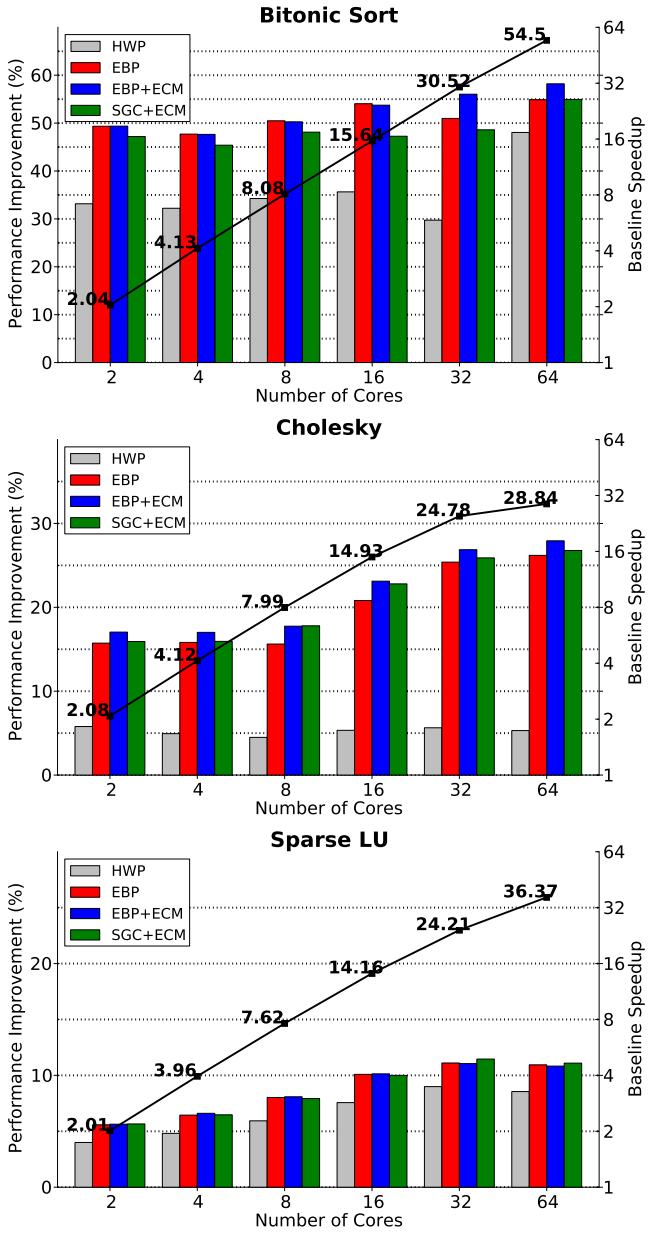



Figure 5.8: Performance improvement over the baseline without prefetching for each of the following configurations: (i) Hardware Prefetcher (HWP), (ii) Explicit Bulk Prefetcher (EBP), (iii) Explicit Bulk Prefetcher with Epoch-based Cache Management (EBP+ECM), (iv) Software Guided Coherence with Epoch-based Cache Management (SGC+ECM). The line plots the baseline speedup, normalized to the serial code execution time that ignores the task pragmas (no runtime overhead). Plots are for Bitonic, Cholesky, and Sparse LU benchmarks.

average performance degradation is 2% and 3% respectively. On average, for the 64-core configuration SGC+ECM is *8% faster* than EBP alone and *14% faster* than HWP.

### L2 Cache Misses

To gain more insight into the latter results, we study the impact of EBP and ECM in the number of L2 misses. Figure 5.9 presents the reduction in L2 misses over the baseline without prefetching for the four configurations and for setups with high cores counts (>=16). EBP with ECM significantly reduces the number of L2 misses, outperforms HWP, and achieves reduction of more than 80% on five out of six benchmarks (Jacobi, Bitonic, Cholesky, Sparse LU). HWP performs relative well in a number of benchmarks (Jacobi, Bitonic, Sparse LU) and reduces the associated L2 misses by up to 75%, however, in the rest of the benchmarks it cannot identify the memory access patterns accurately and the reduction in L2 misses falls below 40%. On the other hand, EBP with ECM consistently provides more than 18% additional reduction in L2 misses over HWP. The additional reduction in L2 misses ranges from over 18% (Jacobi, Bitonic, Sparse LU) to 68% (Cholesky). In FFT the reduction is up to 27% and in Matrix Multiplication is up to 45%.

EBP alone cannot always perform better than HWP (e.g. Jacobi) and is less effective than EBP+ECM. When EBP prefetches data and requires L2 replacements, the replacement policy cannot distinguish between old task data, current task data and next task data, thus the victim selection is not optimal, causes interference, and reduces the effectiveness of prefetching. On average, for the 64-core setup, EBP+ECM reduces L2 misses by an *additional 32%* over the HW-only prefetcher and an *additional 7%* over EBP alone.

In the non-cache-coherent system, SGC with ECM (SGC implements the EBP-NC version) offers a small additional reduction in L2 cache misses. The explanation for this improved behavior lies in the internals of the runtime version for non-cache-coherent systems. The task descriptors are "pushed" from the cache of the master core directly to the caches of the worker cores and messages with task descriptor pointers are delivered to the workers' mailboxes. The standard runtime version enqueues the task descriptors to the workers' queues and the workers effectively "pull" the descriptors from the master's cache; the cache-coherence protocol handles the underlying transfers. On average, for the 64-core setup, SGC+ECM reduces L2 misses by an *additional 6%* over EBP+ECM, by an *additional 13%* over EBP alone, and by an *additional 38%* over the HW-only prefetcher.

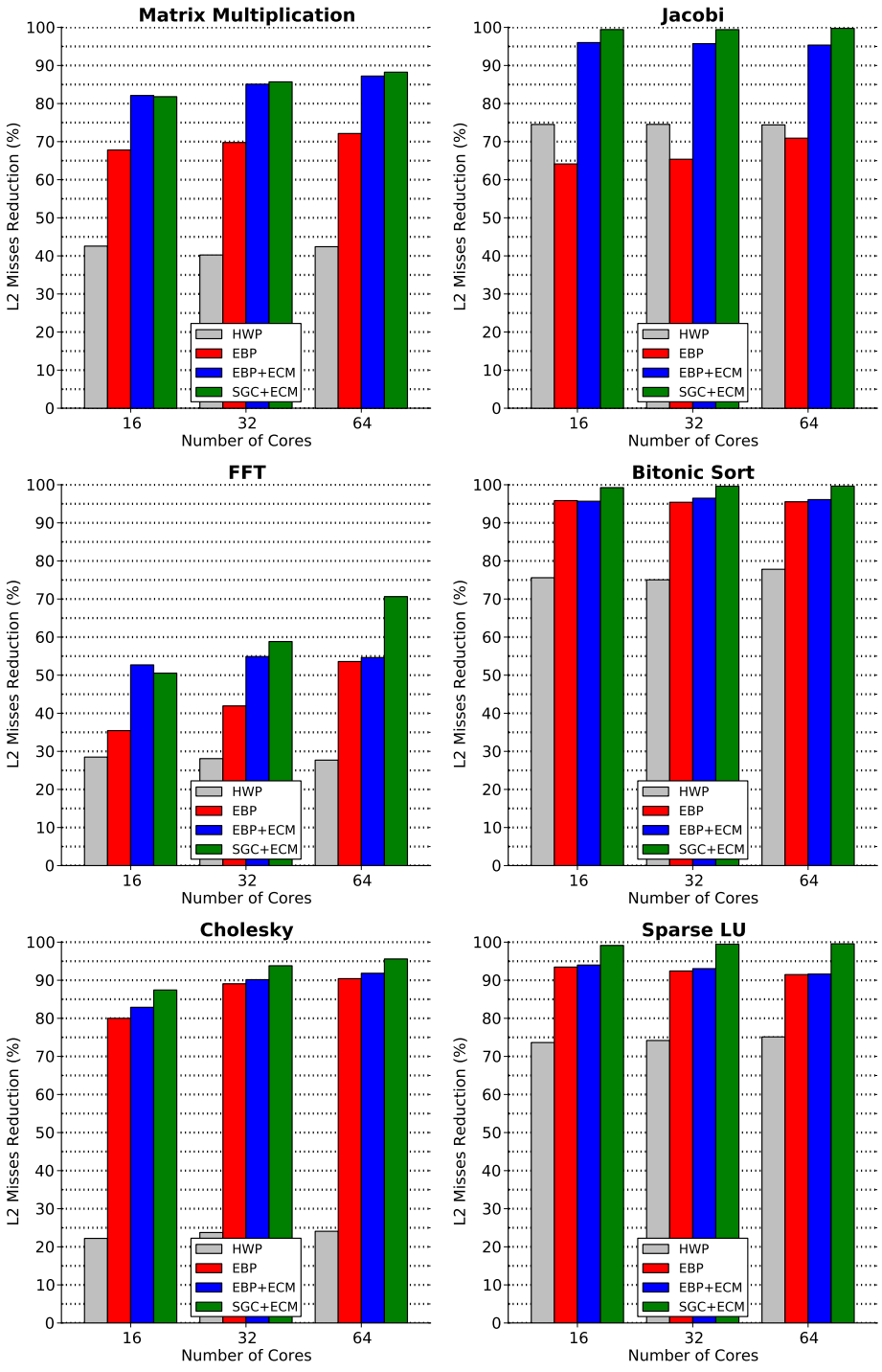



Figure 5.9: Reduction in the number of L2 misses over the baseline without prefetching when running with 16, 32, and 64 worker cores (higher is better).

### 5.2.2 Memory Traffic Analysis

We study the implications of EBP, ECM, and SGC on the memory system by measuring the on-chip and off-chip traffic. Given that the memory traffic is sensitive to task-scheduling, i.e. which task is executed on which core and in what order, we use a fixed off-line schedule that was captured when running the application on the baseline system. We replay the same schedule in the runtime for each of the four configurations. We present the results for on-chip network (NoC) traffic in Figure 5.10 and for off-chip memory (DRAM) traffic in Figure 5.11 for setups with high core counts (>=16). The traffic volumes are normalized to the traffic generated by the baseline without prefetching for each configuration.

#### On-Chip NoC Traffic

The results for on-chip traffic, presented in Figure 5.10, indicate that EBP with ECM does not generate excessive traffic when compared to the baseline without prefetching. In some cases, EBP with ECM generates even less traffic than the baseline. On the contrary, HWP results in increased traffic on most benchmarks because of inaccuracy in its predictions and the associated cache pollution. The improved traffic behavior observed in the 64-core setups is partly attributed to the increased on-chip cache size when the number of cores increases; on high core counts a larger portion of the benchmarks' datasets fits on-chip[3]. In the presence of prefetching (either HWP or EBP), data from the producers' caches are transferred earlier to the consumers' caches (before they are evicted), thus saving traffic. However, the improved behavior of EBP+ECM is also because of the smarter cache management of the task datasets and the throttling technique employed (Section 3.2.3).

EBP+ECM generates less on-chip traffic on all benchmarks when compared to both HWP and EBP alone, Figure 5.10. On the other hand, EBP alone generates significantly more traffic in a number of benchmarks (Matrix multiplication, Jacobi, FFT), which can be up to 55% higher than the baseline (e.g. FFT on 16 cores). EBP alone prefetches the requested cache-lines blindly in the cache and incurs destructive interference between data belonging to the active task contexts (current and next). When EBP prefetches cache-lines that map to fully-occupied sets with data from active tasks, the replacement policy has to evict useful data, whereas ECM throttles prefetching for these sets by consulting the epochs and

[3]We did not use larger datasets because of the prohibitive simulation times.

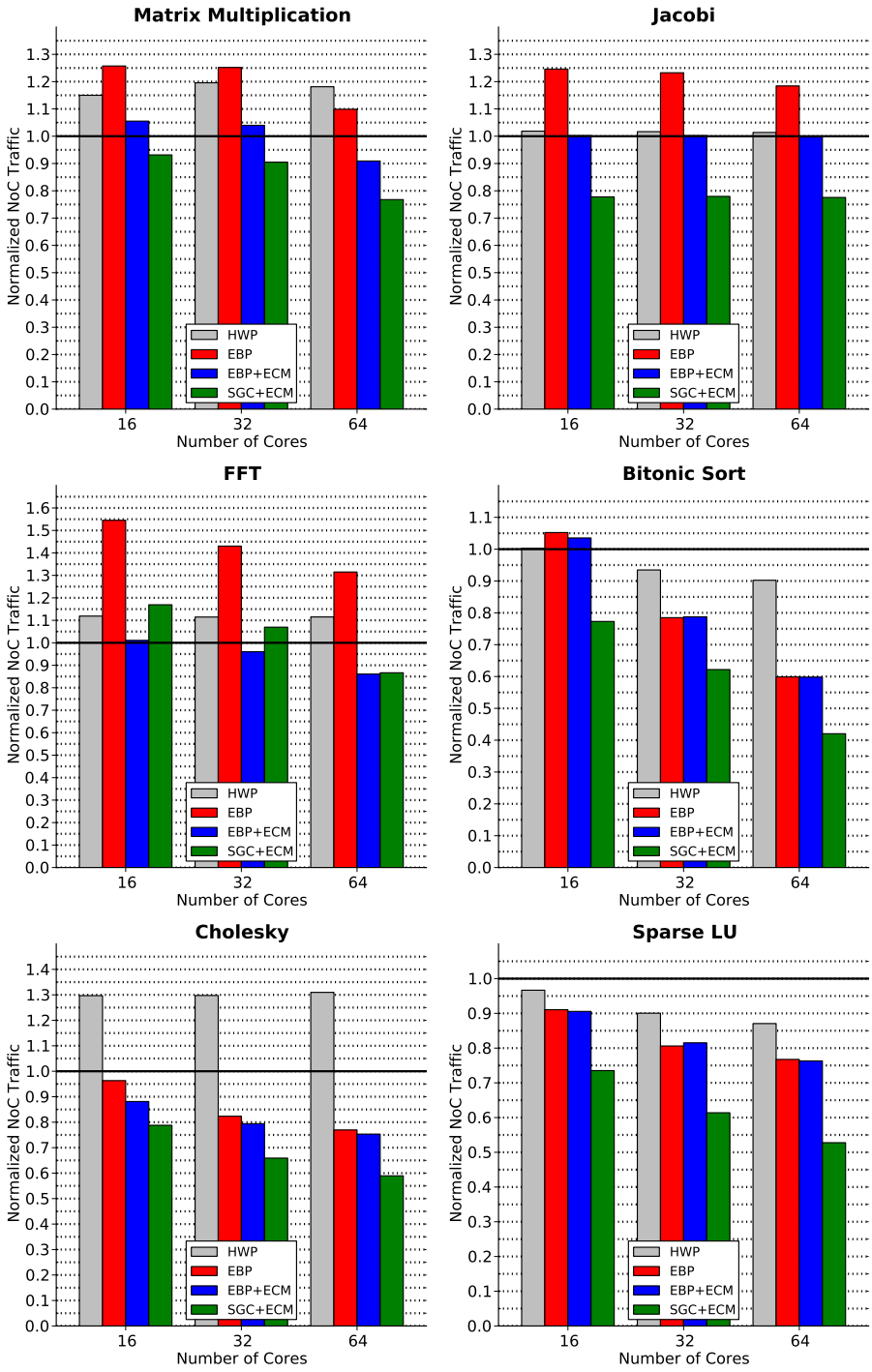



Figure 5.10: NoC traffic volume normalized to the baseline without prefetching when running with 16, 32, and 64 worker cores (lower is better).

their associated quotas. On average, for the 64-core setup, EBP+ECM generates *25% less* on-chip traffic than the HW-only prefetcher, and *14% less* traffic than EBP alone.

In the non-cache-coherent system, SGC with ECM achieves significant reduction in the on-chip traffic when compared to all cache-coherent configurations. The EBP-NC version employed here delegates the prefetch commands directly to the caches of the last producer/writer core; the runtime version for non-cache-coherent systems infers this information from the task graph and the history of tasks. This procedure eliminates a large portion of the control traffic required in cache-coherent systems, i.e. the cache follows the coherence protocol and makes requests (per cache-line) to the directory, which in turn forwards the requests to the current owner of the cache-line (the cache of the last writer). On average, for the 64-core setup, SGC+ECM generates *15% less* on-chip traffic than EBP+ECM, *30% less* traffic than EBP alone, and *41% less* traffic than the HW-only prefetcher. Moreover, when SGC+ECM is compared to the cache-coherent baseline without prefetching, it generates *34% less* on-chip traffic.

### Off-Chip DRAM Traffic

The off-chip DRAM traffic, presented in Figure 5.11, shows a similar trend with on-chip traffic and the pathological cases of EBP described earlier are again clearly shown. HWP increases DRAM traffic in a number of benchmarks due to inaccurate predictions and speculation about which data will be used by a task. On the other hand, the use of ECM allows the cache replacement policy to make better choices about which data to replace and avoids evicting data useful for the current and next task. On average, for the 64-core setup, EBP+ECM generates *30% less* off-chip DRAM traffic than the HW-only prefetcher and *20% less* traffic than EBP alone.

Bitonic presents a noteworthy behavior on 64 cores, where EBP reduces off-chip traffic by more than 80% over the baseline. This behavior is explained by the nature of comparison-based sorting which leads to a coherence pattern where data come as shared (to compare) and then upgrade to exclusive (to swap). In Bitonic, the application has marked this data as *inout* and EBP fetches them directly in exclusive mode. This avoids writebacks when data is dirty in another cache and downgrades due to the reads required for comparisons, assuming a MESI cache-coherence protocol.

The results for the non-cache-coherent system with SGC+ECM vary among the benchmarks. We observe that in a number of benchmarks (Matrix Multiplication,

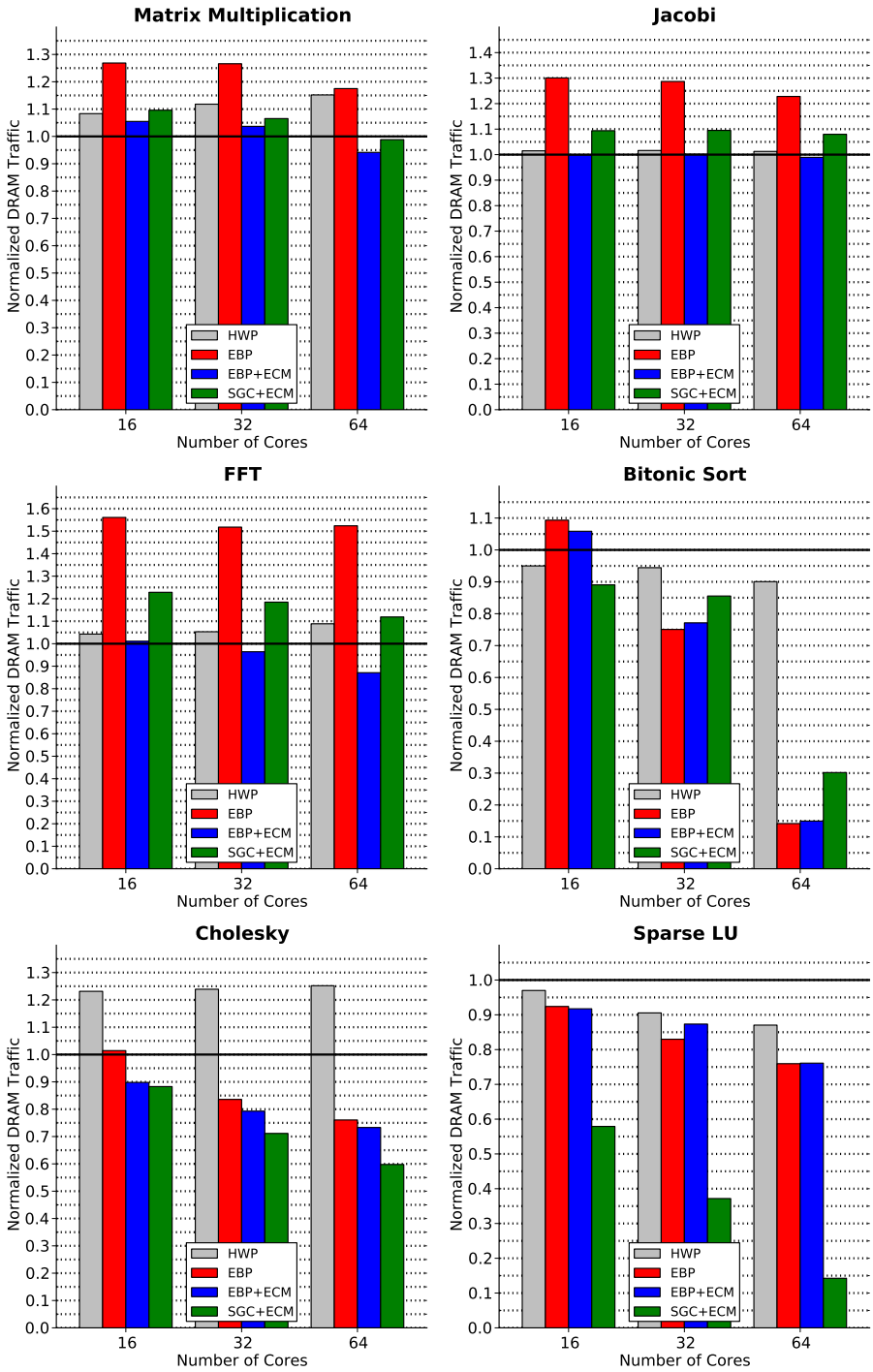



Figure 5.11: DRAM traffic volume normalized to the baseline without prefetching when running with 16, 32, and 64 worker cores (lower is better).

Jacobi, FFT) the DRAM traffic is up to 22% more than the baseline (FFT on 16-cores). In Bitonic the DRAM traffic is up to 16% more than EBP+ECM. The reason for this behavior is because SGC cannot apply the throttling technique with the help of ECM. The prefetch commands are delegated to the source node (last producer) and thus at the remote node EBP-NC cannot know if data can fit the destination cache (initiator). The remote EBP-NC cannot apply any throttling criterion since the occupancy and epochs at the destination cache are not available. Thus, all data are transferred and the portion of them that does not fit is skipped at the destination; cache sets may be full with data for active epochs. During task execution these data are fetched again from DRAM. In Cholesky and Sparse LU the traffic generated by SGC+ECM is less than EBP+ECM by up to 13% and up to 61% respectively. The reason for this improvement is that in the cache-coherent system the MESI coherence protocol serves clean/shared cache-lines from DRAM – silent evictions of clean cache-lines – whereas SGC serves them from the last producer cache; in Sparse LU a very large portion of data is found in the producer's cache. On average, for the 64-core setup, SGC+ECM generates *4% less* off-chip DRAM traffic than EBP+ECM, *22% less* traffic than EBP alone, and *34% less* traffic than HWP.

### 5.2.3 Energy and Power Analysis

Following the methodology discussed in Section 5.2.2, we use the simulator infrastructure to measure the dynamic energy consumption[4] of the memory system components (L1, L2, Directory, DRAM) and the on-chip interconnect (NoC). We present separately the on-chip energy and the total energy of the memory system. Additionally, we calculate the energy-delay product (EDP) metric and the power consumption. The results presented below are normalized to the baseline without prefetching for each configuration.

#### On-Chip Energy

We present our results for the on-chip dynamic energy (L1, L2, Directory, NoC) using breakdowns in Figure 5.12. We observe that the energy dominant components are the L1 cache and the NoC. The energy spent in each component varies among the benchmarks and largely depends on whether a benchmark is compute- or memory-intensive. The results indicate that EBP+ECM consumes almost the

[4]Note that we do not include static energy measurements, which are influenced by execution time, thus our comparisons are conservative.

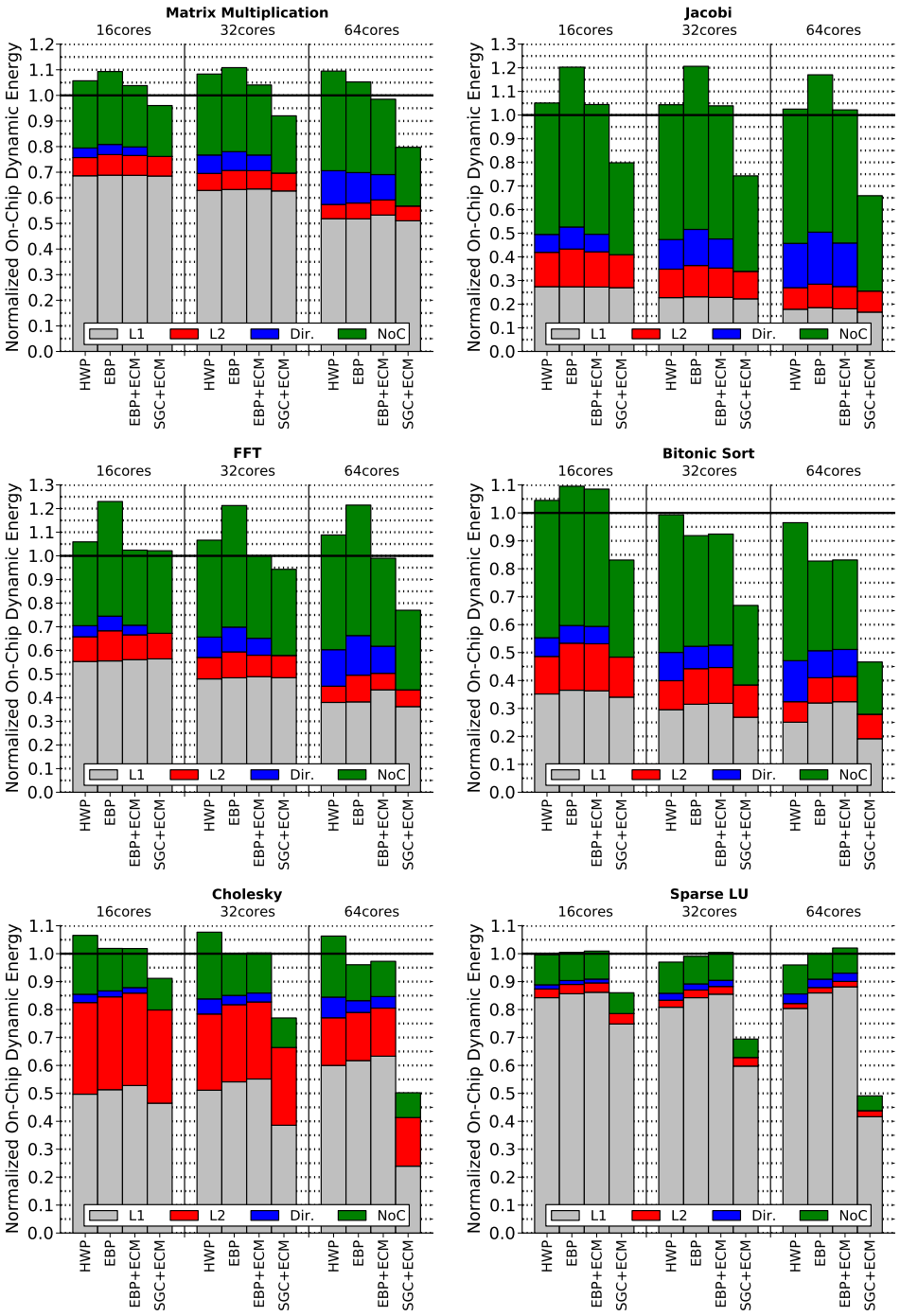



Figure 5.12: On-chip dynamic energy normalized to the baseline without prefetching when running with 16, 32, and 64 worker cores (lower is better).

same energy as the baseline, whereas the HW-only prefetcher is slightly more energy hungry. On the other hand, EBP alone increases energy consumption in a number of benchmarks up to 22% (Jacobi, FFT). On average, for the 64-core configuration, EBP+ECM consumes *6% less* on-chip energy than EBP alone, and *7% less* energy than the HW-only prefetcher.

The results on the non-cache-coherent system with SGC+ECM show significant reduction in the on-chip energy consumption when compared to the cache-coherent system. On a large extend, the improvement comes from the on-chip network due to reduced on-chip traffic (Section 5.2.2) and to a lesser extend by the elimination of the hardware coherence directory. Moreover, we observe a significant reduction in L1 cache energy for Cholesky and Sparse LU on 64-cores. These benchmarks lack enough task parallelism to utilize all cores (illustrated in the speedup curves in Section 5.2.1) and the workers spin waiting for tasks. In the non-cache-coherent version of the runtime the workers "block" on the mailbox, thus saving energy. On average, for the 64-core setup, SGC+ECM consumes *35% less* on-chip energy than EBP+ECM, *42% less* energy than EBP alone, and *41% less* energy than the HW-only prefetcher. Moreover, SGC+ECM consumes *38% less* on-chip energy than the cache-coherent baseline without prefetching.

#### Total Memory System Energy

We include the off-chip DRAM dynamic energy and present the total memory system dynamic energy using breakdowns in Figure 5.13. In almost all the benchmarks, the dynamic energy consumption is dominated by the off-chip DRAM, while the second major component is the L1 cache. The most important finding is the reduction of DRAM energy when prefetching is used (either HWP or EBP). This behavior is due to the open-row management and the scheduling policy of the memory controller (FR-FCFS [98]). With prefetching, the memory controller has the potential to serve more requests from open rows and reduce the number of row activations and precharges, which contribute significantly to DRAM energy. With EBP, task data is requested in bulk and close in time, thus offering the memory controller more opportunities to exploit open row buffer locality. On average, for the 64-core setup, EBP+ECM consumes *14% less* energy than EBP alone, and *28% less* energy than the HW-only prefetcher.

On the non-cache-coherent system, SGC+ECM applies even more aggressive prefetching and allows more outstanding requests to memory since it uses the hardware counters to trigger transfer completion. On the other hand, the number of

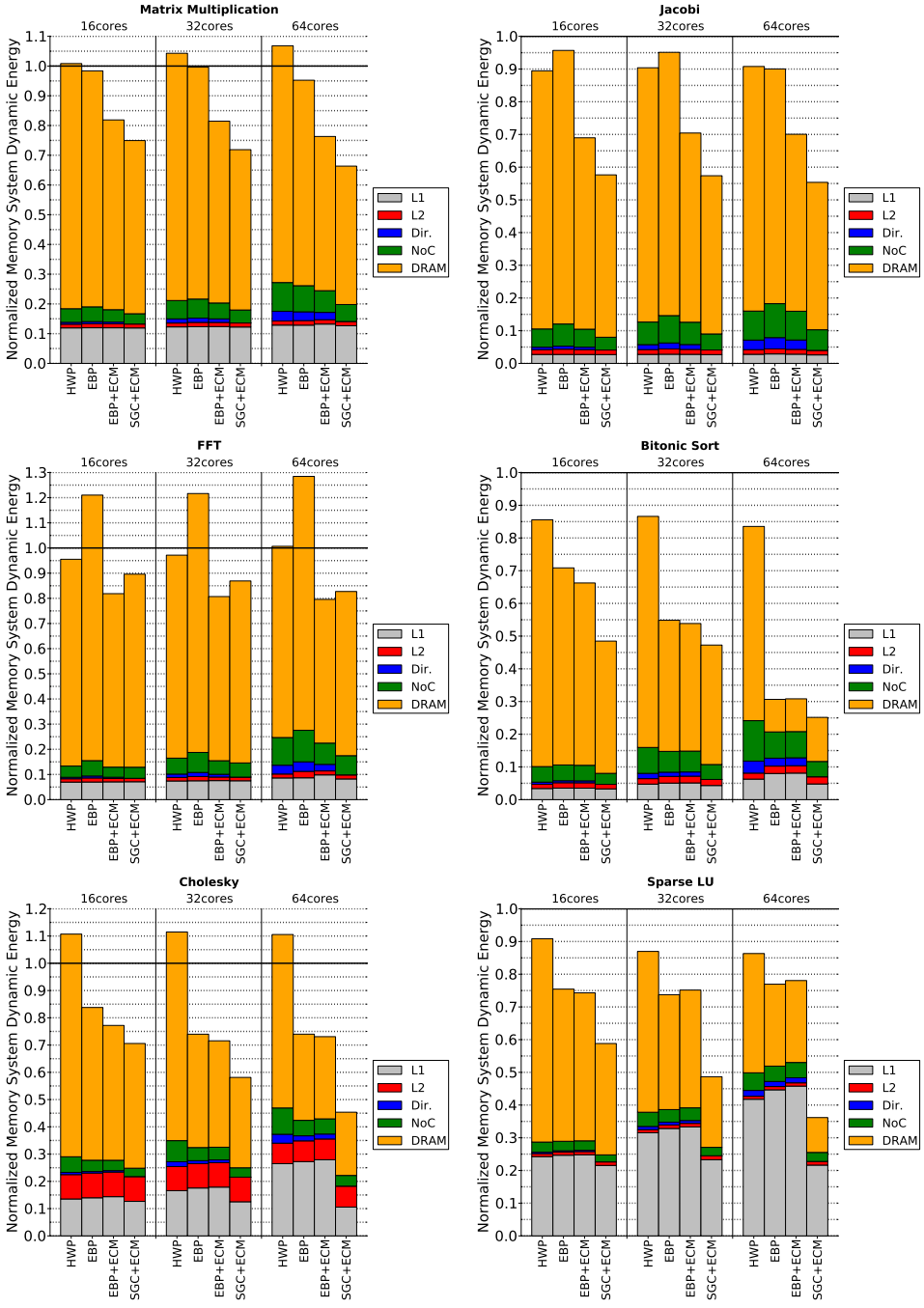



Figure 5.13: Dynamic energy consumption of the memory system's components. The energy is normalized to the baseline without prefetching when running with 16, 32, and 64 worker cores (lower is better).

outstanding requests in the cache-coherent system is limited by the number of miss status handling registers (MSHRs). Furthermore, the reduction in on-chip and off-chip DRAM traffic we observed before (Section 5.2.2) offers additional energy savings. SGC+ECM consumes *16% less* energy than EBP+ECM, *30% less* energy than EBP alone, and *44% less* energy than the HW-only prefetcher. Moreover, SGC+ECM consumes *48% less* energy than the cache-coherent baseline without prefetching.

**Energy-Delay Product**

We use the energy-delay product (EDP) [99, 100] as metric to measure the efficiency of our hardware primitives in our dual goals: *(a)* lower energy consumption and *(b)* higher performance (lower delay). EDP offers equal "weight" to energy or performance loss, so if either energy or delay increase, EDP will increase; lower EDP values are desirable. Figure 5.14 presents our results using the total energy in the memory system and the execution time of each benchmark.

All configurations improve the performance over the baseline without prefetching (Section 5.2.1), but some of them result in increased energy consumption in a number of benchmarks (Section 5.2.3). The EDP metric that treats performance and energy equally provides more insight about the efficiency of the associated mechanisms. EBP+ECM performs better in all benchmarks and consumes less energy so we observe important EDP reduction. EBP alone suffers from increased energy consumption in a number of benchmarks (Jacobi, FFT) and has achieved lower performance than EBP+ECM, so this aggregate effect is also depicted in EDP figures. The hardware prefetcher improved performance less than the competing configurations and consumed more energy, thus the EDP reduction is also lower. On average, for the 64-core setup, EBP+ECM achieves *16% lower* EDP than EBP alone, and *30% lower* EDP than the HW-only prefetcher.

The comparison of the non-cache-coherent system with SGC+ECM against the configurations of the cache-coherent system is intriguing. SGC+ECM showed a small performance degradation in execution time when compared to EBP+ECM, but on the other hand reduced the energy consumption in the memory system. The results indicate that the energy reduction outweighed the performance loss in almost all the benchmarks. The only exception is FFT since the performance loss due to saturation of the task master (as explained before) gives a disadvantage. On average, for the 64-core setup, SGC+ECM achieves *10% lower* EDP than EBP+ECM, *26% lower* EDP than EBP alone, and *40% lower* EDP than the HW-only prefetcher.

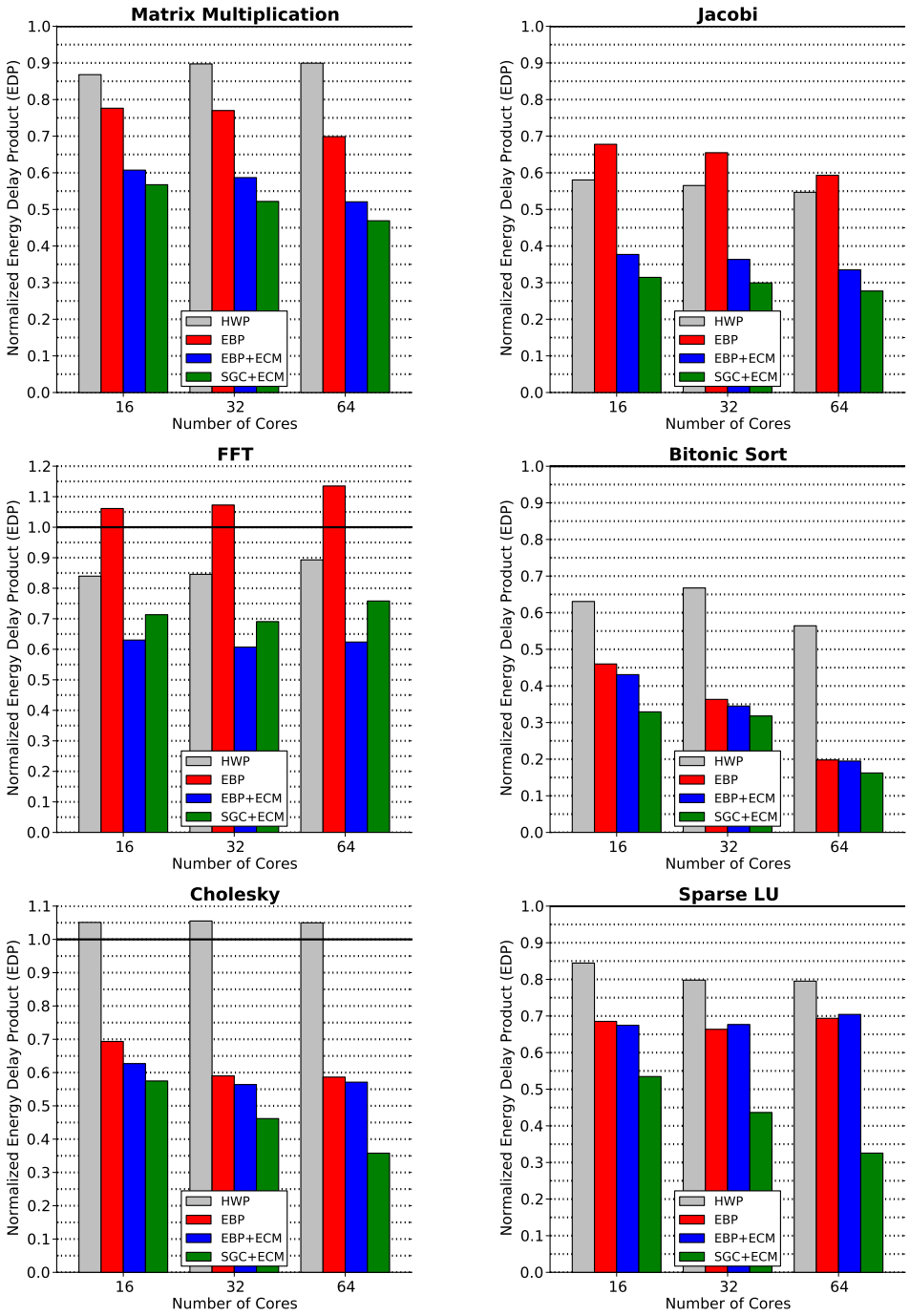



Figure 5.14: Energy-delay product using the dynamic energy of the memory system's components. The energy-delay product is normalized to the baseline without prefetching when running with 16, 32, and 64 worker cores (lower is better).

### Power Consumption

We measure the total dynamic power consumption of the memory system and present our results in Figure 5.15. Power is the rate of energy consumption and thus faster execution without proportional reduction in energy leads to increased power consumption. Our results indicate that EBP+ECM achieves lower dynamic power consumption than the baseline without prefetching in half of the benchmarks (Bitonic, Cholesky, Sparse LU). In Matrix Multiplication and FFT the dynamic power consumption increases up to 13% and up to 7% respectively. In Jacobi, the most memory intensive benchmark, the dynamic power increases by up to 46%. The latter behavior means that the performance improves more than energy reduction in these benchmarks. The performance in Jacobi improved by more than 80% in the higher core counts (>=16) and thus reducing the energy at the same rate is very hard. Finding the "sweet spot" to achieve the same dynamic power with the baseline, at the cost of lower performance improvement, is feasible, however, this is not the focus of this work. The hardware prefetcher dissipates significantly more power in almost all benchmarks by double-digit percentages that can be as high as 51% (Jacobi). EBP alone is not always better than HWP and dissipates more power in several cases (Matrix Multiplication, FFT). On average, for the 64-core setup, EBP+ECM consumes *8% less* power than EBP alone, and *23% less* power than the HW-only prefetcher. When compared to the baseline without prefetching, EBP+ECM consumes on average *2% less* power on 64-cores.

The results on the non-cache-coherent system with SGC+ECM show significant reduction in the dynamic power consumption when compared to the cache-coherent system. SGC+ECM consumes energy more efficiently and achieves lower dynamic power than the baseline without prefetching in most of the benchmarks. The exceptions are Jacobi and FFT where the increase in dynamic power consumption tops at 10%; the explanation in the previous paragraph also holds here. On average, for the 64-core setup, SGC+ECM consumes *26% less* power than EBP+ECM *34% less* power than EBP alone, and *49% less* power than the HW-only prefetcher. When compared to the baseline without prefetching, SGC+ECM consumes on average *28% less* power on 64-cores.

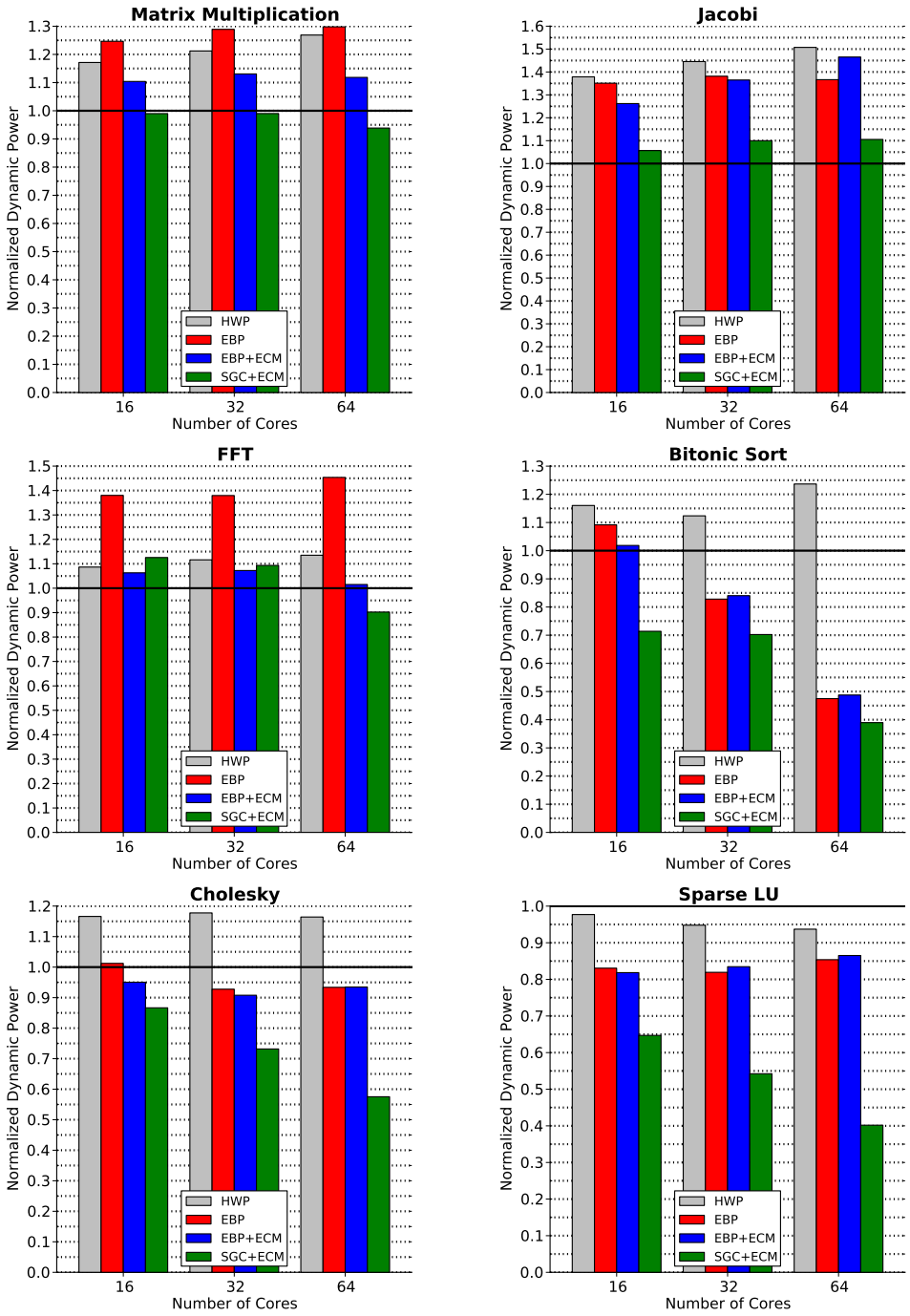



Figure 5.15: Dynamic power consumption of the memory system's components. The power is normalized to the baseline without prefetching when running with 16, 32, and 64 worker cores (lower is better).

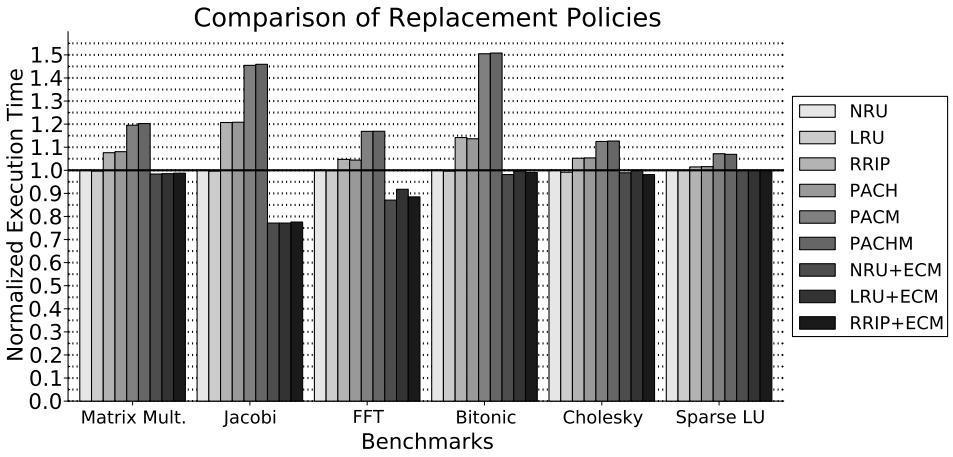



Figure 5.16: Comparison of replacement policies when EBP is utilized, runs for 16 worker cores.

### 5.2.4 Comparing Cache Replacement Policies

We explore a number of candidate replacement policies for use with EBP and present our findings for 16-cores in Figure 5.16. We examine the typical NRU and LRU replacement policies, RRIP [62], and the state-of-the-art prefetch-aware replacement policy PACMan [61] in its three variants: PACH, PACM, PACHM[5]. Moreover, given that ECM is versatile and can be used in conjunction with many replacement policies, we also evaluate three variants with ECM: *(i)* NRU+ECM, *(ii)* LRU+ECM, and *(iii)* RRIP+ECM. We observe that when EBP is utilized, the PACMan variants perform worse than NRU and LRU, whereas the ECM variants can be up to 22% faster than LRU (Jacobi). In PACMan, the reference history that is accumulated during a task lifetime is useless for the next task and requires several misses before cache-lines with old task data become candidates for replacement. This behavior appears to have a negative effect on performance. On the other hand, the ECM epochs help to effectively filter "old" cache-lines and allow the data prefetched for the next task to be preserved in the cache.

[5]For RRIP and PACMan we use 2-bit values and the version without set dueling.

# 6 Conclusions

This chapter concludes this thesis. We summarize our work in Section 6.1, discuss the lessons learned in Section 6.2, and present future work in Section 6.3.

## 6.1 Summary

This thesis designs memory hierarchies that allow software to manage data locality, and presents simple, yet efficient, hardware communication primitives that permit software to guide data transfers – thus to control how and when cores communicate. We show that software has the knowledge and the potential to guide hardware so as to improve performance and reduce the energy cost of communication in manycore architectures.

In Section 3.1 we presented a memory hierarchy that incorporates the functionality of both a coherent-cache and a scratchpad memory in a common address space. Software can opt to utilize a portion of the local cache memory as scratchpad memory and use our explicit hardware communication primitives in order to optimize and control data locality for the cases where it can intelligently do so, i.e. when it has prior knowledge about the dataset and the associated memory ranges. For the portions of the dataset where data accesses are not known in advance or ex-

hibit irregular access patterns, e.g. tree traversals, the software may delegate control to the cache and the coherence protocol. The explicit hardware communication primitives allow software to get involved in the data movement when it knows the locations of the data in the memory hierarchy, i.e. the addresses and the cores that it needs to communicate with. The evaluation of our explicit communication support shows that applications with known datasets and communication patterns can benefit from our hardware primitives to improve performance, to remove the unnecessary control overhead imposed by cache-coherence, and to reduce the energy consumption of on-chip communication. Our experimental results show that our proposed hardware primitives can improve performance by 10% to 40%, and at the same time reduce the energy consumption of on-chip communication by 35% to 70% owing to significant reduction in on-chip traffic by factors of 2 to 4.

In Section 3.2 we considered cache-based memory hierarchies with and without cache-coherence. We proposed a hardware/software co-design, assuming a task-based dataflow programming system, which uses tasks with annotated memory footprints. The task-based dataflow programming models use the memory footprints of tasks to build dependency graphs (DAGs) and maintain significant amount of information that is used to dynamically drive runtime decisions. Essentially, the runtime system discovers producer-consumer relations among tasks, maintains such knowledge internally and uses it to schedule and execute the tasks in the correct order.

We exploited the opportunities and the knowledge inherent in such programming environments and proposed hardware primitives to allow the runtime to guide hardware data transfers and improve cache locality. We introduced the *Explicit Bulk Prefetcher (EBP)*, which allows the runtime to prefetch task data in bulk, before each task executes, and perform common optimizations such as double-buffering. This form of software-guided prefetching controls data movement in the cache and addresses some of the challenging issues with hardware-only prefetchers such as *timeliness*, *accuracy*, and *access pattern prediction*. Moving further, we proposed *Epoch-based Cache Management (ECM)*, a generic lightweight mechanism that allows software to guide the cache replacement policy, expose its knowledge of tasks to the cache hierarchy, assign cache resources to them, and secure that useful cache-lines are not evicted by the prefetched lines. ECM is based on the notion of *Epoch*, which can be defined by software as the lifetime of a task, i.e. the time period during which a task executes. Thus, ECM guides the cache replacement policy, controls cache locality, and allows the cache to identify and

prioritize data belonging to different tasks, so as to minimize data movement. Our experimental results in cache-coherent systems show that our proposed hardware primitives (ECM, EBP), when compared to hardware prefetching, improve performance by an average of 20%, inject 25% less on-chip traffic on average, and reduce the energy consumption in the components of the memory hierarchy by an average of 28%.

The memory footprints of tasks, the producer-consumer relationships identified between tasks, and the scheduling history, allow the runtime to know the addresses and the locations of data throughout the system. The software effectively keeps a directory at task argument granularity, and thus, it can directly guide data movement and communication in the memory hierarchy without the traffic and energy overhead of hardware cache-coherence. We exploited this very important opportunity and proposed hardware primitives for *Software Guided Coherence (SGC)* in non cache-coherent systems. We enable runtime software to orchestrate fetching the most up-to-date version of the task arguments from the appropriate cache(s) directly, and maintain coherence at task granularity. Our hardware support for non cache-coherent systems (ECM, SGC) compared to cache-coherence and hardware prefetching, improves performance by an average of 14%, injects 41% less on-chip traffic on average, and reduces the energy consumption in the components of the memory hierarchy by an average of 44%.

## 6.2 Discussion

Throughout the design and evaluation of our architectural support we faced several issues regarding the development of applications, the scalability of runtime software, and the efficiency of our hardware primitives. We discuss below the main issues, and provide insight on the use of our architectural support.

The development of applications that directly exploit the hybrid cache/scratchpad memory hierarchy and the explicit communication hardware primitives is a very tedious and time-consuming task. It requires very careful management of the limited local memory resources, and sheer understanding of the data exchange patterns between the communicating cores. Achieving direct transfers between the scratchpad memories is the most challenging task, since it requires the programmer to orchestrate the transfers between the asynchronous contexts that produce and consume data. In our evaluation we used datasets that fit in the on-chip memories, however, applying the same techniques (direct transfers between scratchpads)

using larger datasets depends on the intrinsic time proximity of data reuse in each application, and requires more sophisticated application algorithms, such as multi-level tiling [101]. We did not modify the application algorithms to directly compare against the standard versions used in cache-coherent systems.

Moreover, in the hybrid cache/scratchpad memory hierarchy, the use of RDMA commands that transfer small portions of data cannot be always amortized adequately, but RDMA is always beneficial when each transfer exceeds a certain size (i.e. larger than 128-bytes). We proposed the use of coalesced "Remote Stores" for the cases where short transfers are required, however, exploiting this hardware primitive in large application parts is a complicated procedure. The application programmer has to carefully identify the stores that are used for computations, and decide which of them to convert to "remote-stores" (by selecting the appropriate remote addresses), so as to achieve communication inter-mixed with computation. We applied the latter technique in the FFT kernel, and our experience confirms the complexity to use "remote-stores" manually. Applying this technique extensively requires compiler analysis to identify the last write access for each memory address in code blocks [102]. On the other hand, sporadic use of remote-stores for low volume communication is easy and efficient.

The issues we described so far concern the manual use (by the programmer) of the architectural support for the hybrid cache/scratchpad memory hierarchy. We solve these issues in our more generic architectural support for cache-based systems that is co-designed with a task-based dataflow runtime system.

The evaluation of our hardware/software co-design using fine-grain tasks revealed a number of scalability bottlenecks in the task-based runtime software. Our runtime utilizes a single master thread that spawns tasks, performs dependence analysis, and schedules tasks. We find that fine-grain tasks that execute for a few tens of thousands of clock cycles saturate the single master thread, and thus this master thread cannot utilize all the available worker threads/cores. Such behaviors appear in our evaluation mainly when 64 cores are available. To utilize multiple master threads, the runtime software requires a more sophisticated design, especially when the system does not feature coherent-caches; an example of such an advanced and scalable runtime system is Myrmics [103].

Our evaluation also revealed, that the most performance critical part of the runtime system is dependence analysis. Our runtime implements block-based dynamic dependence analysis with arbitrary byte granularity [38] and uses internal metadata structures that keep state for the dependent tasks; the block granularity is

selected by each application. One of the benefits from the latter approach is that it permits arbitrary overlapping between the address ranges of task arguments. However, several task-based applications with fine-grain tasks use *2D memory ranges* with small element blocks, thus the dependence analysis has to examine a large number of small blocks. For instance, a task argument of 32 KBytes that uses $64 \times 64$ elements (element size of 8 bytes), requires the runtime to examine 64 metadata and perform the associated dependence checks. Efficient dependence analysis under the latter constraints is a good candidate for specialized hardware support. Nexus++ [104] offers such hardware support, however, it uses only the base addresses of arguments, and does not support overlapping address ranges.

Our hardware support for *Software Guided Coherence (SGC)* assumes that a task argument is produced by a small number of producer tasks, thus, it would require a small number of commands to directly fetch the most recent version of a task argument from the producers' caches. However, we found that FFT exhibits excessive fragmentation of task arguments, i.e. internal pieces of an argument have been produced by many different tasks that run on many different cores, due to the transpose steps. The latter effect causes the runtime to issue many prefetch commands that cannot be amortized, thus, negatively affecting the performance and the efficiency of our hardware support. However, our scheme still reduces the energy when compared to cache-coherence with hardware prefetching. Such cases can be addressed at application level, the code can use extra memory buffers, change the data layout across phases, and apply the equivalent of *OpenMP copyout* in an on-chip memory that will be treated as the *home location* of data. We did not modify the application algorithms to directly compare against the standard versions used in cache-coherent systems.

## 6.3 Future Work

We list here some ideas for further extensions that came up during the evaluation our architectural support, and can potentially improve performance.

#### Support for Cache Allocation of Write-First Data

The "output" type task arguments specified by the OmpSs/SMPSs task-based programming model, offer an opportunity for hardware optimization. An output task argument effectively means that this argument is considered write-first and thus, the old values of the associated memory range are "don't care". We can offer an

extra operation in the non-cache-coherent version of the *Explicit Bulk Prefetcher* (EBP-NC), e.g. "Allocate" opcode, to allow software to directly allocate the cache-lines associated with an output argument, without requiring to fetch the old data. The contents of these cache-lines can be filled either with zero values, or with values from a pseudo-random generator (security feature), or keep the values of the cache-line that is being replaced (don't spend energy on the data arrays). If the address range is not cache-line aligned, the unaligned portions of the range have to be fetched from main memory. However, the use of such primitive requires software to flush the associated range from the old producer cache, if any, to avoid having the same cache-line dirty in more than one cache. For the latter operation, we can offer an extra operation in EBP-NC, e.g. "Purge" opcode, that would only clean the dirty bits of an address range in the remote cache, instead of flush, in order save energy from the associated write-backs; such write-backs are useless.

**Epoch-Based Flushing and Eager Self-Flushing**

The inherent characteristics of some application algorithms, or the scheduling policies of runtime systems, may cause excessive delay between the execution of dependent tasks (producer-consumer pairs), leading to long data reuse distances, and possibly limited on-chip temporal locality. The latter behavior could be an effect of breadth-first like scheduling that favors new tasks instead of dependent tasks (favored by work-first like scheduling). Exhibiting long data reuse distances effectively means that the producer cores would run several independent tasks before a consumer requests that data, therefore, data produced in the past have low probability of been present in the producer's cache. However, the runtime would still make such requests to the latest producer core to ensure correctness; required by *Software Guided Coherence*. Although we discourage the use of inefficient algorithms and scheduling policies, we can offer hardware support to improve such cases.

We can offer hardware flushing based on epochs, instead of addresses. We can modify the *Epoch-based Cache Management* scheme, so that all the dirty cache-lines of an epoch are automatically flushed before an epoch is reused (becomes active). The epoch-based flushing mechanism would access each cache-set once, and would guarantee the maximum number of probes in the cache; always equal to the number of cache sets. The epoch-based flushing mechanism can be further improved if the hardware tracks with counters the number of dirty cache-lines per epoch. When a cache-line becomes dirty, the associated epoch counter will be incremented, and when an eviction occurs the counter will be decremented. By

counting the number of dirty cache-lines per epoch, flushing can terminate when the associated counter is zero; we hope that most times flushing will be terminated after a few probes (significantly less than the maximum required).

The use of dirty-line tracking counters can assist the automatic flushing mechanism, and we believe that most times these counters would be zero (or close to zero), thus minimizing the need for flushing. In order to further minimize the need for flushing, we can also offer an optional "eager-flush" mechanism that can be enabled by software. The eager-flush mechanism evicts pro-actively data belonging to the oldest (inactive) epoch during processor accesses, i.e. when a processor accesses a cache-set to acquire data for the current task, the dirty cache-lines that belong to the oldest epoch (oldest task) are victimized.

The automatic self-flushing mechanism has an important property: guarantees that data produced outside a "max-epochs" window will not be present in the producer's cache. The latter guarantee can allow software to avoid fetching data from the last producer, if that specific producer core has executed more than "max-epochs" tasks since the actual producer task. In order to exploit this guarantee, software should keep a per-core task sequence number and store, in the internal data-structures, the associated sequence number for each task argument.

# A

# FPGA Prototype

## A.1 FPGA-based Implementation

Our hardware prototype is implemented in a Xilinx Virtex-5 FPGA using four MicroBlaze soft-cores as processors. The processors are 32-bit, in-order, and have a traditional 5-stage pipeline that also supports single-precision floating-point operations. Each processor tile has a private data cache hierarchy, with a configurable L2 cache/scratchpad memory tightly-coupled with our NI. Instructions are fetched from private L1 instruction caches. The prototype is equipped with a 256MB DDR2 SDRAM that is used as main memory and is shared between tiles. Communication between tiles and the on-chip DRAM memory controller is achieved through a 64-bit, 5-port crossbar switch (XBAR) that features three priorities and applies round-robin scheduling; contention-less crossbar traversal costs 1 clock cycle. An additional switch port can be used to provide multi-FPGA connectivity through multiple external high-speed serial links (RocketIO), and thus our modular design can be expanded with multiple boards in order to build larger scale systems. Cache-coherence is not currently supported. The operating clock frequency of the system is currently 75MHz and its block diagram along with the major components is illustrated in Figure A.1.

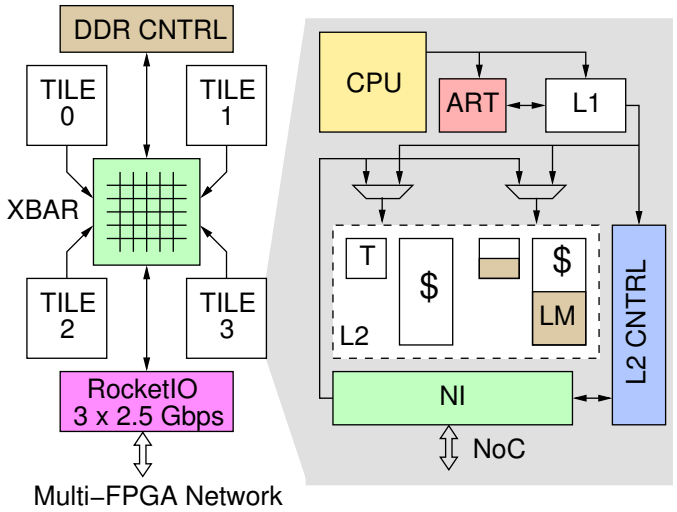



Figure A.1: FPGA Prototype Block Diagram

### A.1.1 Configurable Cache/Scratchpad Memory

Every tile of our prototype implements a private data L1 cache and a private, configurable, data L2 cache/scratchpad. These are smaller than one would expect in a CMP, due to limited FPGA resources. Typically, L1 caches range from 16 to 64 KBytes, 2 to 4 way set associative, with 64-byte lines. Our implementation has scaled down the L1 caches to 4KB, direct-mapped, with 32-byte cache-lines. L1 caches are write-through, with 256-bit wide (one cache line) refills, a single-cycle hit latency, and follow "no-allocate" policy on store misses. L2 caches, on the other hand, are usually much larger, with sizes beyond 1MB, associativity up to 16-ways, and line size of 64 bytes or more. Scaling down again, we have designed a 64 KB, 4-way set-associative write-back L2 cache with 32-byte lines. Our L2 cache supports multiple hits under a single miss in order to minimize processor idle time. The L2 controller serves write-backs and fills on misses, using the transfer primitives of the tightly-coupled NI as described below.

The key component that allows us to configure and use parts of the L2 cache as scratchpad is the Address Region Table (ART); its function is similar to a traditional TLB, but it provides only protection and type information –not physical address translation– hence the ART can be smaller than a TLB (and have no misses), because it can describe potentially huge regions of the address space in each entry. The ART classifies each memory access as one of: *(i)* cacheable, *(ii)* local scratchpad, *(iii)* remote scratchpad, *(iv)* tag access (used to access and set lock bits in L2),

or *(v)* register access (NI control registers that customize specific features). The ART is placed in parallel with the L1 cache, and is probed on every memory access of the processor; a copy of the ART is also used by the incoming NI. Routing in our prototype is based on physical addresses, thus we use a static mapping: each L2 data and tag array has a unique physical address (nodeID and way number are encoded in the address bits).

An important issue for the efficient use of scratchpads and their associated DMAs is the available memory bandwidth. Scratchpad areas in our design are hosted inside the L2 memory banks and the NI accesses them at high rate when performing DMAs. On the other hand, the default set-associative cache organization would require all the ways (tags + data) to be probed in parallel, causing conflicts and thus limiting the available data array bandwidth for the NI. In order to reduce the bandwidth required by the typical L2 cache operation and use it more efficiently for NI operations, we implement a phased L2 cache: the tag arrays are accessed first and the data arrays are accessed only on hits. Our cache-line wide (256-bit) L2 data array, allows L1 misses to be served in a single clock cycle, thus we avoid occupying the data arrays for multiple cycles. The outgoing and incoming NI paths also access data in 256-bit chunks and since the NoC is 64-bit wide, the maximum access rate per path is 1 per 4 clock cycles. As a result, more than 50% of L2 cycles is guaranteed for L1 requests.

Figure A.2 presents the datapath and the pipeline of our design. All memory accesses arriving from the processor are checked against the ART regions and probe the L1 cache. Hits are served normally, while misses, stores, scratchpad and tag accesses, are sent to L2 along with all required control information: type of access and way (if scratchpad). In the first cycle, the L2 controller arbitrates among requests from NI in, NI out and the L1, and probes the tags. In the next cycle, the selected agent accesses the data of a specific cache way. To reduce scratchpad access latency, scratchpad lines are L1-cacheable. The L2 controller keeps the cached scratchpad lines coherent by issuing local invalidations when writes arrive from remote nodes; no further coherence actions are required since the L1 is write-through. Scratchpad loads that miss in the L1 have a minimum latency of 4 clock cycles, while stores take 3 clock cycles to reach the L2. The observed processor latency for stores is 1 clock cycle, stores are immediately acknowledged and propagate in the pipelined memory hierarchy.

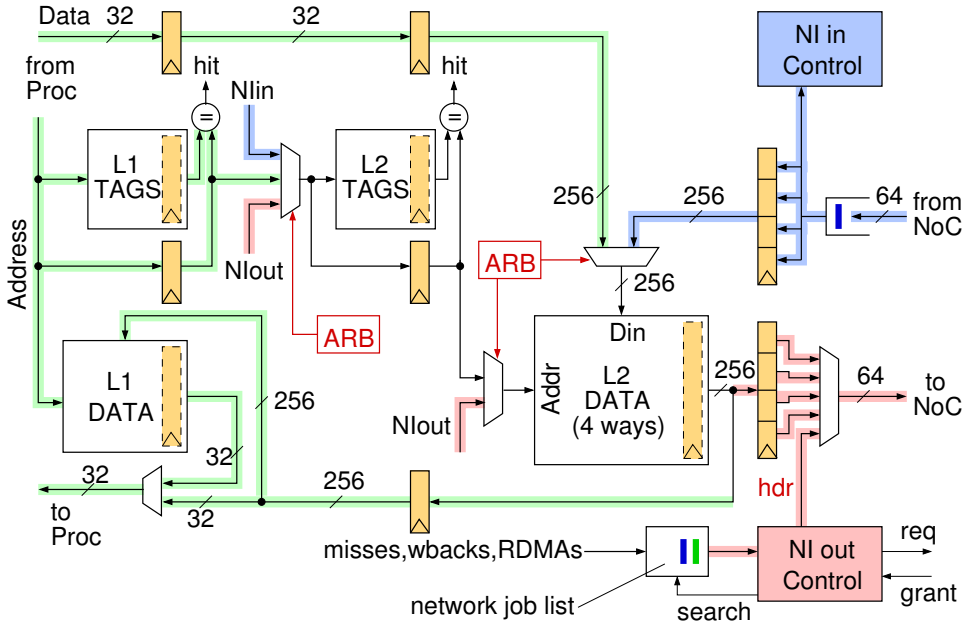



Figure A.2: Cache/Scratchpad Pipeline

### A.1.2 NI Operation and Mechanisms

The NI is tightly-coupled to the L2 cache and serves all transfers from/to the tile's memory and the NoC. The heart of the outgoing NI path is the *Network Job List* that keeps the posted jobs that need to be served. The incoming NI path serves inbound traffic, stores data in-place and, depending on the type of traffic (cache or DMA), collaborates with the L2 controller to complete operations.

*NI Command and Control Lines* are allocated upon software demand inside scratchpad areas. The state bits of locked lines distinguish them to four types:

- *Normal Memory:* normal scratchpad memory without side-effects.
- *Command Buffer:* are analogous to (virtualized) I/O command registers, and buffer RDMA and message requests. They are monitored by command completion hardware, which posts new jobs to the *Network Job List*.
- *Queue:* such cache lines contain metadata (pointers, size, item granularity) in the free tag part, describing a queue implemented as a circular buffer. The actual queue space is allocated separately, by software, inside scratchpad areas, outside the cache line itself. Two types of queues are supported: *(i) Single Reader Queues* (many-to-one) and *(ii) Multiple Reader Queues* (many-to-many). Single Reader Queues require one head and one tail pointer and

the element size can be configured to 4-bytes, 8-bytes, 16-bytes, or a full scratchpad line. The head and tail pointers can be read via loads to specific block offsets. The Multiple Reader Queues have a fixed element size of 32-bytes and require one head pointer and two tail pointers: *(i)* a write-tail pointer for write packets (enqueues) and *(ii)* a read-tail pointer for read packets (dequeues). Incoming *write* packets (e.g. from remote store, message, or RDMA) destined to queue-type lines, are *enqueued* inside the circular buffer, and the NI controller updates the tail (or write-tail) pointer. Incoming *read* packets destined to Multiple Reader Queues record their response address in the queue body and update the read-tail pointer. Bound checking and pointer wrap-around is handled for head and tail pointers, as well as testing for queue full conditions. Matching a dequeue packet with an earlier enqueue packet (or vice-versa) is achieved by comparing the tail pointers with the head pointer and result in posting a new job in the *Network Job List*. The head pointer of Single Reader Queues is updated under software control while the head pointer of Multiple Reader Queues is updated by the NI when it completes the transfer associated with a *match* operation.

- *Counter:* these lines contain a 24-bit counter in the free tag part, and up to four notification addresses in the data part. Writes to word-offset zero increment the counter by the (signed) contents of the write. Upon reaching zero, the counter triggers the transmission of notification packets to the notification addresses by posting several jobs in the *Network Job List*.

Additionally, the NI serves incoming RDMA-Read requests. In order to meet the buffering requirements for incoming requests, without dedicating a separate memory block, we require the software to allocate a *Read Service Queue*, in the form of a Multiple Reader queue, and then assign its address to a special register.

### NI Commands and Protocol

Commands to the NI are issued as a series of stores to the data part of Command Buffer lines. Our protocol defines two types of commands: *(i)* Copy and *(ii)* Message. Copy descriptors are DMAs and have a fixed size of four 32-bit words, while messages have any size up to one cache-line (eight 32-bit words in our prototype). In order to achieve automatic command completion, every descriptor should contain its own size (in bytes) inside the word at offset zero. The first word of every descriptor contains the following fields: *(i)* 8-bits descriptor size (bytes), *(ii)* 8-bit

opcode (copy/message), *(iii)* 16-bit copy size (bytes - max 64 KBytes), used only when opcode is copy. For Copy descriptors this first word is followed by three mandatory virtual address arguments: *(a)* source, *(b)* destination, and *(c)* acknowledgment. For Message descriptors the first word is followed by two mandatory virtual address arguments –*(a)* destination and *(b)* acknowledgment– and up to five optional words that constitute the actual payload of the message. The NI uses its copy of the ART to distinguish local source addresses (write-RDMA) from remote sources addresses (read-RDMA), to validate (for protection purposes) the address arguments.

### Completion Monitor

The NI includes a monitor circuit for command buffers, and uses the descriptor size to detect completion of commands, even in the presence of out-of-order stores, but assuming single-write of each word inside the command buffer line. The monitor is activated when stores arrive to cache-lines marked as command buffers, and a bitmap of the already completed words is formed and updated. The bitmap is kept in the free tag bits of these lines and when the number of consecutive "ones" matches those implied by the descriptor size, then command completion is triggered. Upon completion, a new job description containing the address of the command buffer is posted in the *Network Job List*. Since the completion bitmap is kept in the tags of each command buffer line, interleaved command issuing is supported offering full virtualization (e.g. threads can preempted while composing a command).

### Remote Stores

Store instructions to addresses belonging to *remote scratchpad* regions (as identified by the ART), result in network packets carrying write requests, identical to RDMA or message packets (of data size 1 or more words). Stores marked as "remote" are kept in the *Remote Store Buffer*, and served by the outgoing NI engine as soon as it is free. A write-combining mechanism is implemented: if multiple remote stores to adjacent addresses arrive before some previous ones have departed, they are all coalesced in a single, multi-word-write packet. In order to support remote stores' completion notification, i.e. keep track if all remote stores have been successfully delivered, we use a special NI register that counts the total volume (in bytes) of departed remote store traffic; the acknowledgment address of remote

store generated packets is automatically set to point to this register. Every time remote stores arrive in their destination(s), acknowledgment packet(s) that contain the delivered size are sent back to the sender in order to update the NI counter. The software can check the latter counter for *zero* to ensure that all remote stores have been successfully delivered.

### Completion Notifications

We assume multi-path (adaptive) network routing, hence the multiple packets of a large RDMA may arrive out-of-order; the packet data will be written in-place, given that each of them carries its own destination address, but RDMA completion detection must now be performed by counting the number of bytes that have arrived (our network never generates duplicates). We implement counters to support RDMA completion notification. Each *session*, of one or more RDMA operations, uses one counter (allocated by software) as the acknowledgment address for its operations. The issuer decrements that counter by the total size of all RDMA transfers. Every RDMA packet carries the counter address in its acknowledgment field; upon successful write, an acknowledgment is sent to the counter and increments it by the packet size. When the counter reaches zero the NI automatically sends notification packets to its pre-configured notification addresses.

### Cache Transfer Support

The L2 cache controller issues requests for fills and write-backs by posting new job descriptions in the *Network Job List*. The job descriptions contain the appropriate opcodes and address: source address for a fill and destination address for a write-back. The outgoing NI uses the provided opcodes to format and generate the appropriate outgoing packets. The cache controller uses a Miss Status Handling Register (MSHR) structure, to keep track of outstanding write-backs and misses (transient cache-line states), and updates it appropriately when the NI serves the requests. The number of supported outstanding cache misses is limited by the number of MSHRs; we currently support one outstanding miss.

### Outgoing NI

The outgoing NI engine features a *Network Job List* in order to accept and manage requests for outgoing network operations. The sources of requests are typically the following:

- *L2 Cache Controller:* requests for write-backs and fills.
- *Completion Monitor:* explicit transfers, i.e. RDMAs and messages, when command completion is triggered for command buffers.
- *Counters:* up-to four completion notifications when a counter expires.
- *Multiple Reader Queues:* responses when enqueues and dequeues match.
- *Remote Store Buffer:* remote stores waiting in the remote store buffer.
- *Incoming NI engine:* remote acknowledgments from incoming packets.

Requests are posted in the *Network Job List* in the form of job descriptions. Each job description contains: *(i)* an opcode field that specifies how the arguments are interpreted and how the outgoing NI engine should handle the transfer, *(ii)* an address field that specifies either a local or a remote address (it may be a cacheable address, a command buffer, an acknowledgment, or a Multiple Reader Queue), *(iii)* the destination node number for the generated packet(s), *(iv)* the network priority (three available) of the packet(s) in order to avoid deadlocks of higher level protocols, e.g. cache-coherence.

Upon receiving a job description, the outgoing NI first uses the destination node number to arbitrate for the NoC (request-grant protocol). When a network slot is granted the NI proceeds to the transfer, otherwise the current descriptor is recycled and put in the back of the *Network Job List*. The latter recycling tries to avoid head-of-line (HOL) blocking, when network destinations are congested, without requiring "expensive" per-output queues (VoQs). Recycling allows us to remove the outgoing per-priority network FIFOs, since pending transfers can wait inside the *Network Job List* and the packets need not be generated.

When a network slot is granted by the NoC, the NI operates in "cut-through" mode and generates packets – along with their customized lightweight headers and CRC checksums – that belong to one of the two primitive categories: *Write* or *Read*. Cache write-backs, RDMA writes, messages, remote stores and acknowledgments belong to the *Write* category (carry data payload and acknowledgment address), while cache fills, RDMA reads and remote loads belong to *Read* category (carry the request arguments). Orthogonally to the primitive category, the packets in our prototype are sent with different network priorities as follows:

- *Low priority:* cache fills, RDMA reads, remote loads

- *Medium priority:* write-backs, RDMA writes, messages, remote stores.
- *High priority:* acknowledgments.

The payload of *Write* packets is acquired from the L2 data arrays, by iteratively reading chunks of 256-bits; the chunks are in turn serialized through the 64-bit NoC in four successive clock cycles. The NI segments large transfers, i.e. RDMA-Writes, into smaller packets when they exceed the maximum packet size (256-bytes in our prototype), or when alignment reasons dictate it. An RDMA transfer is served until it occupies a maximum network packet and then the corresponding job is recycled in the *Network Job List*; the associated command descriptor is also updated. Forcing large RDMA transfers to pause, offers fairness and reduces the latency of small packets that may wait behind large RDMAs. Moreover, the segmentation mechanism uses both source and destination addresses in order to generate packets that do not cross 256-byte boundaries. Additionally, our outgoing NI engine supports arbitrary source and destination address alignments (byte offsets) and leverages a barrel shifter to properly align and pad packets; the latter operation is only performed at the source nodes and thus the packets arrive to destinations nodes already aligned. When all packets of an RDMA transfer have been sent, the NI updates the actual command descriptor to signal local RDMA departure and allows the associated command buffer to be reused by software.

**Incoming NI**

The incoming NI exploits the header CRC contained in all packets and operates in "cut-through" mode to reduce latency. As soon as the header CRC is verified, i.e. destination address and packet size are correct, packets' payload can be safely delivered in memory without having to wait for body CRC verification; body CRC is carried in the last word of the packet. Upon reception, the NI writes the packets in per-priority network queues and notifies the incoming engine; network priorities are strictly served in descending order. The incoming NI engine gathers up-to four 64-bits words from the incoming network queues in order to create 256-bit chunks and write them at once in the wide L2 memory. The engine has first to identify whether a packet belongs to cache or scratchpad traffic, by checking the state bits of the destination address. If the destination is a cache-line waiting to be filled, then the NI delivers data in place and signals the L2 controller; only write-type packets are supported for incoming cache traffic. Write-type packets destined to lines in scratchpad space have to perform different steps according to

the type of the line. In plain scratchpad lines, data are delivered in-place and an extra write with the packet size is performed to the acknowledgment address, if non-NULL. All writes from the incoming network, generate local invalidations to the L1 cache to ensure that no stale scratchpad data remain there. Incoming write packets destined to *Counter* lines are handled in an analogous manner; only the first word is considered. If a packet is destined to a *Queue*, then the queue descriptor is accessed and the appropriate tail pointer (read-tail for read packets and write-tail for write packets) is used to enqueue the incoming packet. Read-type packets carrying a DMA request use the queuing steps, mentioned before, to enqueue in the *Read Service Queue*. Read DMA requests are handled as if they were Write DMA's from the local processor; however, a command buffer is fetched from the *Read Service Queue* pool, and a new job description is posted in the *Network Job List*.

## A.2 Hardware Cost, Latency and Software Evaluation

This section reports on the implementation cost of our FPGA prototype, presents latency figures and evaluates software operations on top of our primitives. First, we report on the total area complexity of the prototype and then we compare plain cache and scratchpad designs against our integrated Cache/Scratchpad and NI. Finally, we illustrate the latency of the primitive operations supported by our NI and present some case studies with software evaluation.

### A.2.1 Design Cost in FPGA Resources

Table A.1 presents the hardware cost of the system blocks. The numbers refer to the implementation of the design in a Xilinx Virtex-5 FPGA (XUPV5-LX110T development board) with the back-end tools provided by Xilinx. The most complex block of our NI design is the Outgoing engine which serves jobs from the *Network Job List* and implements a low latency RDMA engine that supports arbitrary byte alignments and sophisticated packet segmentation. The outgoing NI engine costs approximately 40% of the total NI LUTs and 45% of the total NI Flip-Flops. The current total design occupies less that 65% of the available LUTs and Flip-Flops in our FPGA device, however we utilize 90% of the available memory blocks (BRAMs) and thus larger caches cannot be implemented.

| **Block** | **LUTs** | **Flip Flops** | **BRAMs** |
|---|---|---|---|
| MicroBlaze + Instr. Cache | 2712 | 2338 | 4 |
| L1 + ART. | 913 | 552 | 3 |
| L2 Cntrl. + Arrays + Arb. | 1157 | 893 | 23 |
| NI Total | 5364 | 2241 | 2 |
| - Rem-Store Buff. | 398 | 312 | 0 |
| - Compl. Monitor | 223 | 62 | 0 |
| - Counters | 286 | 99 | 0 |
| - Queues | 1011 | 45 | 0 |
| - Outgoing NI | 2042 | 1015 | 1 |
| - Incoming NI | 1404 | 708 | 1 |
| Tile Total | 10146 | 6024 | 34 |
| NoC (5x5) | 2820 | 750 | 0 |
| DDR2 SDRAM Cntrl. | 3745 | 4463 | 0 |
| **Total (4x Tile)** | 47149 | 29309 | 136 |

Table A.1: Hardware Cost Breakdown in FPGA Resources

### A.2.2 Area Benefits of Integrated Cache/NI Controller

We have counted and report separately, in Figure A.3, the area complexity of three different designs: *(i)* all SRAM operating as cache only, and a cache controller; *(ii)* all SRAM operating as scratchpad only, and a NI providing DMA's; *(iii)* our configurable cache/scratchpad with its integrated NI/cache controller. The cache only design supports one outstanding miss, while serving hits under single miss, and does not support coherence. The scratchpad only design supports 8-byte aligned RDMAs and network packet segmentation.

The area here is reported in gates, to ease comparison, assuming that each LUT and each Flip-Flop is equivalent to 8 gates. The measurements do not include the L1 cache and the memory arrays. As seen, the integrated design *(iii)* has a complexity considerably lower than the sum of the complexities of the two dedicated designs, owing to several circuits being shared between the two functionalities. The circuit sharing is mostly observed on memory block datapath, the outgoing and incoming NI, and economizes 35% in hardware complexity.

### A.2.3 End-to-End Latency

Figure A.4(a) presents the latency breakdown of the following primitive NI operations: Remote-Store, Message and RDMA-Write transfers. The SW initiation cost,

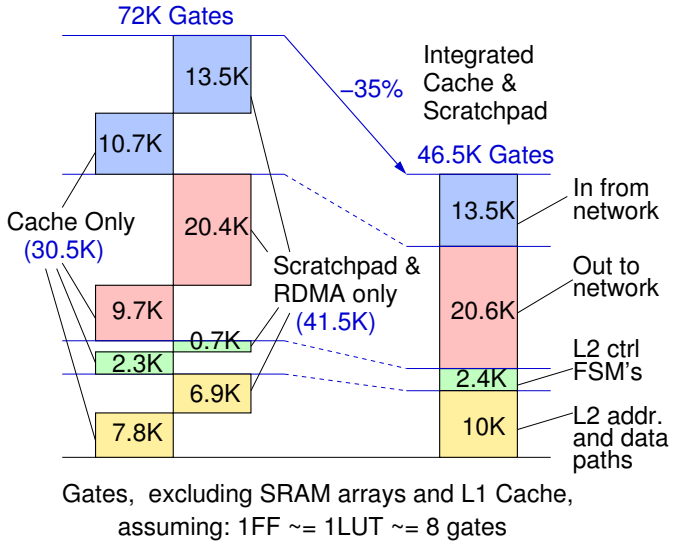
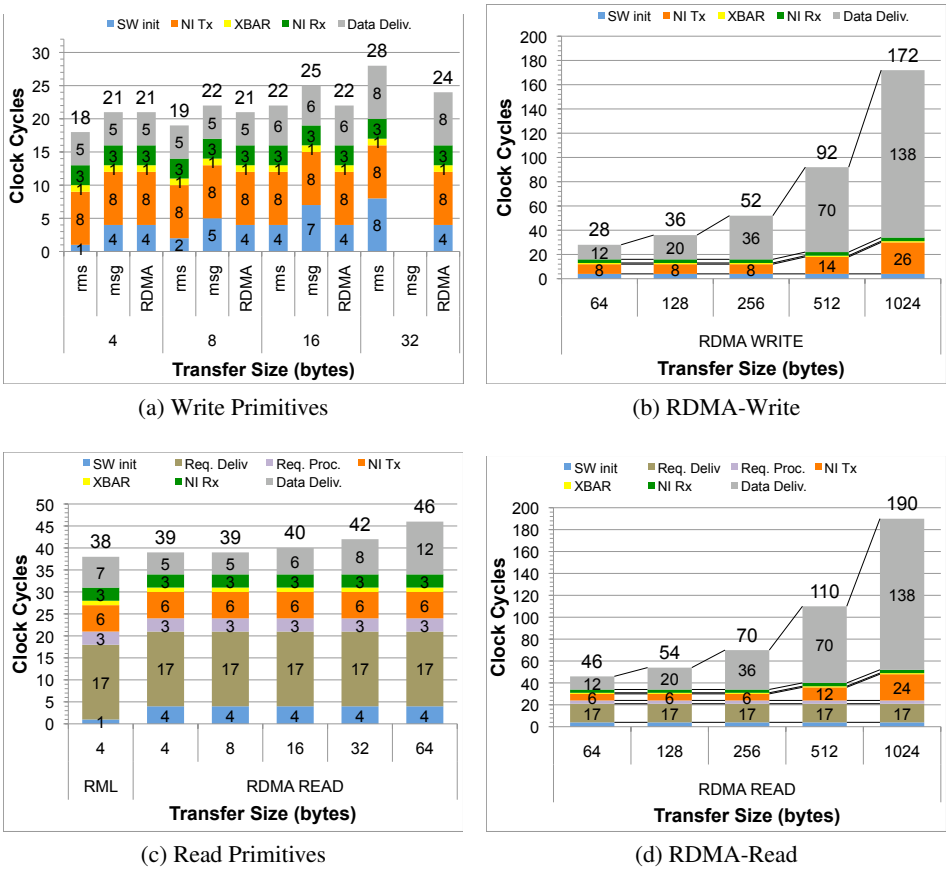



Figure A.3: Comparison of the area complexity for three separate designs: (i) Cache only, (ii) Scratchpad and RDMA-only and (iii) Integrated Cache and Scratchpad.

the NI transmit latency, the crossbar (XBAR) latency and the NI receive latency of every operation are constant under zero network-load conditions – both the outgoing and incoming path implement cut-through. The latency for the delivery of the packets' payload in the remote memory is commensurate to the size of the transfer.

Remote-Stores of 4-bytes cost 18 cycles and are faster than the equivalent messages and DMAs, since the initiation is implicit – no descriptor has to be posted. Minimum-sized messages and RDMAs of 4-bytes have the same end-to-end latency of 21 clock cycles. Although the RDMA has to read the payload from memory, and implies an extra memory access when compared with the case of a message, the outgoing NI manages to hide this extra latency during the NoC arbitration stage. For transfer sizes larger than 16-bytes RDMA achieves lower latency than remote stores and messages, however RDMA requires the packet payload to be already present in memory, thus is suitable for larger bulk transfers. The latency for large RDMAs is presented in Figure A.4(b) which shows that 64-bytes can be delivered remotely in just 28 clock cycles while 512-bytes cost only 92 clock cycles.

The NI transmit path has a latency of 8 clock cycles: 2 of them are attributed to the pipelined path to reach L2, 1 to enqueue a request in the network job-list, 1 for the outgoing NI to process the new request and 4 of them are spent on the




(a) Write Primitives

(b) RDMA-Write

(c) Read Primitives

(d) RDMA-Read

Figure A.4: Remote-Store, Message, RDMA-write, Remote-Load and RDMA-read transfers latency breakdown, as a function of data size (bytes)

NoC arbitration. The NoC request-grant phase takes 2 clock cycles but the granted network slot starts 2 clock cycles later – during that time the NI hides the latency of reading from memory and preparing packet headers. For transfer sizes that exceed the maximum network packet size, i.e. 256-bytes, and need to be segmented, an extra latency of 6 clock cycles is experienced per segment: 2 clock cycles are spent to recycle a request through the *Network Job List* and 4 clock cycles are spent again in NoC arbitration.

The NI receive path latency has two components: *(i)* the incoming cut-through latency and *(ii)* in-place delivery of packet's data in the memory. The incoming cut-through path has a latency of 3 clock cycles: 2 clock cycles are needed to receive the packet headers and check CRC and 1 clock cycle is needed to inform the incoming DMA engine about a new packet arrival. The incoming DMA engine, that

delivers data in-place, needs 2 clock cycles to dequeue the packet headers from the incoming network queues and the remaining latency, until the last word is delivered in memory, is commensurate to the payload size. For 32-byte packets, 4 clock cycles are need to gather a 256-bit chunk and 2 additional clock cycles are needed in order to arbitrate for the tag and memory arrays. The memory arbitration latency is overlapped with the gathering of the next packet words and thus experienced only once per packet.

Figure A.4(c,d) illustrates the latency breakdown of primitive remote read operations: Remote-Load and RDMA-Read. Besides the SW initiation cost, all remote read operations have a constant latency of delivering a request to a remote node which is 17 clock cycles – equal to delivering a packet of 8-bytes, contains the destination address for the source node. Thereafter, the request takes 3 clock cycles to be processed by the NI and be converted into an RDMA-write, as if it was initiated locally. The response latency follows the same steps with an RDMA-write and experiences the same latencies. Back at the initiator, the reception of a Remote-Load response takes an extra 2 clock cycles, when compared to and RDMA-Read, since the data should follow the L2 pipeline and be returned to the processor – RDMA-Reads are delivered in the L2/Scratchpad memory. A remote load of 4-bytes costs as low as 38 clock cycles while an RDMA -Read of the same size costs 39 clock cycles. Reading 64-bytes from a remote node costs just 46 clock cycles while reading 512-bytes takes 110 clock cycles.

### A.2.4 Case Studies: Software Use of Hardware Primitives

This subsection focuses on the use of the proposed hardware primitives by software constructs and illustrates some common cases where our primitives find use. Apart from minimizing the latency of data transfers through virtualized low-latency RDMA and remote stores, software can use our primitives to efficiently implement higher level operations such as: *(i)* Transfer Completion Notification, *(ii)* Barrier, and *(iii)* Distributed and Centralized Task/Job Dispatching.

#### Transfer Completion Notification

We study a common scenario where a producer sends data to a consumer in pre-agreed buffer space that forms a circular queue. The consumer needs to know when all data have arrived and typically a software-built protocol manages the low-level details. The use of interrupts for the reception of packets at the consumer

is prohibitive due to frequent context-switches (especially for small packets) and thus packet reception is typically triggered by checking a flag in the last word of the packet. The problem becomes harder when out-of-order networks come into picture and when the transfer size exceeds the maximum network packet size. The producer has to squeeze flags in the buffers to be transferred and the consumer needs to poll all these flags before arrival is triggered; additionally data are not contiguous in the buffer space since the flags have been injected. Our proposed solution is the use of *Counters* and the acknowledgment address offered by RDMA operations, Section 3.1.4. A counter per-buffer can be allocated at the consumer side and the producer can use its address as acknowledgment address when it issues RDMAs.

We measure the performance of these two sketched implementations in the FPGA prototype for a scenario where 10000 buffers are produced and sent with RDMA to a circular queue with 4 buffer slots at the consumer. For the measurements we vary the buffer size using the following values: *(i)* 200 bytes, *(ii)* 500 bytes, *(iii)* 1000 bytes and *(iv)* 2000 bytes; sizes beyond the maximum network packet size, i.e. 256-bytes, generate multiple RDMA segments. As illustrated in Table A.2, the HW counter approach offers up-to 33% improvement over the software-only approach; the performance gains increase with the size of the transfer since the number of RDMA segments increases.

### Barrier

It is a very common operation used by parallel programs to synchronize a number of parallel threads/tasks. The typical software implementation, for a few participating threads, involves a lock-protected memory location which is increased when each thread reaches the barrier; the last thread that reaches the barrier wakes-up all other waiting threads. In lieu of atomic instructions on MicroBlaze, we use an external hardware mutex module, provided by Xilinx, that is placed on the memory bus and allows *test-and-set (TAS) like* operations in a "fast" non-cacheable address space (SRAM). Using the hardware mutex module, we implement a sense-reversing centralized barrier.

The barrier implementation using the *Counter* primitive is straightforward: the counter is initialized with the number of threads (negative value) and each thread sets a local scratchpad address as notification address of the counter (up-to four supported). Upon reaching a barrier, increments to the counter are sent through remote stores. When the counter becomes zero, it triggers automatic notifications

to the pre-configured notification addresses. Multiple counters can be chained (counter notifies counters) to create larger wake-up trees and thus support higher number of cores in a scalable manner [105].

We measure and compare in Table A.2, the performance of the two implementations in an empty loop with 10000 back-to-back barriers, while varying the number of threads from 1 to 4. The HW counter is up-to 6.8 times faster than the equivalent lock-based implementation on 4 cores.

### Distributed and Centralized Task Dispatching

Spawning and dispatching tasks/jobs is crucial in parallel and distributed systems, thus we study two cases of task dispatching: *(i)* distributed and *(ii)* centralized. Case *(i)* refers to a set of masters that initiate tasks to specific workers (statically scheduled): each worker maintains a queue where multiple masters may enqueue tasks but only the owning worker may dequeue (many-to-one communication). Case *(ii)* refers to a central pool of tasks (queue) where multiple masters may enqueue and multiple workers may dequeue allowing for dynamic scheduling and load-balancing (many-to-many communication). The typical software implementation of *(i)* requires the masters to acquire a lock in order to enqueue a task and increase the tail pointer, while the worker may dequeue without acquiring a lock. However, in case *(ii)*, where multiple workers dequeue, a lock is also required to guard the head pointer. Our proposed solution for *(i)* is a *Single Reader Queue (SRQ)* per worker and for *(ii)* a central *Multiple Reader Queue (MRQ)*; these primitives offer atomic enqueue and dequeue operations, Section 3.1.4.

We measure and compare the performance of the software-only vs. hardware-assisted implementations is an program where each master spawns 10000 empty tasks. We vary the number of masters and workers accordingly and report the results in Table A.2. For case *(i)* the lock-based enqueue incurs an overhead, which for 1 master cannot be amortized by the task size, whereas some of the overhead is overlapped with multiple masters. The SRQ implementation performs up-to 4.9 times faster and the number of masters does not influence the task processing time. In case *(ii)*, the lock contention increases the task processing time when multiple workers serve tasks from the central queue. On the other hand, the MRQ performs very well allowing for up-to 7.7 times faster processing of tasks.

| **Transfer Completion** | | | |
|---|---|---|---|
| size | clock cycles/iteration | | improv. |
| (bytes) | SW only | HW Cnt. | percent |
| 200 | 233 | 206 | 13% |
| 500 | 552 | 449 | 23% |
| 1000 | 1084 | 831 | 30% |
| 2000 | 2152 | 1620 | 33% |

| **Barrier** | | | |
|---|---|---|---|
| cores | clock cycles/barrier | | improv. |
| # | Lock Based | HW Cnt. | factor |
| 1 | 111 | 41 | 2.7x |
| 2 | 194 | 66 | 2.9x |
| 3 | 357 | 78 | 4.5x |
| 4 | 574 | 84 | 6.8x |

| **Distributed Task Scheduling** | | | |
|---|---|---|---|
| Masters | clock cycles/task | | improv. |
| Workers | Lock Based | HW SRQ | factor |
| 1M - 1W | 199 | 40 | 4.9x |
| 2M - 1W | 152 | 40 | 3.8x |
| 3M - 1W | 151 | 40 | 3.8x |

| **Centralized Task Scheduling** | | | |
|---|---|---|---|
| Masters | clock cycles/task | | improv. |
| Workers | Lock Based | HW MRQ | factor |
| 3M - 1W | 232 | 87 | 2.6x |
| 2M - 2W | 237 | 44 | 5.3x |
| 1M - 3W | 270 | 35 | 7.7x |

Table A.2: Comparison of software-only operations vs. hardware-assisted.

# B

# Coherent RDMA

## B.1 Coherent RDMA Support

RDMA is hardware copying between two address regions. Such copying is straightforward if the address regions belong to scratchpads memories (fixed positions) but it becomes more complicated if one of them is cacheable. Data belonging to cacheable regions can lie anywhere in the memory hierarchy and are able to migrate at any given time; they move at blocks (block size = 1 cache line) of multiple words (typically 4-16). The major issue is how to locate all the different blocks that are possibly scattered throughout the system and how to orchestrate a copying to/from many numerous different locations.

Typically, DMA in the I/O space is not coherent in many contemporary platforms and the solutions are either based on operating system support or on snoop-based coherence, which is customary in bus-based systems. Nowadays, RDMA finds its application on-chip, in heterogeneous platforms like IBM Cell [18], and allows moving data to/from private local memories but also to cacheable regions and main memory. Coherence of DMA transfers in Cell is handled by the Element Interconnect Bus (EIB) [106], since its global broadcast nature allows snooping to be applied. The major downside of snoop-based coherence is its limited scalabil-

ity, and thus it is not an acceptable solution. However, the scalable directory-based coherence protocols do not support or handle DMAs.

Cache transfers and RDMAs should have a consistent view of the memory and avoid fetching or using stale data; therefore Coherent-RDMA is a necessity and should be supported in the preferred directory-based coherent systems. We assume that at least one of the two regions participating in the copy (RDMA) operation is scratchpad, i.e. not both are cacheable; DMA engines are attached next to each scratchpad memory and are used to serve their data transfer requirements, they are not an offloading mechanism for data copying between cacheable regions [107].

### B.1.1 Requirements Analysis of Coherence Support for RDMAs

This section studies the coherence protocol requirements in order to support data transfers between cacheable and private (scratchpad) space. We first examine the steps followed by coherent caches and the steps followed by DMA engines and then we propose step-by-step modifications to support coherent DMAs.

For the next sections, we assume a system with private L1 and L2 caches, a shared SNUCA L3 cache with distributed banks and their associated home directories. For the directory-based coherence protocol we assume a MESI protocol similar to that of [108].

#### Cacheable Reads under Directory-Based Coherence

A typical cache, during a read miss, sends a read request message to the appropriate home directory (according to some bits from the block address), and then waits for a data response. The message contains the requested block address, the opcode (get shared-GETS) and the NodeID to be used for the response. Sending a GETS request, involves the allocation of a Miss Status Handling Register (MSHR) to handle all the intermediate (transient) protocol states and other corner cases that might occur, e.g. NACKs, races. According to the state kept in the directory for that block, a series of actions is performed: *(i)* if the block is not-present(NP) then data are fetched from an off-chip memory, sent back to the cache, and the state is set to Shared(S) while the share vector is updated, *(ii)* if the block is in Shared(S) state, then a copy of the block is sent back to the cache and the share vector is updated, *(iii)* if the block is in Exclusive(E) state then a FWD_GETS request is sent to the cache that has the most up-to-date copy in order to downgrade while the status in the directory is set to shared and the share vector is updated.

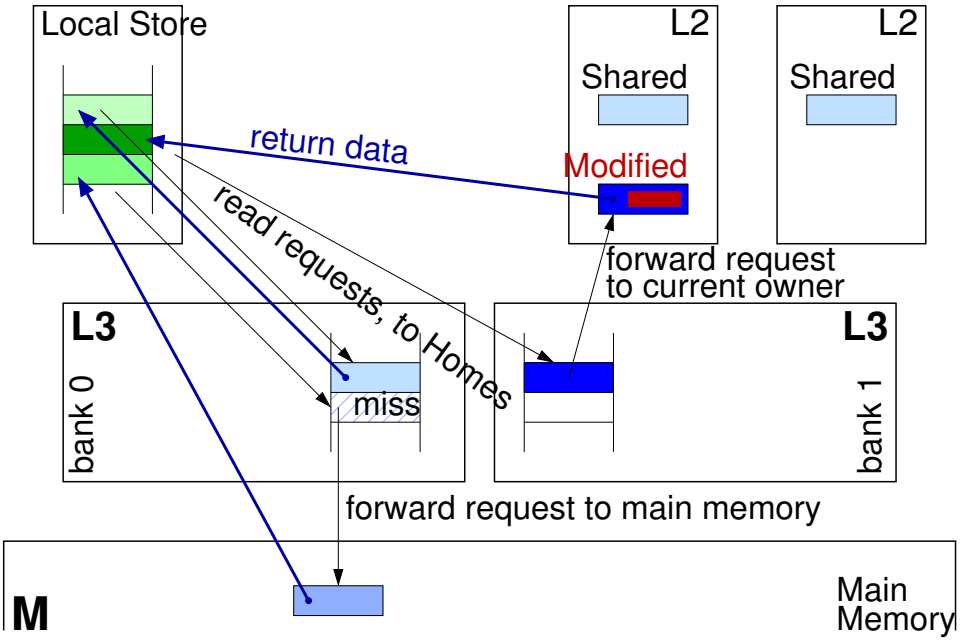



Figure B.1: RDMA from Cacheable Region to Scratchpad

**RDMA Reads between Scratchpad Regions**

As far as a DMA controller is concerned, the steps for an RDMA read involve data transfers between private address spaces hosted in scratchpad memories. The DMA engine sends a read request message to the node responsible for the related memory addresses (the NodeID of the remote node is usually found through a routing table or some bits from the DMA source address) and waits for the data to be sent back with an acknowledgment. The message sent, includes the source address of the data, the destination memory address that belongs to the requesting node, the opcode (READ) and the DMA size, an additional acknowledgment address of a completion counter is provided. The remote node responds with data in the form of one or more RDMA-Write(WRITE) packets in case DMA size exceeds the maximum NoC packet size.

**Coherent RDMA Reads: from Cacheable Region to Scratchpad**

This subsection describes the steps proposed to achieve coherent RDMA reads in a system with directory-based coherence, the flow of data is illustrated in Figure B.1. First, the DMA engine should be able to identify that the source address of a DMA command belongs to a cacheable region (consult the ART for this information),

and then send read request(s), i.e. RDMA-Read packet(s) with READ opcode, to the appropriate home directory. The DMA engine is required to split an RDMA-read request that spans cache block boundaries into multiple smaller requests (of at most one block size), since directories keep state per cache block and it is difficult to handle requests that affect multiple blocks. The fact that RDMA-reads copy data to private spaces, allows the directory and caches to be agnostic of existent scratchpad copies, therefore do not need to track sharing and thus simplify the integration. Moreover, the directory should be able to handle RDMA-read packets and perform the following operations depending on the requested block state:

*Not Present (NP):*

If the block is Not-Present(NP) in the directory tables, i.e. it is not present in any cache, then the request should be forwarded to the appropriate off-chip memory controller (memory channel). Directories do not need to create and keep any state (e.g. sharing) for such blocks. The off-chip memory controller should handle RDMAs as in the normal non-coherent case, i.e. serve RDMA-read requests and respond with RDMA-write packets directly to the requesting node. Moreover, that controller should be able to perform sub-block reads if the requested size is less than a cache block.

*Shared(S) or Present(P):*

If the block is present possibly in Shared(S) state, the directory should forward the request to an appropriate cache to provide the data response. The directory should not mark a scratchpad as a sharer. An appropriate cache is considered the next-level cache (L3 or an L3 slice responsible for that block), usually placed next to the directory (or directory bank), or any cache(L2) that is owner of the block (proximity aware selection would improve latency and limit traffic). All caches should be able to serve RDMA-read requests and respond with RDMA-write packets directly to the requesting node, even for sub-block requests.

*Exclusive(E):*

In case the block is held Exclusive(E) by a single cache, then the request should be forwarded to that cache in order to provide the most up-to-date data. The directory state should not change to Shared(S) and the cache should not downgrade or write-back if the data were Modified(M) locally. The cache should respond to a READ as in the case of a FWD_GETS to a Shared(S) block.

### Cacheable Writes under Directory-Based Coherence

Writing data through caches, typically involves fetching an exclusive copy of the cache block first and then modifying it. Fetching the old cache block for writing is achieved either with a get-exclusive (GETX) request if the block is not present in the cache or an upgrade(UPGD) request if it has been read before and kept in Shared(S) state. A GETX or UPGD request instructs the directory to invalidate all the current sharers if the block is in Shared(S) state. The directory sends invalidation messages, waits for all ACKs, updates the state to Exclusive(E) and in the case of a GETX, sends the data back to the requesting cache. When the block gets into Exclusive(E) state of if it was already in it, then a FWD_GETS request is sent to the sole cache that has the most up-to-date copy in order to provide the data. If the block is not-present(NP), then it is fetched through an off-chip memory controller and brought in Exclusive(E) state.

### RDMA Writes between Scratchpad Regions

The DMA engines perform RDMA Writes by generating RDMA Write(W) data packets carrying the instructed payload and destined to a remote destination address. In case the DMA size exceeds the maximum NoC packet size then multiple packets are generated by carefully modifying the destination memory address in order to put data in place. The NodeID of the remote node is usually found through a routing table or some bits from the DMA destination address. Every RDMA packet carries an acknowledgment address of a counter that is used to trigger the completion of transfer.

### Coherent RDMA Writes: from Scratchpad to Cacheable Regions

This subsection describes the steps proposed to achieve coherent RDMA writes in a system with directory-based coherence, the flow of data is illustrated in Figure B.2. First, the DMA engine should be able to identify that the destination address of a DMA command belongs to a cacheable region (consult the ART for this information), and then send data packet(s), i.e. RDMA-Write packet(s) with WRITE opcode, to the appropriate home directory. The DMA engine is required to split an RDMA-write command that spans block boundaries into multiple packets (of at most one block size), since directories keep state per cache block and it is difficult to handle data that affect multiple blocks. The fact that RDMA-writes push data suddenly from private spaces into a shared coherent space is unusual – caches

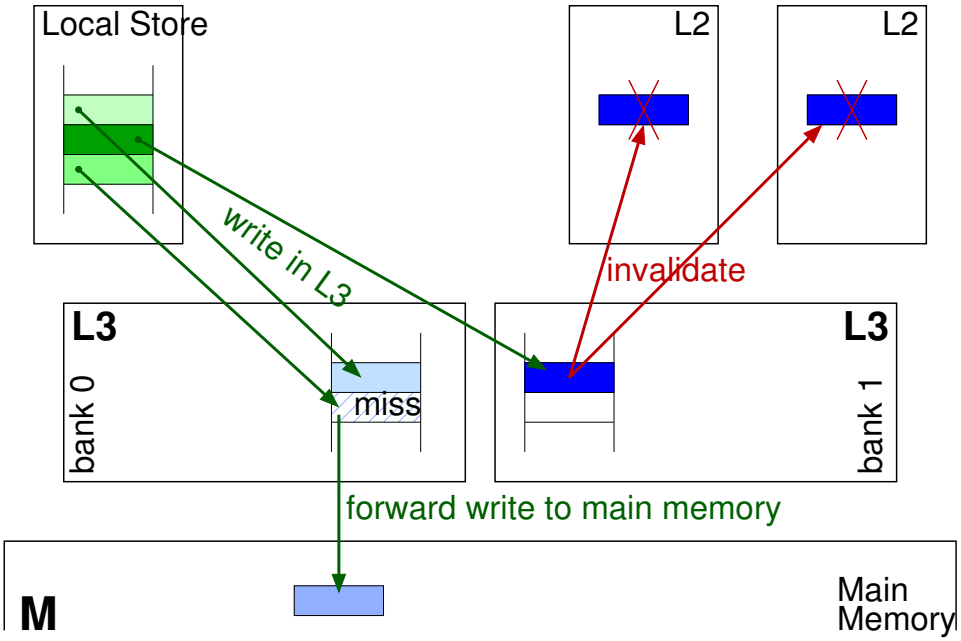



Figure B.2: RDMA from Scratchpad to Cacheable Region

always get exclusive ownership before writing – but can be handled with proper support. The directory should be able to recognize RDMA-Write packets and perform the following operations depending on the affected block state:

*Not Present (NP):*

If the block is Not-Present(NP) in the directory tables, i.e. it is not present in any cache, then the data should be forwarded to the appropriate off-chip memory controller (memory channel). Directories do not need to create and keep any state (e.g. modified) for such blocks. However, the off-chip memory controller should be able to handle sub-block writes and send acknowledgments, to an address contained in the packet, in the form of RDMA-Write.

*Present(P):*

If the block is present in the next-level cache then it should be written and the state should be set to Modified(M). Sub-block writes and acknowledgments should be supported.

*Shared(S):*

If the block is present in Shared(S) state, then the directory should invalidate

all sharers, wait for the invalidation acknowledgments and forward the writing of data to an appropriate cache. The directory should only mark the block as modified(M) without keeping any ownership information. An appropriate cache here is considered the next-level cache (L3 or a slice responsible for that block), usually placed next to the directory (or directory bank). Caches should be able to handle sub-block writes and send acknowledgments, to an address contained in the packet, in the form of RDMA-Write.

*Exclusive(E):*

In case the block is held Exclusive(E) by a single cache, then an invalidate request should be sent to that cache in order to write-back data, if they were modified(M) locally, or send a clean ack, without data. Thereupon, the directory is allowed to perform the RDMA-write and send an acknowledgment. The incoming RDMA could affect the complete block, so the possible write-back from the original cache is useless and in order to optimize, a special no-write-back invalidation should be used. If the incoming RDMA-Write is affecting a sub-block, then we either have to wait for the possible write-back and then modify the part instructed by the RDMA-write or perform the RDMA-Write immediately and request a special partial write-back.

Another more favorable option, that easily unifies all the above cases and simplifies integration is to forward the RDMA-Write to the sole owner, force it to update the block contents in place and set the local state to Modified(M); the latter choice works for all cases, either full or sub-block writes.

### B.1.2 Differences Between Coherent Caches and DMA Engines

Studying carefully the assumptions for the operation of caches using a directory-based protocol and the assumptions of the DMA engine there are some key differences. The different assumptions are sorted below:

*Data transfer size:*

RDMA-read requests and RDMA-write packets specify the size of the data transfer explicitly while cache requests always imply data transfers of cache block size. The latter observation entails that an RDMA might specify a size less than cache block size but might also specify a size larger than a cache block that affects multiple cache-lines. There are also combinations of sizes less than a cache-line and arbitrary addresses that affect multiple cache blocks.

*Request Addresses:*

Cache Writes always use an address that is aligned to cache block boundaries or to half-block if the next-level cache block size is larger (e.g. L2 cache-lines can be 64-bytes while L3 cache-lines can be 128-bytes). Cache read requests are allowed to have an address with arbitrary word offset in order to support critical-word-first. In the case of DMAs, either read or write, there is no restriction in address alignment, meaning that an arbitrary byte offset can be used.

*Sharer Tracking:*

In order to preserve coherence, the directory keeps track of the sharers that have copies of each cache block. On the other hand this is not the case for RDMAs since there cannot be any marking of copies that need to be purged; copies are private and sharing is achieved under software control.

*Writing Exclusion:*

Data writing by caches entails an exclusive ownership of a block and the directory enforces this exclusion. Getting an exclusive ownership presumes that the requesting cache gets a copy of the old data first. In the RDMA paradigm, data can be written to any destination address without reading them before and without accounting any ownership; software explicitly controls transfers and is responsible to enforce and protect any required ownership policies.

*Packet Format:*

The packet format used by the coherence protocol for requests and data transfers contains an opcode (selecting from a rich set: GETS, GETX, PUTX, UPDG, DOWG, FWD_GETS, FWD_GETX, ACK, INV, DATA, DATAX), an address, a source node id, a destination node id and a data payload in case of data transfers. On the other hand RDMAs have only two packet types (READ and WRITE), carry up to three address fields (destination address for WRITEs, additional SOURCE address for READs and ACK address) and a DMA size field.

*Packet Destinations:*

All cache requests and data transfers are sent to the home directory, which always introduces a level of indirection, while RDMA packets are always sent point-to-point.

### B.1.3 Proposed Modifications in Support of Coherent RDMA

This section summarizes the proposed modifications in each participating module, i.e. DMA engine, directory, cache, in order to support Coherent RDMA.

#### Modifications in DMA Engines

In order to support RDMA transfers from/to cacheable regions, the DMA engines should be able to identify a region as cacheable, this can be achieved by checking the outgoing ART table which is already checked for access permissions on every transfer. Additionally, the packets generated by the RDMA requests, either read or write, should be split in several smaller ones so as not to cross cache block boundaries; requires careful segmentation based on the source/destination address offsets and DMA sizes. The fact that RDMAs can read/write from/to cacheable regions requires memory barriers (fences) to include them in their wait-set, the latter can be achieved using our completion detection counters.

#### Modifications in the Directory and Coherence Protocol

The directory should be able to identify READ and WRITE type packets that are sourced from DMA engines and should forward them to the appropriate destination. All RDMA-Read packets should be forwarded to the closest cache that keeps the data if the block is in Shared(S) state, or to the sole cache that holds the most up-to-date copy if the block is in Exclusive(E) state, or to off-chip memory controller if the block is Not-Present(NP) in the cache hierarchy. Upon receiving RDMA-Write packets, the directory should invalidate all possible sharers and forward packets to the next-level cache if the block is in Shared(S) state, forward packets to the sole owner cache(L2) if the block is in Exclusive(E) state, or set state to Modified(M) and forward packets to the next-level cache(L3) if the block is present but not cached. Finally, all packets should be forwarded to the off-chip memory controller if the block is Not-Present(NP).

#### Modifications in Caches

The caches should be able to accept READ and WRITE packets and generate data responses in the form of RDMA-write packets. Incoming RDMA-read packets should generate RDMA-Write data responses and incoming RDMA-Write packets should generate acknowledgments that effectively follow the RDMA-Write format.

Additionally, the caches should be able perform sub-block reads and writes of an instructed size. The latter size affects only sequential bytes inside the same block and does not exceed block size. The provided addresses can be of arbitrary alignment. Moreover, they should support cache block updates on exclusively owned blocks and update state to Modified(M).

### Open Issues in Coherent RDMA Support

One of the open issues for Coherent RDMA Support is handling the negative acknowledgments (NACKs) that may be sent by home directories. Several directory implementations make use of NACKs in some special cases when a request cannot be handled immediately or when they run out of buffer space. NACK-free directories eliminate most NACKs but a small number of residual cases have significant complexity and cost [109]. One possible solution for such cases is to force the directory to piggyback the original RDMA packets inside the NACKs. The latter option allows the DMA engines to replay the requests/packets without keeping any state at the source; there is no need to have MSHR like structures to handle corner cases.

# Bibliography


[1] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. Leblanc. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 98, 1974.

[3] Chuck Moore. Data processing in exascale-class computer systems - keynote. In *The Salishan Conference on High Speed Computing*, 2011.

[4] ITRS. The International Technology Roadmap for Semiconductors: 2012 Update. http://www.itrs.net/Links/2012ITRS/2012Chapters/2012Overview.pdf.

[5] Steve Keckler. Life after dennard and how i learned to love the picojoule - keynote. In *Proceedings of the 44rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '44, 2011.

[6] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, 2011.

[7] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power limitations and dark silicon challenge the future of multicore. *ACM Trans. Comput. Syst.*, 30(3):11:1–11:27, August 2012.

[8] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power challenges may end the multicore era. *Commun. ACM*, 56(2):93–102, February 2013.

[9] Samuel H. Fuller and Lynette I. Millett. Computing performance: Game over or next level? *Computer*, 44(1):31–38, 2011.

[10] US Department of Energy (DoE). *Scientific Grand Challenges: Crosscutting Technologies for Computing at the Exascale*. 2010.

[11] Computing Community Consortium. 21st century computer architecture: A community white paper. 2012.

[12] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[13] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. *High Performance Computing for Computational Science – VECPAR 2010*, 6449:1–25, 2011.

[14] Micron Technology, Inc. Hybrid Memory Cube. http://www.micron.com/products/hybrid-memory-cube.

[15] Nicholas P. Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua Fryman, Ivan Ganev, Roger A. Golliver, Rob Knauerhase, Richard Lethin, Benoit Meister, Asit K. Mishra, Wilfred R. Pinfold, Justin Teller, Josep Torrellas, Nicolas Vasilache, Ganesh Venkatesh, and Jianping Xu. Runnemede: An architecture for ubiquitous high-performance computing. In *Proceedings of the 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 198–209, 2013.

[16] B. Dally. Power, programmability, and granularity: The challenges of exascale computing - keynote. In *IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 878–878, 2011.

[17] Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar. *Multi-Core Cache Hierarchies*. Morgan and Claypool Publishers, 1st edition, 2011.

[18] J. A. Kahle et al. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Develeopment*, 49(4/5):589–604, 2005.

[19] NVIDIA Corporation. Nvidia's next generation cuda compute architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[20] K. Fatahalian, T.J. Knight, M. Houston, M. Erez, D.R. Horn, L. Leem, J.Y. Park, M. Ren, A. Aiken, W.J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proc. ACM/IEEE Conf. on Supercomputing (SC)*, Florida, 2006.

[21] Scott Schneider, Jae-Seung Yeom, Benjamin Rose, John C. Linford, Adrian Sandu, and Dimitrios S. Nikolopoulos. A comparison of programming models for multiprocessors with explicitly managed memory hierarchies. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 131–140, 2009.

[22] George Tzenakis, Konstantinos Kapelonis, Michail Alvanos, Konstantinos Koukos, Dimitrios S. Nikolopoulos, and Angelos Bilas. Tagged procedure calls (tpc): Efficient runtime support for task-based parallelism on the cell processor. In *Proceedings of the Fifth International Conference on High-Performance Embedded Architectures and Compilers (HIPEAC)*, volume 5952, pages 307–321, January 2010.

[23] Judit Planas, Luis Martinell, Xavier Martorell, Jesús Labarta, Rosa M. Badia, Eduard Ayguadé, and Alejandro Duran. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[24] J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proc. IEEE Conf. on Cluster Computing*, CLUSTER '08, pages 142 –151, 2008.

[25] Matthew D. Allen, Srinath Sridharan, and Gurindar S. Sohi. Serialization sets: a dynamic dependence-based parallel execution model. In *Proc. of the ACM Symp. on Principles and Practice of Parallel Programming*, PPoPP '09, pages 85–96, 2009.

[26] Hans Vandierendonck, Polyvios Pratikakis, and Dimitrios S. Nikolopoulos. Parallel programming of general-purpose programs using task-based programming models. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, HotPar'11, pages 13–13, 2011.

[27] James Christopher Jenista, Yong hun Eom, and Brian Charles Demsky. Ooojava: software out-of-order execution. In *Proceedings of the 16th ACM sym-*

*posium on Principles and practice of parallel programming*, PPoPP '11, pages 57–68, 2011.

[28] Micah J. Best, Shane Mottishaw, Craig Mustard, Mark Roth, Alexandra Fedorova, and Andrew Brownsword. Synchronization via scheduling: techniques for efficiently managing shared state. In *Proc. of the ACM Conf. on Programming Language Design and Implementation*, PLDI '11, pages 640–652, 2011.

[29] Manolis Katevenis, Vassilis Papaefstathiou, Stamatis Kavadias, Dionisios Pnevmatikatos, Federico Silla, and Dimitrios Nikolopoulos. Explicit communication and synchronization in sarc. *IEEE Micro*, 30(5):30–41, 2010.

[30] G. Kalokerinos, V. Papaefstathiou, G. Nikiforos, S. Kavadias, M. Katevenis, D. Pnevmatikatos, and Xiaojun Yang. Prototyping of a configurable cache/scratchpad memory with virtualized user-level RDMA capability. *Transactions on High-Performance Embedded Architectures and Compilers (Transactions on HiPEAC)*, 5(3), 2010.

[31] G. Kalokerinos, V. Papaefstathiou, G. Nikiforos, S. Kavadias, M. Katevenis, D. Pnevmatikatos, and Xiaojun Yang. FPGA implementation of a configurable cache/scratchpad memory with virtualized user-level RDMA capability. In *Proc. IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS2009)*, pages 149–156, July 2009.

[32] Vassilis Papaefstathiou, Manolis G.H. Katevenis, Dimitrios S. Nikolopoulos, and Dionisios Pnevmatikatos. Prefetching and cache management using task lifetimes. In *Proceedings of the 27th ACM International Conference on Supercomputing*, ICS '13, pages 325–334, 2013.

[33] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Trans. Parallel Distributed Systems*, 20(3):404–418, 2009.

[34] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. of the ACM Symp. on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, 1995.

[35] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[36] Intel Corporation. Intel Threading Building Blocks (TBB). http://www.threadingbuildingblocks.org.

[37] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Handling task dependencies under strided and aliased references. In *Proc. of the ACM Int. Conf. on Supercomputing*, ICS '10, pages 263–274, 2010.

[38] George Tzenakis, Angelos Papatriantafyllou, John Kesapides, Polyvios Pratikakis, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. BDDT: block-level dynamic dependence analysis for deterministic task-based parallelism. In *Proc. of the ACM Symp. on Principles and Practice of Parallel Programming*, PPoPP '12, 2012.

[39] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66–77, 2012.

[40] Hans Vandierendonck, George Tzenakis, and Dimitrios S. Nikolopoulos. A unified scheduler for recursive and task dataflow parallelism. In *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, PACT '11, pages 1–11, 2011.

[41] Robert Bocchino et al. A type and effect system for deterministic parallel java. In *Proc. of the ACM Conf. on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116, 2009.

[42] K. Mai, T. Paaske, N. Jayasena, R. Ho, W.J. Dally, and M. Horowitz. Smart Memories: a Modular Reconfigurable Architecture. In *Proc. of the 27th International Symposium on Computer Architecture (ISCA)*, 2000.

[43] K. Sankaralingam, R. Nagarajan, R. Mcdonald, R. Desikan, S. Drolia, M.S. Govindan, P. Gratz, D. Gulati, H. Hanson, Changkyu Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S.W. Keckler, and D. Burger. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *Proc. of the IEEE/ACM Intl. Symposium on Microarchitecture (MICRO)*, 2006.

[44] P. Ranganathan, S. Adve, and N.P. Jouppi. Reconfigurable caches and their application to media processing. In *ISCA'00: Proceedings of the 27th International Symposium on Computer Architecture*, pages 214–224, 2000.

[45] J. Kubiatowicz and A. Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proc. of the ACM Intl. Conf. on Supercomputing (ICS)*, Tokyo, 1993.

[46] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. *ACM SIGOPS Oper. Syst. Rev.*, 28(5):38–50, 1994.

[47] S. Mukherjee, B. Falsafi, M.D. Hill, and D.A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proc. of the 23rd International Symposium on Computer Architecture (ISCA)*, 1996.

[48] G.T. Byrd and B. Delagi. Streamline: Cache-Based Message Passing in Scalable Multiprocessors. In *Proc. of the Intl. Conf. on Parallel Processing (ICPP)*, 1991.

[49] M.J. Byrd, G.T. Flynn. Producer-Consumer Communication in Distributed Shared Memory Multiprocessors. *Proc. of the IEEE*, 87(3):456–466, March 1999.

[50] R.A.F. Bhoedjang, T. Ruhl, and H.E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, 1998.

[51] M. Wen, N. Wu, C. Zhang, Q. Yang, J. Ren, Y. He, W. Wu, J. Chai, M. Guan, and C. Xun. On-chip memory system optimization design for the ft64 scientific stream accelerator. *IEEE Micro*, 28(4):51–70, 2008.

[52] Jayanth Gummaraju, Mattan Erez, Joel Coburn, Mendel Rosenblum, and William J. Dally. Architectural support for the stream execution model on general-purpose processors. In *Proc. of the Int. Conf. on Parallel Architecture and Compilation Techniques*, PACT '07, pages 3–12, 2007.

[53] Jayanth Gummaraju, Joel Coburn, Yoshio Turner, and Mendel Rosenblum. Streamware: programming general-purpose multicore processors using streams. In *Proc. of the Int. Conf. on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 297–307, 2008.

[54] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual International Symposium on Computer Architecture*, pages 358–368, 2007.

[55] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Stealth prefetching. In *Proc. of the Int. Conf. on Architectural support for programming languages and operating systems*, ASPLOS XII, pages 274–282, 2006.

[56] Pedro Diaz and Marcelo Cintra. Stream chaining: exploiting multiple levels of correlation in data prefetching. In *Proc. of the Int. Symp. on Computer architecture*, ISCA '09, pages 81–92, 2009.

[57] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *Proc. of the Int. Symp. on Computer architecture*, ISCA '03, pages 388–398, 2003.

[58] ARM Ltd. Cortex A9 Preload Engine. http://infocenter.arm.com/.

[59] Prabhat Jain, Srinivas Devadas, Daniel Engels, and Larry Rudolph. Software-assisted cache replacement mechanisms for embedded systems. In *Proc. of the IEEE/ACM Int. Conf. on Computer-Aided Design*, ICCAD '01, pages 119–126, 2001.

[60] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the compiler to improve cache replacement decisions. In *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques*, PACT '02, pages 199–208, 2002.

[61] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. PACMan: prefetch-aware cache management for high performance caching. In *Proc. of the IEEE/ACM Int. Symp. on Microarchitecture*, MICRO-44, pages 442–453, 2011.

[62] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proc. of the Int. Symp. on Computer Architecture*, ISCA '10, pages 60–71, 2010.

[63] Xiaocheng Zhou and Shoumeng Yan. A case for software managed coherence in many-core processors. In *2nd USENIX Workshop on Hot Topics in Parallelism*, HotPar'10, 2010.

[64] C. Fensch and M. Cintra. An os-based alternative to full hardware coherence on tiled cmps. In *IEEE 14th International Symposium on High Performance Computer Architecture*, HPCA, pages 355–366, 2008.

[65] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *Proceedings of the 36th annual International Symposium on Computer Architecture*, ISCA '09, pages 140–151, 2009.

[66] Alberto Ros and Stefanos Kaxiras. Complexity-effective multicore coherence. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 241–252, 2012.

[67] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 155–166, 2011.

[68] John H. Kelm, Daniel R. Johnson, William Tuohy, Steven S. Lumetta, and Sanjay J. Patel. Cohesion: An adaptive hybrid memory model for accelerators. *IEEE Micro*, 31(1):42–55, 2011.

[69] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In *Proc. 10th Intl. Symposium on HW/SW Codesign (CODES)*, Colorado, 2002.

[70] U.J. Kapasi, S. Rixner, W.J. Dally, B. Khailany, J.H. Ahn, P. Mattson, and J.D. Owens. Programmable Stream Processors. *IEEE Computer*, 36(8):54–62, 2003.

[71] P. Bellens, J.M. Perez, R.M. Badia, and J. Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *Proc. ACM/IEEE Conference on Supercomputing (SC)*, Tampa, Florida, 2006.

[72] M. Katevenis. Interprocessor Communication seen as Load-Store Instruction Generalization. In *The Future of Computing, essays in memory of Stamatis Vassiliadis*, Delft, The Netherlands, September 2007.

[73] E. Markatos and M. Katevenis. Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters. In *Proc. of the 2nd Symposium on High-Performance Computer Architecture (HPCA)*, 1996.

[74] E.A. Brewer, F.T. Chong, L.Tl Liu, S.D. Sharma, and J.D. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, St. Barbara, 1995.

[75] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proc. of the ACM Symp. on Principles and Practice of Parallel Programming*, PPoPP '10, pages 341–342, 2010.

[76] Spyros Lyberis, George Kalokerinos, Michalis Lygerakis, Vassilis Papaefstathiou, Dimitris Tsaliagkos, Manolis Katevenis, Dionisios Pnevmatikatos, and Dimitris Nikolopoulos. Formic: Cost-efficient and scalable prototyping of manycore architectures. In *Proc. of the IEEE Symp. on Field-Programmable Custom Computing Machines*, FCCM '12, pages 61–64, 2012.

[77] Dennis Abts, Steve Scott, and David J. Lilja. So many states, so little time: Verifying memory coherence in the cray x1. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 11.2–, 2003.

[78] Meng Zhang, Alvin R. Lebeck, and Daniel J. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 471–482, 2010.

[79] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45(1):73–83, February 2011.

[80] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[81] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[82] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *Proc. of IEEE Int. Symp. on Performance Analysis of Systems and Software*, ISPASS '09, pages 33–42, 2009.

[83] A. B. Kahng, Bin Li, Li-Shiuan Peh, and K. Samadi. Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration. In *Design, Automation & Test in Europe Conference*, DATE '09, pages 423–428, 2009.

[84] Dennis Abts, Natalie D. Enright Jerger, John Kim, Dan Gibson, and Mikko H. Lipasti. Achieving predictable performance through better memory controller placement in many-core cmps. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 451–461, 2009.

[85] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. of the ACM/IEEE Conf. on Supercomputing*, Supercomputing '91, pages 176–186, 1991.

[86] Fredrik Dahlgren and Per Stenström. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 7(4):385–398, 1996.

[87] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 24–36, 1995.

[88] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[89] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[90] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. The m5 simulator: Modeling networked systems. *Micro, IEEE*, 26(4):52–60, 2006.

[91] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the ACM Conf. on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[92] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *Comp. Arch. Letters*, 10(1):16 –19, 2011.

[93] HP Laboratories. Cacti 6.5: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. http://www.hpl.hp.com/research/cacti/.

[94] Intel Corporation. The Intel Xeon Phi Coprocessor. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[95] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proc. of the Int. Conf. on Compiler Construction*, CC '02, pages 213–228, 2002.

[96] L. S. Blackford et al. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28:135–151, 2001.

[97] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proc. of the IEEE*, 93(2):216–231, 2005.

[98] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proc. of the Int. Symp. on Computer architecture*, ISCA '00, pages 128–138, 2000.

[99] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277–1284, 1996.

[100] Stefanos Kaxiras and Margaret Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, 1st edition, 2008.

[101] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. Multi-level tiling: M for the price of one. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 51:1–51:12, 2007.

[102] Yunheung Paek, Jay Hoeflinger, and David Padua. Efficient and precise array access analysis. *ACM Trans. Program. Lang. Syst.*, 24(1):65–109, January 2002.

[103] Spyros Lyberis. Myrmics: A scalable runtime system for global address spaces. Technical Report FORTH-ICS/TR-436, Institute of Computer Science, FORTH, Heraklion, Crete, Greece, July 2013.

[104] T. Dallou and B. Juurlink. Hardware-based task dependency resolution for the starss programming model. In *41st International Conference on Parallel Processing Workshops (ICPPW)*, pages 367–374, 2012.

[105] S. Kavadias, M. Katevenis, M. Zampetakis, and D.S. Nikolopoulos. On-chip Communication and Synchronization with Cache-Integrated Network Interfaces. In *Proc. ACM Intl. Conf. on Computing Frontiers (CF'10)*, Bertinoro, Italy, 2010.

[106] Thomas William Ainsworth and Timothy Mark Pinkston. Characterizing the cell eib on-chip network. *IEEE Micro*, 27(5):6–14, 2007.

[107] John Heinlein, Kourosh Gharachorloo, Robert P. Bosch, Jr., Mendel Rosenblum, and Anoop Gupta. Coherent block data transfer in the flash multiprocessor. In *IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing*, pages 18–27, 1997.

[108] 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, pages 241–251, 1997.

[109] Mainak Chaudhuri and Mark Heinrich. The impact of negative acknowledgments in shared memory scientific applications. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):134–150, 2004.

:wq