UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA

Departament d'Informàtica de Sistemes i Computadors
Universitat Politècnica de València

Adaptive prefetch for multicores

MASTER'S THESIS

Máster en Ingeniería de Computadores

Author

Vicent Selfa Oliver

Advisors

Prof. Julio Sahuquillo Borrás
Prof. María Engracia Gómez Requena
Prof. Crispín Gómez Requena

September 5, 2014

Abstract

Current multicore systems implement various hardware prefetchers since prefetching can significantly hide the huge main memory latencies. However, memory bandwidth is a scarce resource which becomes critical with the increasing core count. Therefore, prefetchers must smartly regulate their aggressiveness to make an efficient use of this shared resource.

Recent research has proposed to throttle up/down the prefetcher aggressiveness level, considering local and global system information gathered at the memory controller. However, in memory-hungry mixes, keeping active the prefetchers even with the lowest aggressiveness can, in some cases, damage the system performance and increase the energy consumption.

This Master's Thesis proposes the ADP prefetcher, which, unlike previous proposals, turns off the prefetcher in specific cores when no local benefits are expected or it is adversely interfering with other cores. The key component of ADP is the activation policy which must foresee when prefetching will be beneficial without the prefetcher being active. The proposed policies are orthogonal to the prefetcher mechanism implemented in the microprocessor.

The proposed prefetcher improves both performance and energy with respect to a state-of-the-art adaptive prefetcher in both memory-bandwidth hungry workloads and in workloads combining memory hungry with CPU intensive applications. Compared to a state-of-the-art prefetcher, the proposal almost halves the increase in main memory requests caused by prefetching while improving the performance by 4.46% on average, and with significantly less DRAM energy consumption.

Keywords: Cache, Prefetch, Adaptive, Memory Hierarchy, Performance Indexes, Multicore, Deactivation Policies, Global Feedback

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents the motivation for the work developed in this Master's Thesis. First, prefetching is introduced, discussing its pros and cons and showing the potential of adaptive approaches to overcome some of the shortcomings of prefetching. After that, the need of adequate metrics for multicore evaluation is introduced. Finally, an overview about the contents of this Master's Thesis is given.

1.1 Prefetching

Addressing memory latencies is a major design concern in modern chip multiprocessors (CMPs). Prefetching is a well-known technique that hides memory latencies by fetching data blocks before the core demands them. Because of prefetching benefits, modern microprocessors [1–4] typically implement several prefetchers along their cache hierarchy. In current CMPs, memory requests from different cores compete among them for off-chip DRAM bandwidth. As the core count increases, the memory becomes a major contention point, and the system performance highly depends on how the limited memory bandwidth is managed.

In these environments the prefetching schemes must be revisited. A direct solution to increase bandwidth availability would be to turn off the core prefetchers, so avoiding speculative prefetches. However, this way could result in important performance losses for those applications where prefetchers achieve a noticeable coverage so hiding memory latency. Therefore, solutions should be focused on reducing the number of useless prefetches, saving bandwidth and energy.

Recent approaches [5] propose the use of throttling up/down mechanisms to control the prefetcher aggressiveness. The aggressiveness level is throttled down when scarce or no performance benefits are expected from prefetching. As opposite, aggressiveness is throttled up when performance improvements are foreseen.

These approaches try to keep the benefits of prefetching while reducing the memory bandwidth consumption with respect to aggressive prefetching. Nevertheless, since memory bandwidth requirements from different applications widely differ among them, this situation may lead the performance of some cores to improve but at the cost of hurting the performance of some others. To deal with fairness, the *Hierarchical Prefetcher Aggressiveness Control (HPAC)* scheme [6] takes into account *global* system performance information (e.g. memory-bandwidth interferences among the different applications at the memory controller).

In this Master's Thesis we claim that bandwidth-aware prefetchers are required in current multicores to make an efficient use of the limited memory bandwidth. This kind of prefetcher is needed to keep or enhance the performance gains of adaptive prefetchers, especially when running memory-hungry workloads.

1.2 Evaluation Methodologies for Multicores

Researchers modeling and evaluating performance and energy in multicores face three main issues, which are still evolving. First, there are multiple methodologies being used to evaluate these systems, mainly inherited from single-threaded processor research. Second, due to the nature of multiprogram workloads, new performance metrics different from those used in single-thread processors are required. Many metrics have been defined and distinct metrics are used across the published work. Finally, multicore processors are really complex systems which require from sophisticated and complementary (e.g. energy and performance) simulators.

1.3 Contributions of this Master's Thesis

The main focus of this Master's Thesis is on the design of new prefetchers to efficiently work in multicore processors both from the performance and energy points of view. These systems suffer from interferences of memory requests of the multiple cores, which can strangle the scarce main memory bandwidth. In such a case, a twofold effect rises in the system: i) the overall system performance suffers, and ii) the memory energy consumption rises. These problems exacerbate with aggressive prefetching. However, prefetching cannot be deactivated for all the cores, since it is well known that many applications benefit from prefetching.

This work presents an adaptive bandwidth-aware prefetcher designed to make an efficient use of the limited memory bandwidth, which the main contribution of this work. The devised approach tries to keep or enhance the performance gains of aggressive prefetchers while consuming a fraction of the energy and main mem-

ory traffic. Our Activation/Deactivation Prefetcher (ADP) incorporates two main mechanisms: On the one hand, as an adaptive prefetcher, it implements throttling up and down policies to regulate the aggressiveness. On the other hand, activation and deactivation policies are also implemented taking into account global system information, which is the main novelty of the proposed prefetcher. This improves the performance while reducing main memory energy consumption.

To design an efficient prefetcher we firstly characterized the dynamic behavior of each SPEC-CPU2006 benchmark across its execution time, analyzing how prefetching and main memory activity impact on performance, which is the second major contribution of this work. We found that although prefetching brings important performance improvements in some phases of the program execution, in other phases it scarcely or negatively affects the performance. Current adaptive prefetchers do not match this finding, since most of them dynamically regulate the aggressiveness but never turn off the prefetchers completely or do not consider the system main memory requirements.

Nowadays the microprocessor market is dominated by multicores, and most research work has sharply moved to this kind of processors. This quick shift of the research has left researchers with outdated research methods taken from the evaluation of their single-threaded counterparts. This new situation has lead researchers to face three important research issues targeting multicores: simulation methodologies, performance and energy metrics, and simulation frameworks. The first contribution of this Master's Thesis is a survey of the mentioned methodological issues across a sample of 28 papers published in 2013 in the top-computer architecture conferences (ISCA, HPCA, PACT, and MICRO), focusing on multicores with multiprogram workloads. For the sake of completeness, this work presents an overview of the simulation methods, discusses the performance indexes used to evaluate the proposals, and analyzes how the simulation details (used in the different system components) can affect both performance and energy results.

1.4 Master's Thesis Outline

Chapter 2 introduces a broad set of concepts tightly connected with this work. Chapter 3 presents the simulated system in which the experimental results have been obtained. Chapter 4 discuses prior work related to this Master's Thesis. Chapter 5 presents an in-depth analysis of performance metrics used in multicore research in order to determine the most suited for this work. Chapter 6 presents the characterization analysis and our proposed prefetching mechanism, and finally, Chapter 7 summarizes this thesis, discusses future work, and enumerates the related publications.

Chapter 2

Background

2.1 Multi2Sim

Multi2Sim [7] is a cycle accurate event driven simulation framework for CPU-GPU heterogeneous computing written in C. It includes models for superscalar, multithreaded, and multicore CPUs, as well as GPU architectures.

The CPU simulation framework consists of two major interacting software components: the functional simulator and the architectural simulator. The functional simulator (i.e. emulator) mimics the execution of a guest program on a native x86 processor, by interpreting the program binary and dynamically reproducing its behavior at the ISA level. The architectural simulator (i.e. detailed or timing simulator) obtains a trace of x86 instructions from the functional simulator, and tracks execution of the processor hardware structures on a cycle-by-cycle basis.

The experimental results of this Master's Thesis have been obtained using version 4.2 of Multi2Sim, which supports the execution of a number of different benchmark suites without requiring any porting, including SPEC2006, as well as custom self-compiled user code. The architectural simulator models many-core superscalar pipelines with out-of-order execution, a complete memory hierarchy with cache coherence, interconnection networks, and can be easily extended to model additional components.

A drawback of Multi2Sim is that it does not accurately model the main memory nor a memory controller. Consequently, main memory requests have no contention at all and latencies are fixed. This is not realistic and a serious limitation if evaluating a prefetcher and measuring main memory energy consumption.

To overcome this, we have integrated DRAMSim2 [8], a dedicated main memory simulator.

2.2 DRAMSim2

DRAMSim2 is a cycle accurate simulator that provides a DDR2/3 memory system model that includes ranks, banks, command queue, the memory controller, etc. and the buses by which they communicate. It provides a C++ API that can be used to integrate it in a full system simulator easily. As Multi2Sim is written in pure C, the project has been forked and extended with a C API and other requirements of this Master's Thesis.

2.3 Main Memory Organization

DRAM memories usually are presented as DIMMs, a circuit board with a handful of chips or DRAM devices and associated circuitry attached to it. Each DIMM module has several DRAM devices, which are grouped in ranks. A rank is a collection of DRAM devices that operate in lockstep (all chips in a rank respond to a single command) and read or write 64 bits to the bus, which is 64 bits wide, as defined in the JEDEC [9] standard. The number of devices that compose a rank depend of the specific device width, thats is, the number of data pins of the device. For example, a DIMM module using $\times 4$ DRAM devices, which can output four bits per transaction, will have $64/4$ devices per rank.

Internally, each of these DRAM devices implements multiple independent banks, which are the smallest memory structures that can be accessed in parallel with respect to each other (bank-level parallelism). Each bank operates independently, but banks in the same rank must comply with some timing restrictions to reduce resource utilization and thus peak power consumption. As devices in a rank operate in lockstep, when a bank is accessed it is accessed in all the devices that compose the rank, transmitting a total of 64 bits from or to the bus.

In addition, each bank is composed of multiple memory arrays, where the number of arrays is equal to the data width of the DRAM part. DRAM banks are therefore composed of two-dimensional arrays of capacitor-based DRAM cells. Each bank has a row-buffer which stores the values of the last accessed row and consists of an array of sense-amplifiers that act as latches. Therefore, subsequent accesses to the same row are faster if the row is in the row buffer. Actually, at the end of an access, the memory controller can either keep the row open in the row buffer (open-row policy) or close it if no more accesses to this row are expected (closed-row policy).

The DRAM memory controller manages the flow of data into and out of DRAM devices connected to that DRAM controller in the memory system. Specifically, it defines the Row-Buffer-Management Policy, the Address Mapping Scheme, and the Memory Transaction and DRAM Command Ordering Scheme.

A memory system can have a single channel, or multiple channels. A channel is the physical link between the processor and a set of DIMM modules that communicate through it. Multiple ranks are connected to the same channel so all receive the same commands, but only one replies. If there are multiple channels, they can be independent, with a dedicated memory controller, or configured in lockstep mode to have a wider interface (e.g. dual-channel).

Three different commands are needed to access a DRAM bank: i) a precharge command to precharge the row bitlines, ii) an activate command to open the row corresponding to the row address into the row buffer, and finally iii) a read/write command to access the row buffer at the position indicated by the column address. Depending on the row policy used by the controller and the address of the requests, very different latencies to memory can be perceived by the processor. If the row accessed is in the row buffer, the latency is low, as only a read/write command has to be issued. If the row is not in the row buffer, then the memory controller needs to issue the three commands: precharge, activate and read/write. If the controller uses a close-row policy, this means that there is no valid data in the row buffer and therefore every access has to issue an activation command and a read/write command. In all cases perceived latency is the sum of the latencies of the issued commands, as they must be executed sequentially and in order.

The performance and latency of a DRAM system using an specific row-buffer policy depends on the workloads being executed, so there is not a clear winner. It is the same for other memory controller parameters like Address Mapping Scheme, and the Memory Transaction and DRAM Command Ordering Scheme.

2.4 Prefetch Techniques

The aim of prefetching is to bring the data from the memory hierarchy closer to the processor before the processor requests it. This way the long latency associated to memory accesses can be significantly reduced. There are two key design aspects. On the one hand, the future processor accesses must be predicted so they can be fetched in advance. On the other hand, it must be decided where this data will be stored.

2.4.1 One Block Lookahead

OBL is a very basic prefetch technique which assumes a sequential access pattern. When a cache miss occurs, a prefetch request for the next block is enqueued. Variations of this technique can launch more than one prefetch, although the accuracy of this kind of prefetching is not, in general, very high. The main advantages are its simplicity, that it requires simple hardware, and the low area overhead.

2.4.2 Tagged Prefetch

TP is an improvement over OBL which adds a bit to every block that marks if it was or not prefetched. Additional prefetches are triggered if there is a hit in a prefetched block. It provides a good trade-off between performance and power/area costs [10].

2.4.3 Global History Buffer Based Prefetchers

Current prefetch techniques are based on the detection of regular memory access patterns to predict future requests. To detect these patterns, hardware tables that keep a limited number of recent memory accesses are used. As these tables are expensive both in area and power consumption, they must be implemented efficiently. Therefore, several state of the art prefetchers use a Global History Buffer table as proposed in [11].

GHB decouples table key matching from storage of prefetch related information. The resulting prefetching structure has two levels. The former is the Index Table, which is accessed with a key, like conventional prefetch tables. The key may be a load instruction PC, a cache miss address, a tag, etc. The latter is the Global History Buffer, a n-entry FIFO table, implemented as a circular buffer that holds the n most recent cache miss addresses. Each entry in the GHB table holds an address and a pointer. The pointers are used to chain the GHB entries into address lists. Each address list is a time-ordered sequence of related addresses that share an Index Table key.

Consequently, a GHB powered prefetcher can be classified according to two parameters: the method used to derive an index from an address and the pattern detection mechanism. Building on this, authors of [11] propose a regular taxonomy for this kind of prefetchers, where a prefetcher is denoted by a pair *X/Y*. Here, *X* refers to the method used to index the cache misses and *Y* the pattern detection mechanism.

Depending on the method used, *X* can be *Global* (G), *Program Counter* (PC) or *CZone* (CZ).

Global: When there is a cache miss, the offending address is stored in the GHB, and if not already added, in the IT. Subsequent misses are stored in the GHB and linked to the previous misses to the same address.

Program Counter: In this case, the IT does not store the memory address that caused the cache miss, but the PC of the instruction that triggered the cache access. The information stored in the GHB depends on the mechanism used, but usually contains the accessed address.

CZone: This technique [12] indexes accesses using a subset of the upper bits of the address. The number of bits used is implementation dependent. This way, the memory space is divided into segments or *Concentration Zones* (CZones). The pattern detection is done within each CZone. This approach is particularly appropriate in L2 caches and below, as it does not require extra information, such as the PC of the instruction that generates the memory access.

On the other hand, *Y* can be *Constant Stride* (CS), *Delta Correlation* (DC) or *Address Correlation* (AC).

Constant Stride: This technique detects sequences of accesses separated by a constant offset and launches prefetch requests following this access pattern. The number of prefetch requests depends on the aggressiveness of the mechanism, which can be tunned because the optimum aggressiveness could vary significantly between applications and also between different phases of the same application.

Delta Correlation: A delta is an offset between two addresses. This method of pattern detection [13] looks for recurring patterns in the stream of deltas obtained from the program cache accesses. When a recurring pattern is found, prefetches are enqueued following it.

Addresses	47		49		54		56		58		63		65
Deltas		2		5		2		2		5		2	

Table 2.1: Stream of deltas

Both the number of prefetches enqueued and the target size of sequences searched are implementation dependent. In the example in Table 2.1, the two most recent deltas are 5 and 2, corresponding to accesses to addresses 58, 63 and 65. If the stream of deltas is scanned for the same sequence, a match corresponding to accesses 49, 54 and 56 is found. Since the next delta is 2, (address 58) a prefetch for address 67 $(65 + 2)$ can be enqueued.

Address Correlation: This approach (AC) uses Markovian chains to predict what will be the most likely next access after a miss. When a miss occurs, a table that contains previous misses is scanned, looking for misses to the same address and the accesses that followed that miss. Using that information a Probabilistic Automaton (PA) is constructed to determine which addresses to prefetch.

2.4.4 Stream Buffers

A key prefetch design issue is where to store the prefetched blocks. They can be stored in the cache or in auxiliary dedicated buffers. One of the major drawbacks of prefetching techniques is that if the prefetches are inaccurate and are stored in the cache, they can replace useful cache blocks (cache blocks that would have been accessed in the future) without providing any benefit. This is called cache pollution. A high cache pollution can hurt performance so minimizing it is a must.

One solution is to store prefetched blocks in auxiliary buffers [14]. These buffers act as an additional memory adjacent to the cache that only contains prefetched blocks that have not been requested yet by the processor (that is, its state is still speculative and may never be accessed). When a request for a memory block arrives, the buffers and the cache are searched in parallel. If the data is found in the buffers, the block is no longer speculative so it is brought to the cache, thus avoiding the harmful effects of pollution.

This approach is specially useful for pattern-detecting prefetching mechanisms, as when a pattern is detected, it can be assigned to an exclusive buffer (called stream) where all successive prefetches that follow that pattern will be stored. Each stream is organized like a FIFO buffer that can contain several prefetched blocks, depending on the aggressiveness of the prefetcher and the maximum prefetch distance.

2.5 Coherence Protocols

When clients in a system maintain caches of a common memory resource, problems may arise with inconsistent data. A coherency protocol is a protocol which maintains the consistency between all the caches in a system of distributed shared memory. The protocol maintains memory coherence according to a specific consistency model. Older multiprocessors support the sequential consistency model, while modern shared memory systems typically support the release consistency or weak consistency models.

2.5.1 MOESI Protocol

Several approaches exist when tackling cache coherency protocols. This Master's Thesis focuses in MOESI (modified, owned, exclusive, shared, invalid), as is the one supported by the AMD64 architecture and is implemented in Multi2Sim. The states of the MOESI protocol are:

- I (Invalid): A cache line in the invalid state does not hold a valid copy of the data. Valid copies of the data can be either in main memory or another

processor cache.

- E (Exclusive): A cache line in the exclusive state holds the most recent,correct copy of the data. The copy in main memory is also the most recent, correct copy of the data. No other processor holds a copy of the data.
- S (Shared): A cache line in the shared state holds the most recent, correct copy of the data. Other processors in the system may hold copies of the data in the shared state, as well. If no other processor holds it in the owned state, then the copy in main memory is also the most recent.
- M (Modified): A cache line in the modified state holds the most recent, correct copy of the data. The copy in main memory is stale (incorrect), and no other processor holds a copy.
- O (Owned): A cache line in the owned state holds the most recent, correct copy of the data. The owned state is similar to the shared state in that other processors can hold a copy of the most recent,correct data. Unlike the shared state, however, the copy in main memory can be stale (incorrect). Only one processor can hold the data in the owned state; all other processors must hold the data in the shared state.

State transitions for the MOESI protocol are illustrated in Figure 2.1 and explained below.

- Rd/-: Local read.
- Wd/-: Local write.
- GetX: Invalidation request received (other core wants exclusive access).
- GetS: Block request coming from another core.
- Wr/GetX: Invalidation request to all the other copies of the block and local write.
- Rd/GetS: Read request to other cache in the same level or to main memory. The former leaves the block in a shared state while the latter leaves the block exclusive.

2.6 Benchmarks

To evaluate the proposals, a wide set of benchmarks from SPEC 2006 [15] suite has been used. A brief description of the benchmarks used is presented below.

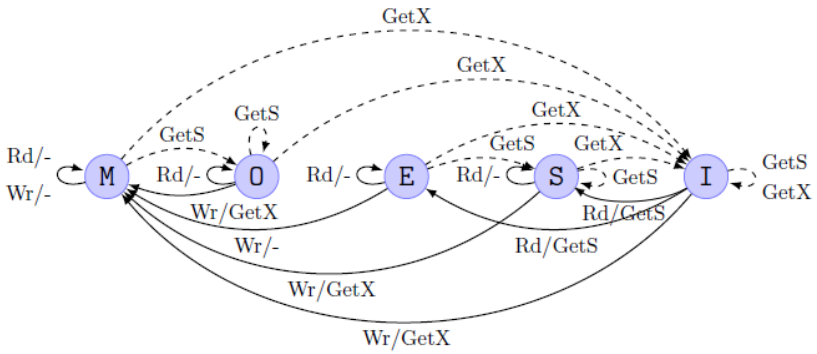


Figure 2.1: MOESI state transitions

2.6.1 Integer Point Arithmetics

perlbench: C, Programming Language. Derived from Perl V5.8.7. The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs).

bzip2: C, Compression. Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.

gcc: C, C Compiler. Based on gcc Version 3.2, generates code for Opteron.

mcf: C, Combinatorial Optimization. Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.

gobmk: C, Artificial Intelligence. Plays the game of Go, a simply described but deeply complex game.

hmmer: C, Search Gene Sequence. Protein sequence analysis using profile hidden Markov models (profile HMMs).

sjeng: C, Artificial Intelligence: chess. A highly-ranked chess program that also plays several chess variants.

libquantum: C, Physics / Quantum Computing. Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.

h264ref: C, Video Compression. A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2.

omnetpp: C++, Discrete Event Simulation. Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.

astar: C++, Path-finding Algorithms. Pathfinding library for 2D maps, including the well known A* algorithm.

xalancbmk: C++, XML Processing. A modified version of Xalan-C++, which transforms XML documents to other document types.

2.6.2 Floating Point Arithmetics

bwaves: Fortran, Fluid Dynamics. Computes 3D transonic transient laminar viscous flow.

gamess: Fortran, Quantum Chemistry. Gamess implements a wide range of quantum chemical computations. For the SPEC workload, self-consistent field calculations are performed using the Restricted Hartree Fock method, Restricted open-shell Hartree-Fock, and Multi-Configuration Self-Consistent Field.

milc: C, Physics / Quantum Chromodynamics. A gauge field generating program for lattice gauge theory programs with dynamical quarks.

zeusmp: Fortran, Physics / CFD. ZEUS-MP is a computational fluid dynamics code developed at the Laboratory for Computational Astrophysics (NCSA, University of Illinois at Urbana-Champaign) for the simulation of astrophysical phenomena.

gromacs: C and Fortran, Biochemistry / Molecular Dynamics. Molecular dynamics, i.e. simulate Newtonian equations of motion for hundreds to millions of particles. The test case simulates protein Lysozyme in a solution.

cactusADM: C and Fortran, Physics / General Relativity. Solves the Einstein evolution equations using a staggered-leapfrog numerical method.

leslie3d: Fortran, Fluid Dynamics. Computational Fluid Dynamics (CFD) using Large-Eddy Simulations with Linear-Eddy Model in 3D. Uses the MacCormack Predictor-Corrector time integration scheme.

namd: C++, Biology / Molecular Dynamics. Simulates large biomolecular systems. The test case has 92,224 atoms of apolipoprotein A-I.

dealII: C++, Finite Element Analysis. C++ program library targeted at adaptive finite elements and error estimation. The testcase solves a Helmholtz-type equation with non-constant coefficients.

soplex: C++, Linear Programming / Optimization. Solves a linear program using a simplex algorithm and sparse linear algebra. Test cases include railroad planning and military airlift models.

povray: C++, Image Ray-tracing / Image rendering. The testcase is a 1280×1024 anti-aliased image of a landscape with some abstract objects with textures using a Perlin noise function.

calculix: C and Fortran, Structural Mechanics. Finite element code for linear and nonlinear 3D structural applications. Uses the SPOOLES solver library.

GemsFDTD: Fortran, Computational Electromagnetics. Solves the Maxwell equations in 3D using the finite-difference time-domain (FDTD) method.

tonto: Fortran, Quantum Chemistry. An open source quantum chemistry package, using an object-oriented design in Fortran 95. The test case places a constraint on a molecular Hartree-Fock wavefunction calculation to better match experimental X-ray diffraction data.

lbm: C, Fluid Dynamics. Implements the "Lattice-Boltzmann Method" to simulate incompressible fluids in 3D.

wrf: C and Fortran, Weather. Weather modeling from scales of meters to thousands of kilometers. The test case is from a 30km area over 2 days.

sphinx3: C, Speech recognition. A widely-known speech recognition system from Carnegie Mellon University.

Chapter 3

Baseline System

This chapter presents the multicore processor used as baseline in this work. Figure 3.1 depicts a block diagram of the system, wich consists of four cores, Network On Chip (NoC), and a memory controller. Each core contains a processing unit, private caches and a prefetching engine. The prefetcher brings blocks from the main memory to the second level cache (L2). The NoC and the memory controller are two key system resources that are shared among cores. Both of them have been modeled in detail for the sake of accuracy in the obtained results, since they have a strong influence on the memory latency perceived by the processor on a memory access. Table 3.1 shows the configuration parameters of the core, the interconnection network and the main memory. Main memory parameters have been set according to a recent commercial MICRON DDR3 memory device [16].

3.1 L2 Prefetching System

The prefetcher modeled is the stride-based prefetcher described in [12]. The basic idea is to dynamically partition the physical address space in different zones, referred as CZones, and detect strided references within each of those zones [12,13]. Two memory references are within the same partition if their addresses have the same tag (higher order) bits. The processor sets the size of the tag by storing a mask in memory-mapped references.

A Global History Buffer [11] is used to store the tag of the currently active partitions. Additionally a stride field is added to maintain the prefetch stride. Strided references within each partition are dynamically detected by using a finite state machine (FSM). This FSM verifies whether the last three accesses are offsetted by a fixed stride. If so, the FSM assumes that a pattern has been detected and starts prefetching.

The speculative data is fetched and stored in an auxiliary buffer, called stream

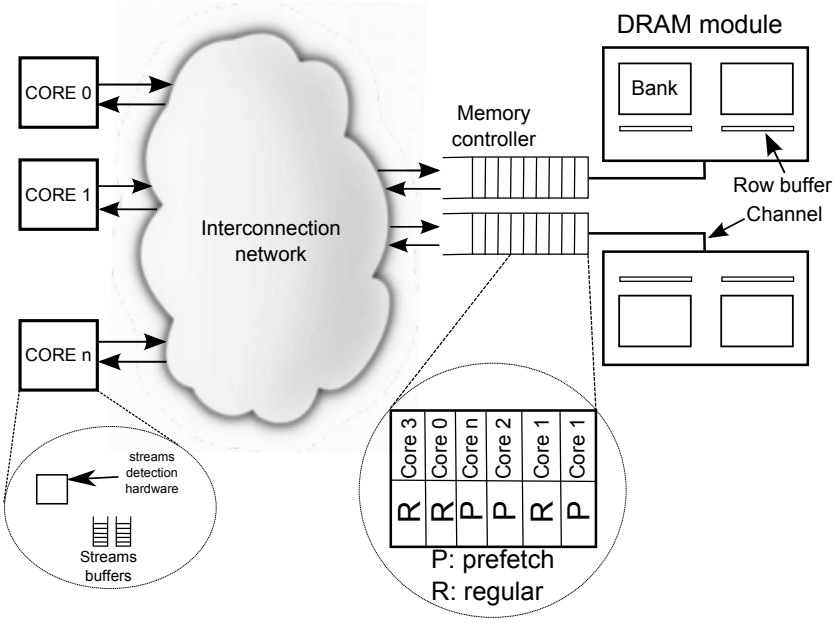


Figure 3.1: Baseline multicore with prefetching components and memory controller.

buffer. Each stream buffer tracks a different stream and consists of a tag, an available bit, and some data lines of that stream depending on the prefetcher aggressiveness. When there is an L1 cache miss, the block is searched both in L2 cache and in the stream buffers.

Depending on the application, an aggressive prefetch can significantly improve the performance or cause memory contention and degrade performance. This is why we consider the application behavior to activate or deactivate prefetching depending on the application.

More information about this prefetching mechanism can be found in Section 2.4.

3.2 Network-on-Chip

As the core count increases, the NoC is becoming one of the critical components which determines the overall performance, energy consumption, and reliability of emerging multicore systems [17–19]. The NoC connects all the on-chip components and while it improves CMP scalability, its latency also adds to the total memory access time, specially with the increasing CMP sizes. On an access that misses L1 and L2 caches, the main memory is accessed by traversing the NoC to reach the memory controller. After reading the data from the DRAM, the data

take another trip over the NoC to reach back the L2 cache. Clearly, the NoC

Processing core	
Cores	4
ISA	x86
Frequency	3GHz
Issuing policy	Out of order
Branch Predictor	bimodal/gshare hybrid: gshare with 14-bit global history + 16K 2-bit counters, bimodal with 4K 2-bit counters, and selection with 4K 2-bit counters
Issue/Commit width	4 instructions/cycle
ROB size	256 entries
Load/Store queue	64/48 entries
Cache hierarchy	
L1 Icache (private)	32KB, 8ways, 64B-line, 2cc
L1 Dcache (private)	32KB, 8ways, 64B-line, 2cc
L2 (private)	256KB, 16ways, 64B-line, 11cc, 16 MSHR
Prefetching logic	
Stream prefetcher	32 16-entry streams in L2
Aggressiveness	4 blocks
Interconnection network	
Topology	2D Mesh
Routing	X-Y
Input/ouput buffer size	128B
Link bandwidth	64B/cycle
Main memory & memory controller	
DRAM bus cycle	1066MHz
DRAM device	DDR3 (2133 Mtransfers/cycle)
Latency	t_{RP}, t_{RCD}, t_{CL} 13.09ns each
DRAM banks	8
Page size	8KB
Burst length (BL)	8
Scheduling policy	FCFS
Row Buffer Policy	Close Page

Table 3.1: System configuration.

latency plays a significant role in overall memory access latency. This proposal models a mesh topology tipically implemented in current processors, where each node consists of a core and a router, being the router connected to neighboring nodes through links in the two dimensions. Routers are 5×5, with four network ports and one internal port that connects the switch with the core. Contrary to other well-known simulators, congestion and contention are realistically modeled as key contributors of the network latency.

3.3 Memory controller and memory organization

After traversing the NoC, memory requests reach the memory controller to access the main memory. Demand requests and prefetches issued by the processor arrive to the memory controller and wait in a queue to access the memory.

In our baseline system we assume that the memory controller has a unique queue for both demand requests and prefetches. To isolate the effect of the reduction in issued prefetches the memory controller uses a closed-page row buffer policy. In addition, requests to main memory are scheduled on a first come first served basis. The number of bits and their relative order in the memory address for the address mapping used in DRAM memory accesses is shown in Table 3.2. Note that the simulated processor's maximum address space is 4GB, so addresses are 32 bits wide. In addition, the number of bits for row and column are fixed (device dependent) and the number of banks in DDR3 is 8. Since the bus used is 64 bits wide the number of bits for the byte offset is also fixed. That limits the number of ranks and banks that can be addressed. In order to simplify the design and the result analysis, the system has two ranks and one memory controller/channel.

Row	Column	Rank	Bank	Channel	Offset
15	10	1	3	0	3

Table 3.2: Main Memory Address Mapping

Both main memory organization and memory controller are key contributors to the memory latency perceived by the processor [20]. So, again, they need to be modeled in detail to obtain representative results. To this end, we have linked the DRAMsim2 [8] simulator to Multi2Sim, to have a detailed multicore simulator with an accurate DRAM and DRAM controller model. This way allows us to obtain accurate results of both performance and energy consumption.

Chapter 4

Related Work

This chapter describes relevant previous work controlling the aggressiveness of the prefetcher, the reduction on prefetch requests and other proposals focusing on multicore systems.

In [13] the AC/DC adaptive method for prefetching data from main memory to the L2 cache is proposed. Like the mechanism devised in this work, AC/DC uses concentration zones (also called CZones) [12] that divide memory into fixed size zones. The mechanism is enhanced to make use of delta correlations to find access patterns. They propose an adaptive algorithm that dynamically adjusts the prefetch degree with a range from 2 up to 16. The mechanism provides the opportunity to turn off the prefetcher but only in those cases where prefetching hurts performance, but no policy is devised to turn on the prefetcher again.

An adaptive prefetcher is also proposed in [5], which dynamically selects between five different levels of aggressiveness, ranging from very conservative to very aggressive. The baseline prefetcher is a stream prefetcher like the used in this work. Similarly to our work, the prefetcher selects at the end of each sampling interval the aggressiveness for the next interval. For this purpose, accuracy, lateness, and pollution metrics are used, throttling the aggressiveness level up or down. This mechanism was extended for multicore purposes in [6], where the cores use FDP for throttling up/down the prefetcher, but the local decision at each core can be overridden by the memory controller, which collects global information about the memory requirements of each application. Unlike our work, these proposals always keep the prefetcher enabled, and the metrics used to evaluate the prefetcher are different.

In FST [21], authors propose a throttling mechanism that limits the number of memory requests that each application is allowed to launch to the main memory. Unlike the previous schemes, both regular and prefetch memory requests are throttled. This mechanism works on global information, such as the interference that an application causes to its corunners, or the memory bandwidth each

application consumes.

A prefetcher that classifies prefetches according to their impact on performance is proposed in [22]. This impact is estimated with a history table that collects the *stall* cycles caused by each load. The mechanism prioritizes the prefetches from loads that have caused more stalls. That is, prefetcher is mainly guided by core performance instead of prefetcher performance.

In [23] a mechanism to determine at runtime the appropriate prefetcher is proposed. It evaluates the accuracy of aggressive offset prefetchers rather than actually fetching the data into the cache. To do this, each prefetch address is added to a bloom filter. Each time there is an access, the bloom filter is checked to see if its address can be found there. In this way the prefetcher accuracy is estimated. The candidate prefetchers are evaluated one at a time, in a multiplexed fashion, with the sandbox being reset in between each evaluation. The main drawback of this mechanism is that the prefetch decisions are taken local to the cores without considering global system conditions.

Prefetching performance can be also improved by enhancing the policies managing memory requests at the shared resources, that is, the arbiter at the NoC or the scheduling policy at the memory controller side. Regarding the NoC, some interesting approaches [24, 25] implement virtual channels and dynamically adjust the priority between regular and prefetch requests coming from multiple cores. These works are orthogonal to our proposal and can be applied together to further achieve performance improvements. Regarding the memory controller policies, recent proposals [26–28] have also focused on multicores. These policies take into account the prefetcher performance to dynamically select the priority of both regular and prefetch requests.

Chapter 5

Evaluation Methodologies

Multicore processors are dominating the microprocessor market and most research work has moved to this kind of processors. Multicore research methods are still immature and evolving from the single-threaded processor counterparts. Three main research issues must be faced when evaluating performance and energy in multicores.

First, multiple simulation methodologies are being applied to evaluate these systems, without being an agreement about which to use. Also, simulating these processors requires methods others than those used in monolithic processors, especially when running multiprogram workloads. These workloads consist of mixes composed of multiple benchmarks, each one running simultaneously on a different core. Unlike single-threaded or parallel workloads, the execution time of the benchmarks composing a multiprogram mix can widely differ among them. Therefore, things like determining an adequate measurement interval in order to obtain representative results are not trivial.

Second, due to the nature of multiprogram workloads, new performance metrics are required, different from those used in single-thread processors. Since multiple benchmarks are running concurrently, summarizing performance and energy is not usally easy, as they often present disparate behaviors. In fact, there is not a clear agreement but continuous contradictions rise [29, 30] about the proper metrics to be evaluated. Moreover, in some studies [31, 32] the same index name (i.e. fairness) is used to refer to distinct metrics.

Finally, multicore processors are really complex systems which require from sophisticated and complementary (e.g. energy and performance) simulators.

This chapter pursues to help researchers face the three mentioned research issues.

5.1 Single-Threaded Processor Evaluation

A simple and straightforward methodology has been typically used to compare the performance of single-threaded processors. For a single benchmark, the less the execution time, the better the performance. Thus, when evaluating a new proposal, it is common to quantify the speedup of the proposed scheme over the same machine without such modification. Studies typically use a set of applications taken from *benchmark suites*, composed of applications and kernels with different characteristics (memory intensive applications, prefetch friendly, computation intensive, etc.). Different benchmark suites are available and are selected depending on the target goal of the study. For a set of benchmarks, the performance metrics usually considered are the geometric mean of the speedups, the IPC throughput or the hamonic mean of IPCs of all the studied benchmarks, although in some cases authors interleave or present results for several statistics in the same paper for the same index. This metrics are valid because the different benchmarks are run in isolation and therefore they do not interfere with each other. The harmonic mean should be considered when the evaluated metric represents a rate (e.g. IPC or instructions per cycle) and the Arithmetic Mean when the analyzed metric is not a rate (e.g. memory latency).

As computational power of microprocessors evolved with technological advances and architectural innovations in the last decade, the benchmark suites have been updated from time to time to be representative; e.g. SPEC92, SPEC95, SPEC2000, SPEC 2006 [15]. This fact jointly with the detailed modeling of the major microprocessor components in current simulators makes the long simulation times of recent benchmarks a major concern. Even in the fastest detailed simulators, the simulation time is many orders of magnitude slower than the hardware they simulate. Simulation time can grow up to several months when executing a complete benchmark, thus, several approaches have been proposed to reduce this time.

The method commonly followed in single-thread research is simply to run each benchmark for a given number of X instructions (or cycles) after skipping the first Y millions of instructions. The values of X and Y widely differ across existing research work. This method has been typically used in spite of being known that does not bring representative results of the whole benchmark execution.

To tackle the mentioned shortcomings, some research [33] has focused on methods to identify the representative sections (simulation points) of a workload. The goal is to provide highly accurate performance estimates of the complete program execution using short simulations that start from precalculated simulation points. When multiple simpoints are used to estimate the performance of a given benchmark, it must be determined the starting point for those simpoints and the required simulation length for each one. In addition to setting the granularity,

one can also set the ceiling on the maximum number of samples to be simulated. When using simulation points, the issue of warming up the microarchitectural structures needs to be dealt with. A typical approach is the use of checkpoints although other approaches (i.e. stale state and no warm up at all) are also possible. If checkpoints are supported, then a checkpoint can be made at the start of each simulation point. This avoids fast-forwarding to each simulation point for each run of the program. In addition, all of the simulation points can be run in parallel accelerating the simulation time.

In summary, a common wrong methodology habit in most existing single-threaded processor studies is the bounded execution time that concentrates in a small fraction of time at the beginning of the benchmark execution. In addition, the use of a non-adequate mathematical mean (i.e. harmonic, arithmetic, or geometric) to evaluate a given performance index can also be observed in some research.

5.2 Methodologies for Multicores

Simulating in detail multicore processors running multiprogram workloads leads to what is known as the "simulation wall", that is, the impossibility of evaluating in detail the performance of a future many-core system running complex multiprogram applications. Usually, the simulator tool runs as a single-threaded process but simulates a multi-core architecture, say n cores. This means that the simulation takes at least $n\times$ longer than in a multi-threaded processor. Also, interactions in shared resources (e.g. the NoC, last-level caches, memory controllers, etc.) among the applications running on different cores also account in the simulation time, increasing it even more.

The advent of multicores has made researchers more concerned with methodological problems due to benchmarks presenting different behavior and execution times. This section summarizes the simulation methodologies commonly adopted in multicore research and then discusses how they have been used in 2013 top computer conferences.

5.2.1 Typical Simulation Methodologies

As mentioned above, simulating the entire execution of each benchmark composing the multiprogram workload can take several months to complete. Below we discuss a representative set of methodologies commonly used to tackle this problem, most of them generalized from their single-core processor counterparts.

Complete execution. This approach has been applied only in small benchmarks that can be fully executed. Due to the long time required to perform de-

tailed simulation of current benchmarks, only few papers (less than 6.9%) simulate benchmarks to completion. These papers use small benchmarks for evaluating specific parts of the system (e.g. STT-MRAM write energy [34]).

Fast forward plus execution. This method consists in fast forwarding the initial X instructions of each benchmark and then concurrently executing a set of Y instructions for each of them. A variant of this approach fast forwards and executes on the basis of cycles instead of instructions.

Notice that when executing different benchmarks for a target number of instructions, due to benchmarks progress at different speeds, some of them finish earlier than others. Therefore, only a subset of benchmarks is running on the system in a part of the measurement interval, what can impact the overall performance. Some researchers realized from this situation and the typical adopted solution is to relaunch or to keep executing these benchmarks. This way keeps constant the number of running threads until the slowest benchmark finishes its execution. However, statistics for a given benchmark must be gathered when it executes the target number of instructions and ignored after this point.

Fast forward plus warm up plus execution. This method consists in warming up the major microprocessor components (e.g., caches or branch predictors) after skipping the initial instructions, with the goal of achieving the same system state as that achieved with normal activity. The performance *of these components* usually improves after an initial phase, known as training phase. In this method, the time used in the warm up is not considered in the statistics.

Representative phases or simpoints. Some research [33] has focused on methods to identify the representative sections (simulation points) of a workload. The goal is to provide highly accurate performance estimates of the complete program execution using short simulations that start from precalculated simulation points. When multiple simpoints are used to estimate the performance of a given benchmark, it must be determined the starting point and simulation length required for each simpoint. In addition to setting the granularity, one can also set the ceiling on the maximum number of samples to be simulated. Although this approach has been successfully applied to single-threaded simulations, more work is required to expand their scope to multiprogram workloads. In these workloads, each benchmark has its own set of representative phases with distinct characteristics (start at different points of time and present different lengths), which should be taken into account.

5.2.2 Discussion

This section discusses a representative set of the simulation methodologies commonly used in current research work on multicore processors with multiprogram workloads (sample from the HPCA, ISCA, Micro and PACT in 2013).

As mentioned above, less than 6.9% of the papers simulate benchmarks to completion. The remaining studies (i.e. 93%) only evaluate the intervals considered to be representative of the whole benchmark. A bit less of three quarters of them (by 70.4%) only simulate a single interval of the benchmark, and the other 30% simulate at least two different slices of the benchmark to reflect distinct representative phases of the benchmark execution.

Among those papers simulating a single slice, a very low percentage (around 10%) performs detailed simulation of the initial part of the benchmark (e.g. 4 billion instructions [35]). The remaining studies fast forward the initial instructions to skip the initialization part of the benchmark trying to reach a representative *slice* of the benchmark execution. Two different approaches can be distinguished across the revised papers, those that skip the same amount of instructions for all the benchmarks and those skipping a different number of instructions for each benchmark to reach a representative simulation point of each benchmark. In addition, about one third of them (by 31.6%) warms up the major microarchitectural components after fast forwarding.

An interesting observation is that most of the studies (by 88.26%) implement some kind of fast forwarding of instructions regardless a single or multiple slices are simulated.

5.3 Evaluation Indexes for Multiprogram Workloads

5.3.1 Performance Indexes

Commonly some benchmarks run faster on one machine while other benchmarks run faster on other machine. In such a situation, depending on how we summarize the performance of the benchmark set, we might drawn distinct conclusions. Different metrics have been proposed for quantifying throughput without a full agreement on which are the best indicators of throughput in these systems. Some papers even use several of such metrics. This section gives an overview of the performance metrics that have been used in multicore research. In those cases where the same index name has been used in different papers but with distinct meanings, the multiple meanings are discussed.

Instructions per cycle (i.e. IPC) and speedup have been the performance metrics commonly used during the last decade to measure the performance of superscalar monolithic processors. Due to their basic nature, both of them have been widely used in its original way in multicore research, the IPC (e.g. [32, 35–38]) to quantify the overall system performance, and the speedup (e.g. [39–43]) to evaluate the performance increase that a given proposal achieves over a baseline machine. Equation 5.1 shows the **speedup** for benchmark i in a set of n bench-

marks.

$$S_i = \frac{IPC_{i,new}}{IPC_{i,base}} \quad \forall i \in \{1, n\} \tag{5.1}$$

Several indexes have been derived from this metric to be applied in multiprogram workloads with the aim of summarizing the performance of the program mixtures.

Cumulative IPC or IPC throughput [44], as it is named by some authors, is obtained as the sum of the IPC of all the programs executing concurrently (see Equation 5.2). This metric is frequently used to evaluate multicore performance, but it lacks any notion of fairness, so it is possible to "maximize performance" by favoring high-IPC programs, so it should be used jointly with a fairness metric.

$$IPC_{sum} = \sum_{i=1}^{n} IPC_i \tag{5.2}$$

Harmonic Mean of IPC, shown in Equation 5.3 is used in some papers [30] to analyze throughput. The reason to use HM instead of other mean is because IPC is a rate, and HM is the more appropriate mean for rates. Other reason is that as the harmonic mean tends strongly toward the least elements averaged, it can introduce the notion of fairness that IPC throughput lacks.

$$IPC_{hm} = \frac{n}{\sum_{i=1}^{n} \frac{1}{IPC_i}} \tag{5.3}$$

Arithmetic Mean of Speedups is obtained averaging the speedups (Equation 5.4) of all the individual benchmarks in a multiprogram workload. As explained below, this metric is not usually the most adequate.

$$S_{am} = \frac{\sum_{i=1}^{n} S_i}{n} \tag{5.4}$$

Geometric Mean of Speedups. This statistic mean, shown in Equation 5.5, is preferred over Arithmetic Mean when comparing ratios. Since relative performance is always a distribution and not just a number, one should use a mean adequate to the distribution of the analyzed data. Arithmetic Mean should be used when the data fit a standard distribution. Geometric Mean is appropriate when data fit a lognormal distribution. In general is not adequate to use an Arithmetic Mean on a distribution of ratios, unless the standard deviation is low, since in general the fitting is better with a lognormal distribution [45].

$$S_{gm} = (\prod_{i=1}^{n} S_i)^{\frac{1}{n}} \tag{5.5}$$

Individual Speedup. A large set of research papers consider the Individual Speedup instead of the raw Speedup. Because of some resources (e.g. last level caches) being shared among different cores, the performance of a program executed in a multicore can be strongly influenced by the co-runners (programs running concurrently with it). This metric estimates how much the performance for each individual benchmark is affected by its co-runners, by considering both the IPC of each application in the multiprogram environment and its IPC in stand alone execution. The Individual Speedup is obtained as the division of the individual IPC (of each application in the multiprogram mixture) to the IPC of that application in isolated execution [29,43,46]. Notice that the IPC of a given benchmark in the multiprogram workload is smaller than in isolated execution, thus some authors (e.g. [32]) refer to this index as **Slowdown**. Several metrics for the multiprogram workloads are derived from this index.

$$IS_i = \frac{IPC_{i,multi}}{IPC_{i,alone}} \quad \forall i \in \{1, n\} \tag{5.6}$$

Weighted Speedup is the sum of the Individual Speedups of all the workloads executing concurrently [29, 47]. This index has a similar drawback as the raw speedup. If there are some applications with high or very high weighted speedup in a mix, these values can hide the poor performance of others applications in the same mix. That is, one could obtain a high Weighted Speedup by favoring selected applications at the expense of others.

$$WS = \sum_{i=1}^{n} IS_i \tag{5.7}$$

Harmonic Mean of Individual Speedups has been used to deal with the drawbacks of Weighted Speedup (e.g. [44, 48]) because as said, the HM tends to mitigate the impact of large outliers and aggravate the impact of small ones.

$$WS_{hm} = \frac{n}{\sum_{i=1}^{n} \frac{1}{IS_i}} \tag{5.8}$$

Fairness. This metric pursues to estimate to what extent the performance enhancements are achieved favoring some benchmarks at the cost of others. A given proposal is *fair* if it provides similar performance improvements (or losses) across the multiple applications running concurrently.

With this aim, several indexes have been used in the literature referred to as either *fairness* or their complementary, *unfairness*. For instance, in [31], authors propose to use Equation 5.9 to estimate unfairness.

$$Unfairness = \frac{\max(IS_i)}{\min(IS_j)} \quad \forall (i, j) \in \{1, n\} \tag{5.9}$$

Although this metric has been used to effectively design fairness oriented memory controllers, in [32] it is claimed that this metric has a major shortcoming, since it only considers the behavior of the outliers, and does not take into account the average behavior. To deal with this shortcoming, other authors estimate fairness as shown in Equation 5.10, where σ_S and μ_S are the standard deviation and average of the weighted speedups across all individual benchmarks. This formula uses the so-called coefficient of variation (σ/μ) that measures the variability of the IS compared to the average IS.

$$Fairness = 1 - \frac{\sigma_{IS}}{\mu_{IS}} \tag{5.10}$$

5.3.2 Understanding Performance Metrics: A Practical Approach

This section pursues a twofold objective. First, to illustrate how the conclusion drawn from the performance analysis can differ depending on the metrics used for comparison purposes. Second, to provide insights about what kind of metrics is required for the study of multiprogram workloads.

To carry out the analysis we have devised an hypothetical four-core scenario discussed below. The scenario is presented in two tables and consists of four four-core machines: a baseline system and three (A, B, and C) machines to be compared. Each four-core machine runs four applications.

Table 5.1 summarizes the workload characteristics and speedups of machines A, B, and C. The *Benchmark* column identifies the evaluated benchmark; when all benchmarks run concurrently, this column also indicates the core it is assigned to. Column *Baseline IPC* shows the IPC of each benchmark both running alone (labeled as *Alone*) and running concurrently (labeled as *together*) with the remaining benchmarks. Both results are obtained with the baseline machine. IPC running alone should always be greater or equal than IPC running together. The *Evaluated Machines IPC* column depicts the IPC of the studied benchmarks running concurrently in the three evaluated machines. Column *Speedup* compares the IPC of the benchmarks running concurrently in machines A, B and C with the IPC obtained also concurrently but in the baseline machine. Finally, column *Weighted Speedup* shows the performance effect of running concurrently the benchmarks in the proposed machines with respect to the alone execution in the baseline machine.

As observed, this scenario assumes that machine A runs applications in core0 and core1 at 70% of their speed in isolated execution in the baseline machine, and applications in core2 and core3 at 30% (see column *Weighted Speedup*). System B and system C run all the concurrent applications at 50% and 30% of the speed of a single-core machine, respectively. On average, machine A and machine B run

Benchmark	Baseline IPC		Evaluated Machine IPC		
	Alone	Together	A	B	C
#0	3,00	1,00	2,10	1,50	0,90
#1	2,00	1,00	1,40	1,00	0,60
#2	1,50	1,00	0,45	0,75	0,45
#3	1,00	1,00	0,30	0,50	0,30

Benchmark	Speedup			Individual Speedup		
	S_A	S_B	S_C	IS_A	IS_B	IS_C
#0	2,10	1,50	0,90	0,70	0,50	0,30
#1	1,40	1,00	0,60	0,70	0,50	0,30
#2	0,45	0,75	0,45	0,30	0,50	0,30
#3	0,30	0,50	0,30	0,30	0,50	0,30

Table 5.1: IPCs and Speedups of the benchmarks in the 4-core compared machines.

each benchmark at around half its speed (46% and 50%, respectively) with respect to isolated execution in the baseline machine; however, machine A accelerates those having higher IPC. Note that since we are averaging IS, a ratio, we should use the Geometric Mean instead of the arithmetic one.

Table 5.2 summarizes the performance of the compared machines. This example shows that best performing machine differs depending on the considered metric, as discussed below:

- If performance is analyzed considering the Average Speedup (S_{am}), then machine A is the best performing one, since it accelerates the execution of the pair of applications with highest IPC, those running in core0 and core1, with respect to the other two machines (see Table 5.1 column *Speedup*). These values are by 13% and 89% higher than those of machines B and C, respectively.

- If performance is compared using the Geometric Mean of Speedups (S_{gm}), then machine B achieves the best performance, closely followed by machine A, and both of them clearly outperform machine C.

- If the performance is evaluated using IPC_{sum}, the best performing is A, followed by B and C, and the differences are significant.

- If IPC_{hm} is used, the best performing is clearly B, followed by A and C.

Machine	S_{am}	S_{gm}	IPC_{sum}	IPC_{hm}
A	1,06	0,79	4,25	0,59
B	0,94	0,87	3,75	0,80
C	0,56	0,52	2,25	0,48
Machine	WS	WS_{hm}	Unfairness[i]	Fairness[ii]
A	2,00	0,42	2,33	0,54
B	2,00	0,50	1,00	1,00
C	1,20	0,30	1,00	1,00

[i] Eq. 5.9 [ii] Eq. 5.10

Table 5.2: Summarizing performance and fairness metrics in the compared machines.

- If performance is evaluated with Weighted Speedup (WS), then machines A and B perform on par.
- If performance is studied using the Harmonic Mean of Weighted Speedup (WS_{hm}), machine B is the best one with $WS_{hm} = 0.5$ and machine C the worst one with $WS_{hm} = 0.3$. As mentioned above, this metric evaluates both fairness and performance. In spite of that, machine C, with very good fairness, scores very low. That is because although WS_{hm} can be used to have a rough estimation of fairness, performance performance presents a higher weight in the final result.
- Finally, if a pure fairness-oriented approach is used to evaluate the results, then machine B and C would be selected since both of them perform equally fairly, regardless the fairness metric used. Notice that both fairness metrics are consistent since machine A is about twice as unfair (Eq. 5.10) than the others, and around half as fair (Eq. 5.9) than the other compared machines.

Let us now compare the three machines for a better understanding considering the different metrics together.

Compared to B, machine A has better performance by 13% in S_{am} and by 13% in IPC_{sum}. On the other hand, machine B outperforms machine A by 10% in S_{gm}, by 36% in IPC_{hm}, and by 19% in WS_{hm}. Both machines provide the same WS WS.

When deciding which machine performs better one can choose B, arguing that both S_{am} and IPC_{sum} metrics have flaws, because the first one is only valid if the input data fit a standard distribution and the second one can hide serious fairness

problems. Other approach in order to decide is to take fairness into account, and machine B has clearly more Fairness and less Unfairness. Therefore, one can safely conclude that machine B is better.

Machine C is clearly the worst performing one since it runs all the benchmarks at the lowest studied speed. However, if it is compared to machine A with fairness metrics, then machine C should be selected over machine A, which demonstrates that using only fairness oriented metrics is not enough and would yield to wrong conclusions.

According to this example, performance analysis could use i) only WS_{hm} as it measures performance and it is affected by fairness or ii) it could combine Fairness with another metric, like WS, S_{gm} or even IPC_{sum}.

In summary, there is not a silver bullet. Multiple metrics can be used to measure performance, some of them combined, but to obtain reliable results, a researcher must know the pros and cons of the used metrics, the measured variable, and then choose wisely.

5.3.3 Discussion

This section discusses the metrics commonly used to evaluate multicores.

Regarding performance, the indexes presented in Section 5.3.1 quantify the overall system performance, but depending on the part of the system on which the proposal focuses, more concrete indexes can be used. For this purpose, we grouped the commonly used performance metrics in three main categories depending on the part of the system they evaluate: i) overall system performance, ii) memory performance, and iii) cache performance.

The total execution time of a part of the program is considered in some papers [49–51] to evaluate the performance of the entire system. However, these benchmarks are really small and only used to evaluate specific parts of the systems. For instance, in those papers tackling main memory topics.

In general, overall system performance is commonly measured using IPC-related metrics, also referred to as throughput metrics, like those presented in Section 5.3.1. The cumulative IPC is the commonly accepted metric and has been used in 14 from the 28 revised papers. To compare IPCs or execution times of different proposals, the speedup and its variants (means) are the common metric of choice. Speedup-oriented metrics are used in 18 papers and 9 of them also use the weighted speedup.

In most of the proposals the final aim is to increase or at least maintain the overall system performance. As discussed above, when evaluating the system performance also fairness must be considered to demonstrate that the proposal is not enhancing the overall performance by favoring those benchmarks with highest IPC. Although this claim is widely accepted by the scientific community, only a

few set of research papers evaluate fairness. The main reason is that most fairness definitions [31, 32] have been recently proposed.

Most of the analyzed publications use throughput-oriented metrics, focusing on raw speedup (used by 64% of the papers) over weighted speedup, which is only used in half of these works as a complementary metric to raw speedup. Both fairness definitions have been equally used across the studied papers.

An important fraction of the research work on multicores has focused on main memory, since this component becomes a major performance bottleneck in current multicores due to the limited available bandwidth. Across this work, the memory bandwidth has been used in 3 ([43,52,53]) of 5 papers focusing on memory topics, while other specific metrics like the row-buffer hit rate or normalized memory latency have been used in the other 2 papers [46, 54].

Cache performance, especially the last level cache, has become an important research topic in the multicore era since a miss in this level incurs long memory access latencies, which can severely damage the overall system performance. Research work focusing on this topic similarly uses both the MPKI (misses per kilo instructions) [41, 55] and the LLC miss rate [39, 49, 54, 56, 57].

On the other hand, the power budget is an important design constraint in current multicores since it affects energy, package, and cooling costs. Therefore, the multicore era yields designers to enhance the performance with a reasonable energy consumption. As researchers are concerned by this critical design issue, most of the papers (19 from 28) evaluate energy consumption in their proposals, either dynamic, static, or both [37, 38, 43, 49–51, 56, 58]. Unfortunately, due to the high complexity of this task and lack of adequate tools, energy consumption is usually measured taking only into account specific parts of the system, but not the full system.

5.4 Performance and Power Simulators

Researchers must select an adequate simulation platform to carry out their research, which can vary depending on the goal of study.

In an ideal simulation framework, researchers should use a single tool able to model the entire system in an accurate manner. In such a case, the obtained results are representative and less effort is required in comparison to utilizing distinct simulators, each one modeling a different subsystem.

Considering that the focus of this work is on multicores, the selected simulator should be at least able to model the core microarchitecture, the caches, the NoC, the coherence protocol, and the main memory, since these are the fundamental pillars of the system.

Several simulation frameworks have appeared during the last few years. A representative set of these tools that can be found in research papers is: Multi2Sim [7], Sniper [59], MARSSx86 [60], and gem5 [61].

Researchers often find that their simulator does not model a given subsystem accurately enough, which can provide results that may be far from the real behavior of actual systems. Looking at the sample of papers from the 2013 main conferences, all of them with the only exception of a single paper [62] accurately model the processor microarchitecture, but other system components receive less attention. By 40% of the papers use a non-detailed main memory system, even using a constant main memory access time and only 22% of them model the memory controller and the NoC in detail. As a rule of thumb, all the subsystems should be modeled in detail if overall performance metrics are evaluated. Otherwise, i.e if only the part of the system being analyzed or a subset of the full system are modeled in detail, overall performance results would not be representative. In other words, if a proposal is being evaluated, the impact of the proposal on the overall performance cannot be known in a precise and accurate way.

As the transistor features shrink with each technology generation, power consumption has become increasingly important. This has led researchers to estimate energy in addition to performance. Similarly to the rise of detailed microprocessor simulators, a wide set of power simulators has spread over the last few years. These simulators implement power models of the electronic components that estimate the consumed energy (leakage and dynamic energy) in specific subsystems. Below we list the main features of a representative subset of these simulators.

CACTI [63] estimates the energy, access time, and cycle time of RAM-based components (e.g. caches or register files). Orion [64] is a power-performance interconnection network simulator. It can be used to measure the consumption in electrical and nanophotonic networks. McPAT [65] models area, power and timing for the components of a complete chip multiprocessor, including in-order and out-of-order processor cores, networks-on-chip, shared caches, and integrated memory controllers for some specific processors. It is tightly coupled with CACTI to model memory structures. DRAMSim [8] implements detailed models for a variety of existing DRAM devices (e.g. DDR3). It has been designed to simulate a sophisticated memory system where transactions can be freely re-ordered, the address mapping scheme can be independently configured, the row buffer management policy can be independently adjusted, and DRAM refresh policies can be flexibly implemented or turned off entirely. The DRAM device and system configurations, timing parameters and power consumption parameters can all be set independently and adjusted in configuration files.

Multicore simulation infrastructures are usually feed with upon-agreed benchmarks to measure the performance, energy, and area of the system. As the focus of this work is on multiprogram workloads where multiple single-threaded pro-

Mix	Benchmarks			
m0	cactusADM	hmmer	libquantum	wrf
m1	tonto	h264ref	hmmer	omnetpp
m2	bwaves	gamess	GemsFDTD	sjeng
m3	astar	bzip2	gcc	GemsFDTD
m4	gamess	GemsFDTD	leslie3d	wrf
m5	gcc	libquantum	povray	xalancbmk
m6	milc	sjeng	tonto	xalancbmk
m7	bzip2	dealII	lbm	sjeng

Table 5.3: Four-core mixes composition.

grams run concurrently in the system, researchers must design mixes composed of several benchmarks. These benchmarks are usually chosen from the SPEC CPU suite [15].

5.5 Experimental Evaluation

The goal of this section is to experimentally illustrate how the discussed methodologies provide different performance and energy results, which do not always vary in the same way but depend on the workload characteristics.

For illustrative purposes, we considered a typical multicore system consisting of a tiled 3GHz 4-core chip multiprocessor accordingly to the machine parameters presented in Table 3.1. A set of eight mixes consisting of four SPEC2006 CPU benchmarks (see Table 5.3) has been designed to carry out the simulation study.

5.5.1 Effect of the Simulation Methodology on Performance and Energy

This section explores how performance and energy differ depending on the used simulation methodology.

Figure 5.1 depicts the cumulative IPC results that different methods provide across the studied mixes. *no FF no WU* executes 600M instructions without neither fast forwarding (FF) nor warming up (WU). *FF WU* executes 600M instructions after fast forwarding 500M instructions and then warming up 200M instructions. *FF no WU* executes 600M instructions after fast forwarding 500M instructions, but the warming phase is skipped.

An interesting observation is that no method provides always the highest IPC

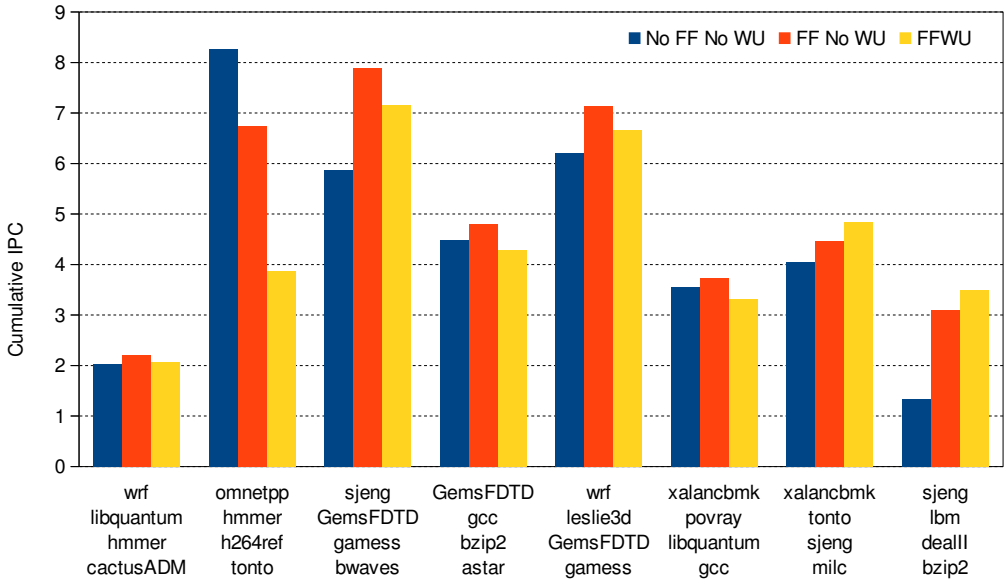


Figure 5.1: Cumulative IPC for three of the studied methods.

but it depends on the workload. For instance, the *FF WU* method presents the highest IPC for two mixes, the *FF no WU* method shows the highest IPC for five mixes, and even the *No FF No WU* shows the highest IPC in one mix. Moreover, IPC deviations between methodologies can be as much as $\pm 30\%$ depending on the workload.

A similar rationale can be applied to the remaining performance metrics. As example, Figure 5.2 presents the *harmonic mean of weighted speedup* comparing methods across the studied mixes.

On the other hand, since the execution time varies depending on the used methodology, the consumed energy will also differ. As example, Figure 5.3 shows the consumed energy (in mJ) in the main memory across the studied mixes. Again, important differences appear depending on the workload.

These results illustrate the importance of the methodology to obtain representative performance and energy results that lead the researcher to draw precise conclusions. Notice that unlike single-thread research, warming up the major microprocessor components (*FF WU*) does not necessarily show higher IPC compared to the no warming up *FF No WU* method. The main reason is that the warming up phase is not deterministic since cores advance a different number of instructions and a different fragment of code is executed.

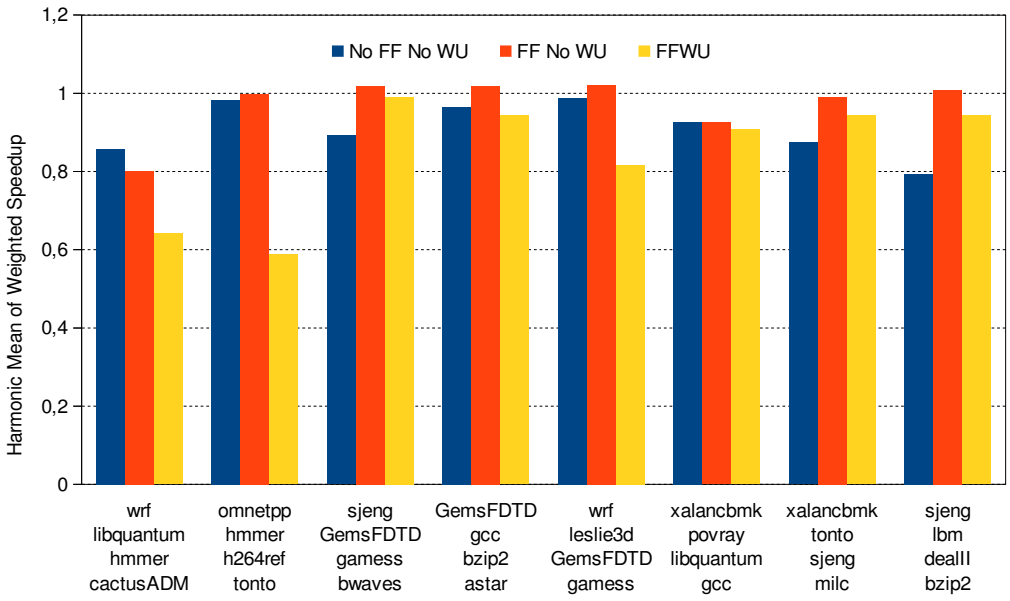


Figure 5.2: Harmonic mean of weighted speedups for the studied methods.

5.5.2 Effect of Modeling Details on Performance

The aim of this subsection is to illustrate how modeling details affect the performance results. For illustrative purposes, this section focuses on the memory controller (MC) while keeping the remaining machine as the baseline described above.

Three different memory controllers have been modeled: constant latency, a typical in-house MC, and a deeply detailed MC. The first assumes a constant memory latency and no contention. Two different variants have been tested, one with 40-cycle (row buffer hit time) and the other with 120-cycle (row buffer miss time) latencies. The in-house simulator represents a memory controller typically implemented by researchers. This model includes details such as bank and channel contention and different access times depending on whether the access hits or misses in the row buffer. Finally, the realistic MC simulator accurately models the hardware behavior with DRAM commands for DDR modules, refresh, powerdown, standby, etc. For this purpose we used DRAMsim [8].

Figure 5.4 shows the memory latency for the different MC models with the *FF WU* methodology. Two main conclusions can be drawn. First, a constant latency model is not realistic at all since memory latencies can widely vary. The reason is that memory contention and row buffer hit ratio strongly depend on the workload characteristics. Second, although an in-house simulator provides closer results to the deeply detailed model, significant differences (in excess or in defect) in time

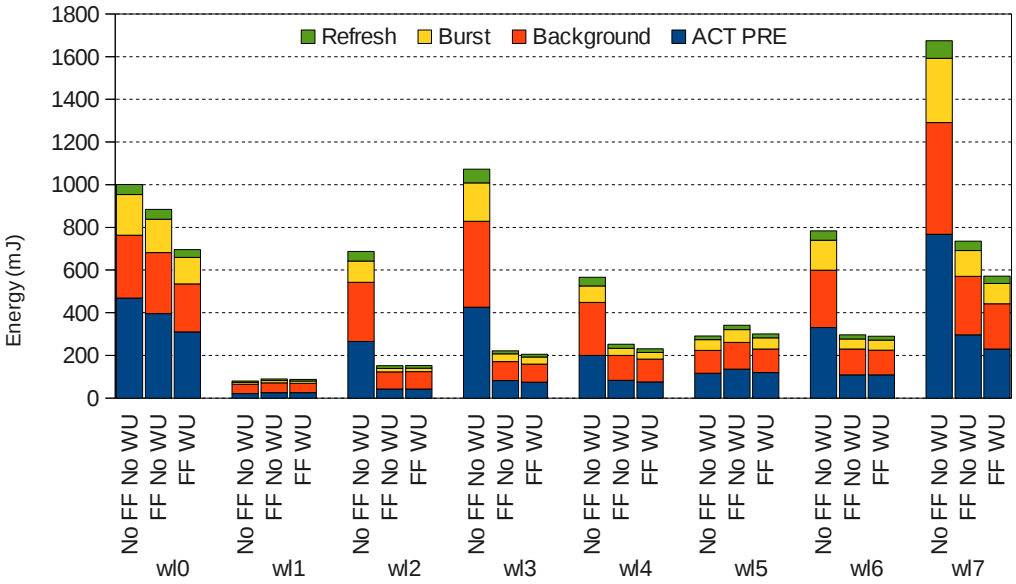


Figure 5.3: Effect of applied methodology on main memory energy consumption.

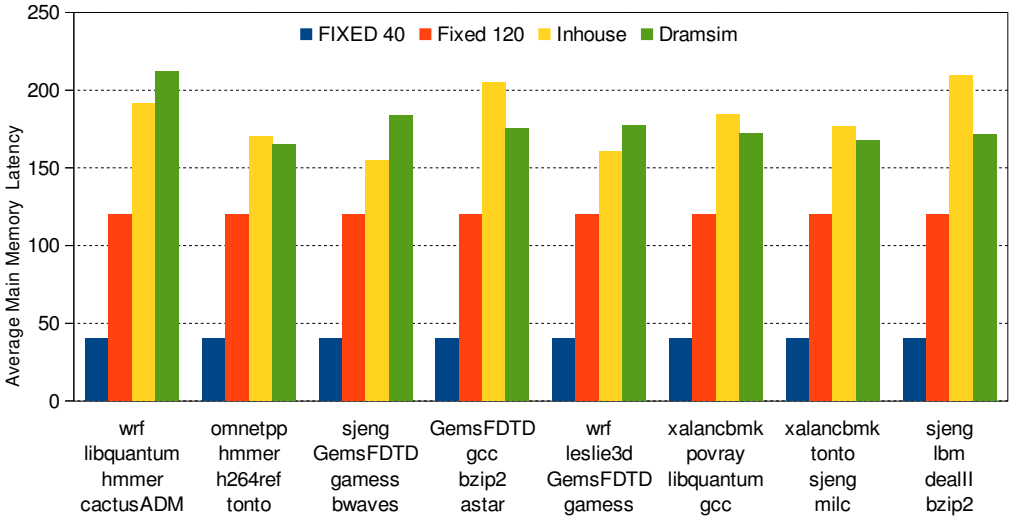


Figure 5.4: Memory latencies for the studied memory controllers.

still rise. Moreover, since the execution time differs, the use of different models will affect the performance and consumed energy as discussed above.

5.6 Summary

This chapter has studied a sample of 28 papers published in 2013 in top computer architecture conferences and has shown that there exists wide disparity among evaluation research methodologies, performance and energy metrics, and simulator environment characteristics used in multicore research. The aim of the work presented is to provide some guidelines to help researchers to carry out their research on multicores.

Regarding methodologies, we have proven that results can differ in a significant percentage depending on the metric used. Taking into account that a significant amount of proposals achieve benefits falling in between 3% and 5%, researchers should use the most representative methodology in order to obtain accurate results.

On the other hand, we have demonstrated that depending on the metrics used, the results of research study and therefore the conclusions, can widely differ.

Finally, regarding simulation frameworks, this paper has shown the importance of selecting not only a detailed performance simulator for the processor but also the selection of complementary tools for energy and/or performance of specific subsystems (e.g. main memory).

Chapter 6

Characterization Study And Prefetcher Proposal

This chapter presents a characterization study of some benchmarks from SPEC2006 benchmark suite. Building on the obtained results, an adaptive prefetching mechanism that throttles up/down its aggressiveness and also disables when no performance benefits are expected is introduced. A system with the proposed prefetcher is compared to i) the same system without prefetching enabled ii) with an aggressive prefetching scheme and iii) with other state-of-the-art adaptive approach.

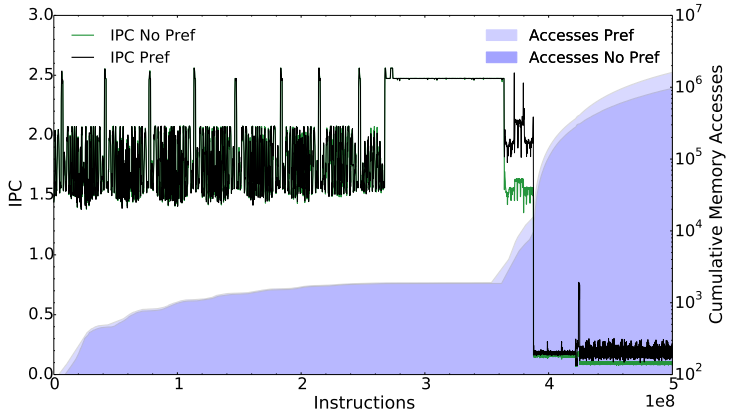
6.1 Characterization Study

This section characterizes how prefetching affects the dynamic memory behavior of the applications and analyzes how this behavior impacts on performance (i.e. IPC). To analyze this relationship, all the benchmarks have been run in stand-alone execution in a system with and without prefetching[1].

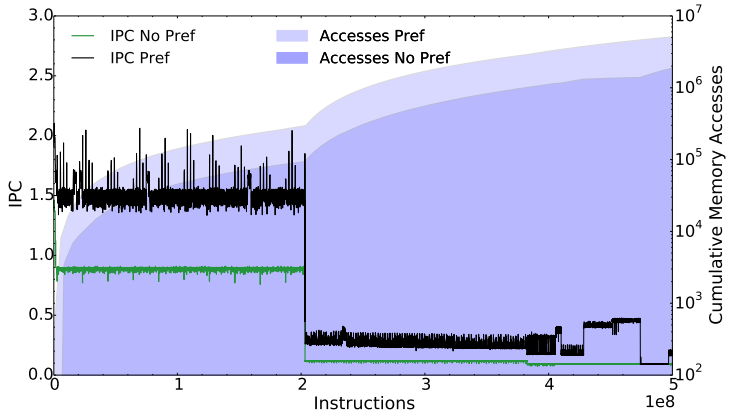
Benchmarks have been classified in four main categories by combining the behavior of two main performance metrics, i) how prefetching affects the IPC, and ii) the memory activity of the application. For illustrative purposes, Figure 6.1 and Figure 6.2 show examples of benchmarks belonging to the different categories. Each graph shows the IPC evolution (left Y axis) across the execution time in 500K-instruction intervals with prefetching and without prefetching. To analyze the relationship with the memory behavior, the cumulative amount of memory accesses (right Y axis) is also shown in the same plot using a logarithmic scale.

Below, the main characteristics of each category are discussed:

[1]The results have been obtained with the system described in Chapter 3, with the prefetcher enabled or disabled.



(a) Category 1: zeusmp

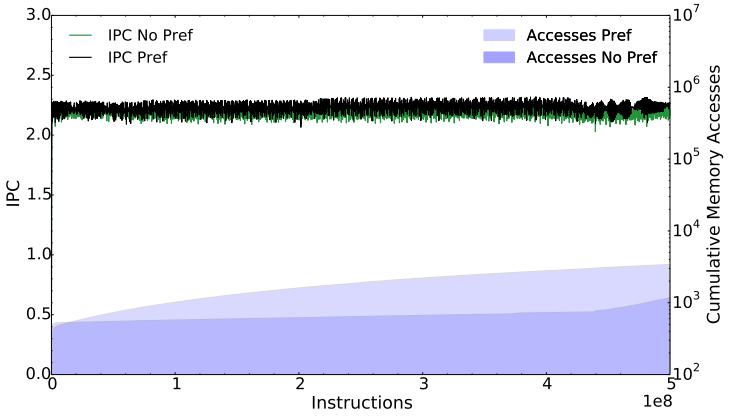


(b) Category 2: cactusADM

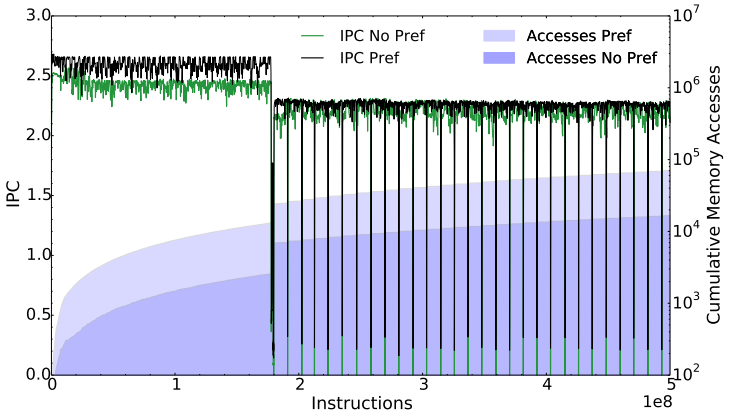
Figure 6.1: Characterization study. Categories 1 and 2.

- Category 1. This category includes memory intensive applications (e.g. number of memory accesses in the interval greater than 10^5 in a noticeable amount of execution intervals) which also present a significant amount of execution phases where prefetching does not improve the performance. Examples of these applications are `xalancbmk` and `zeusmp` (see Figure 6.1a). As it can be seen, `zeusmp` is very memory intensive at the end of the execution, and that causes a sharp IPC drop (below 0.5).

- Category 2. This group includes memory intensive applications where prefetching brings performance increases during almost all the execution time. Examples of benchmarks in this category are `hmmer` and `cactusADM` (see Figure 6.1b). As can be seen, the IPC of `cactusADM` drops below 0.5 beyond the 2×10^8 interval, due to the increase in the memory activity.



(a) Category 3: povray



(b) Category 4: namd

Figure 6.2: Characterization study. Categories 3 and 4.

- Category 3. Applications falling in this category neither are benefited by prefetching nor are memory intensive (there are not important IPC drops due to the memory activity). Some benchmarks in this category are `gamess` and `povray`. The latter shown in Figure 6.2a.

- Category 4. This category includes non memory intensive applications in which prefetching is capable of boosting the performance. Some examples are `tonto` and `namd` (see Figure 6.2b).

Applications from the first group could turn off the prefetcher with minimal or scarce performance loss, since there are execution phases where prefetching only slightly affects the performance but consumes memory bandwidth. Notice that at the end of the execution a noticeable amount (log scale) of prefetch requests bring

minor performance benefits. The second category includes those applications that are extremely benefited by prefetching, thus prefetching should not be deactivated along their execution. The third group is composed of applications in which prefetching can be disabled in almost all the execution time, but as they do not trigger a significant amount of prefetches, the expected outcome on performance of disabling the prefetcher will be low. Finally, the fourth group includes those memory intensive applications where prefetching boosts the performance. These applications do not have a high number of useless prefetches since the total amount of prefetches is relatively low, thus deactivating the prefetcher is not a good choice, since performance would drop without a noticeable traffic reduction.

6.2 Proposed Prefetching Scheme

The characterization study has shown that prefetching can be deactivated across different execution phases of the applications; at the same time, aggressive prefetching must be allowed in some intervals of the execution to boost the performance. Thus, existing adaptive prefetching approaches should be carefully enhanced to allow the prefetcher deactivation, taking into account the global main memory needs. This may bring important traffic savings (and consequently energy savings) while not affecting the performance or even improving it.

In other words, when prefetching is not properly working, the best solution to address performance and energy is to deactivate it. The key challenge that arises in the design of the deactivation policy is to decide when to activate the prefetcher again if the prefetcher is turned off, because when prefetching is disabled, there is no information related to the prefetcher accuracy. To deal with this shortcoming, this approach relies on core related metrics (e.g. ROB stalls).

The proposed activation and deactivation policies are applied to each individual core, considering both local and global information. The latter, is mainly supplied by the memory controller, which has a complete view of the system, since it is aware of the overall memory requirements of each core. With this information, the devised deactivation policy estimates which cores need more bandwidth and which ones are consuming bandwidth in excess, leading to a more effective and fairer prefetching scheme.

Basically, the prefetcher is deactivated when it is estimated that prefetches from that core are not improving the performance (i.e. low or very low accuracy), and activated when the mechanism estimates that prefetching is going to enhance the performance. The global information is used, when bandwidth is a scarce resource, to force the local prefetcher to throttle down or to turn it off with the aim of increasing the available bandwidth for other cores. The inputs needed by the devised policies are gathered during fixed-length intervals (e.g. 100K processor

Algorithm 1: Deactivation/throttling algorithm.

```
if corunners need more bandwidth
then
    reduce prefetch aggressiveness;                    // trans.  1
    if (low or very low accuracy) and low coverage
    then
        disable prefetch;                              // trans.  2
    end
end
else
end
 if low coverage
then
    increase prefetch aggressiveness;                  // trans.  3
end
```

cycles); at the end of the interval a decision for each core is taken and applied for the next interval. Next we detail the two proposed policies, one applied when prefetcher is active for deactivating it or adjusting its aggressiveness and the other applied when prefetcher is off to decide when activating it.

Deactivation/throttling policy. Algorithm 1 is applied when prefetching is active. It deactivates or throttles up/down the prefetcher. When prefetching is enabled, this policy is applied at the end of each interval to decide the proper aggressiveness level for the next interval. Next, we describe the metrics used by the algorithm to choose the optimal prefetcher aggressiveness:

- Amount of L2 cache misses due to demand requests[2] saved by prefetches during the current interval for this core, usually referred to as prefetch *coverage*.
- Prefetcher *accuracy*, that is, the percentage of prefetched blocks that are used by the core.
- Traffic in the main memory controller for each core and each bank, estimated by *BWNO*[3] [6].

[2]Since our system does not include L1 prefetchers, all L2 misses are caused by on demand requests.

[3]Bandwidth Needed by Others, where others refers the application corunners. Estimated as the average number of banks other cores are waiting in.

Algorithm 2: Activation algorithm.

```
if sudden rise in # of misses or                    // act. cond. 1
 high ROB stall (in %) or                           // act. cond. 2
 sudden IPC drop                                    // act. cond. 3
then
    if corunners do not need more bandwidth
    then
        activate prefetcher;                        // trans. 4
    end
end
```

As observed, when the running applications need more bandwidth, the core aggressiveness is reduced (labeled as transition 1 in Figure 6.3) and disabled in case of low coverage and accuracy (transition 2). The main aim is to perform a fair distribution of the memory bandwidth, and reduce the number of prefetch requests in those cores whose performance is not being benefited by prefetches. If corunners do not need more bandwidth (i.e. if there is spare bandwidth available) and the prefetcher is showing low coverage, then the prefetcher throttles up (transition 3) to improve the coverage.

Activation policy. This policy is applied at the end of each interval if the prefetcher is disabled in order to decide whether or not it should be reactivated. Algorithm 2 presents the devised activation rules. This policy must be smarter than the previous one since, as the prefetcher is not enabled, no information can be gathered to check its behavior.

Without information from the prefetcher, the proposal makes use of core performance metrics to guess if prefetching would be effective in the next execution interval. The prefetcher is activated in case of i) there is a scarce bandwidth availability and ii) prefetching benefits are predicted. The prediction relies on three independent conditions, and at least one of them must be fulfilled in order to reac-

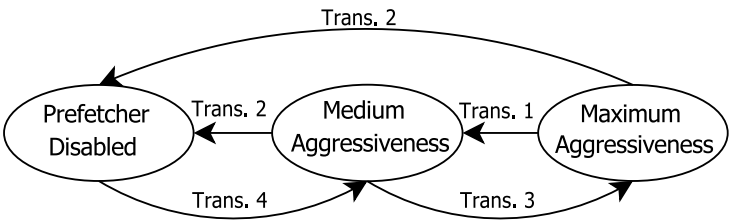


Figure 6.3: ADP state transitions. Transitions on the edges correspond to the transitions in Algorithm 1 and Algorithm 2.

tivate the prefetcher (labeled as transition 4 in Figure 6.3). Below, these conditions are discussed.

- **First condition**. The performance of a given processor is strongly related to the number of LLC[4] misses due to the huge latencies of accessing the main memory. Taking this fact into account, this condition checks if the number of L2 cache misses during the current interval has exceeded a given threshold with respect to the last interval the prefetcher was active.

- **Second condition**. The percentage of cycles that the core is stalled is another major performance indicator. Core stall cycles refer to cycles where the ROB is blocked, so the core cannot follow decoding instructions. To implement the devised activation policy, we focus on the percentage of stall cycles caused by memory instructions. If this percentage surpasses certain threshold, it means that core performance is suffering and the main reason is likely to be the long time that memory instructions take to complete. Therefore, reactivating the prefetch mechanism could help, as it may reduce that time.

- **Third condition**. The aim of this condition is to correct possible inaccurate decisions taken by the deactivation policy. To this end, it checks if performance (i.e. IPC) has suddenly dropped below a given threshold after deactivating the prefetcher. On such a case, it is reactivated again.

Notice that the proposal requires to add very little extra hardware in order to measure the performance of the core and much of it is based on existing hardware [66] already located at the multicore tiles.

Figure 6.3 represents the finite state machine corresponding to all the states of the ADP prefetcher. Each node represents a prefetcher aggressiveness level and arcs represent transitions among these levels levels. These transitions refer to three discussed transitions discussed in the deactivation and activation policies, shown in algorithms 1 and 2. Medium and maximum aggressiveness level have been assumed as two and four blocks for the experiments.

6.3 Evaluation Methodology

Experiments have been performed with multiprogrammed workloads composed of applications from the SPEC2006 benchmark suite. Each application was ran until it executed 300M instructions after a fast forward phase of 500M instructions.

[4]The L2 in our system.

Based on the characterization study presented in Section 6.1, a set of mixes has been designed to study the effects of prefetching on performance and energy in two main scenarios: under normal conditions and in extreme conditions stressing the main memory. To evaluate the first scenario mixes were designed with benchmarks randomly chosen from the characterized categories. To evaluate the second scenario, the designed mixes only include memory intensive applications from category 1 and category 2. We refer to the first type of mixes as *combined* and to the second type as *memory intensive*. Table 6.1 shows the mixes composition. Combined mixes include mixes from m0 to m3 and memory-intensive mixes from m4 to m7.

Mix type	Mix	Benchmarks (categories)			
Combined	m0	tonto (4)	h264ref (3)	hmmer (2)	omnetpp (1)
	m1	bwaves (2)	gamess (3)	Gems (3)	sjeng (3)
	m2	astar (1)	bzip2 (1)	gcc (2)	Gems (3)
	m3	gamess (3)	Gems (3)	leslie3d (2)	wrf (2)
Memory Intensive	m4	xalancbmk (1)	gcc (2)	gobmk (2)	dealII (1)
	m5	leslie3d (2)	dealII (1)	soplex (2)	gromacs (2)
	m6	mcf (2)	soplex (2)	perlbench (2)	xalancbmk (1)
	m7	dealII (1)	soplex (2)	xalancbmk (1)	gobmk (2)

Table 6.1: Mix composition.

Thresholds				
Very low Accuracy $< 20\%$	Low Accuracy $< 40\%$	Medium Accuracy $< 80\%$	High Accuracy $\geq 80\%$	IPC drop $> 10\%$
Low Coverage $< 30\%$	Rise in misses $> 15\%$	High % ROB stall $> 60\%$	High BWNO > 2.75 banks	

Table 6.2: Thresholds used in ADP.

6.4 Experimental Evaluation

To evaluate the proposed prefetcher, ADP is compared against no prefetching and two other prefetching schemes: HPAC, where throttling up and down policies are

used to control the prefetcher aggressiveness, and an aggressive prefetcher that is never deactivated nor throttled. The adaptive prefetching schemes (HPAC and ADP) use 2-block and 4-block as medium and high aggressiveness levels, respectively. The lowest aggressiveness level is assumed to be 1-block for HPAC, while ADP completely deactivates the prefetcher. The threshold values used in the experiments for Algorithm 1 and Algorithm 2 are shown in Table 6.2. Parameters were empirically determined using a limited number of simulation runs and optimized to reduce the number of memory accesses.

6.4.1 Performance Analysis

This section analyzes the benefits of the studied prefetching schemes on performance. Before studying the behavior of the prefetching approaches on the multicore processor, we explore how these approaches behave on applications in standalone execution. That is, with no interference among the memory requests of the multiple co-running applications. Figure 6.4 shows the results.

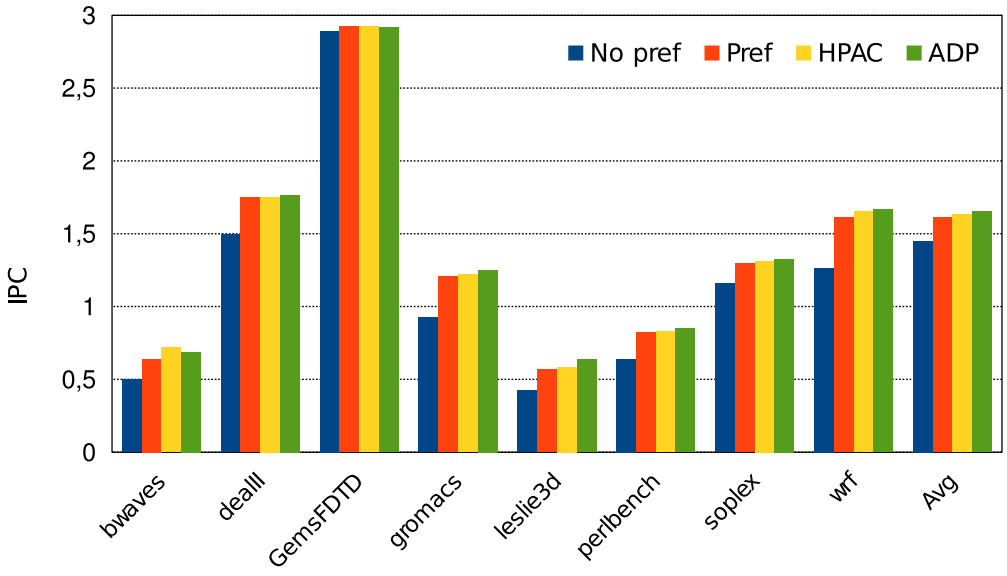


Figure 6.4: Performance of prefetchers running benchmarks in isolation.

As observed, aggressive prefetching can bring important performance benefits in most of the applications, which are on average by 16% and can be as high as 34% in some applications in the machine configuration studied in this Master's Thesis. Notice that in memory intensive applications (e.g. bwaves and leslie3d), the benefits of the aggressive prefetcher on performance are surpassed by the adaptive approaches. This means that aggressive prefetching suffers with

low memory bandwidth, whose availability is scarce in current multicores, hence adaptive prefetchers are required. Therefore, from now on, the analysis will focus on the multicore processor described above.

Figure 6.5 shows the Cumulative IPC achieved by the studied approaches across the designed mixes and Figure 6.6 shows the Harmonic Mean of IPC for each mix. In both figures `No pref` and `Pref` refer to the non-prefetching and always prefetching (i.e. aggressive prefetcher) with 4-block hardwired aggressiveness, respectively, while HPAC and ADP refer to the adaptive approaches. The plots also present the average values for combined mixes (HM0–3), memory intensive mixes (HM3–7) and for all the mixes (HM).

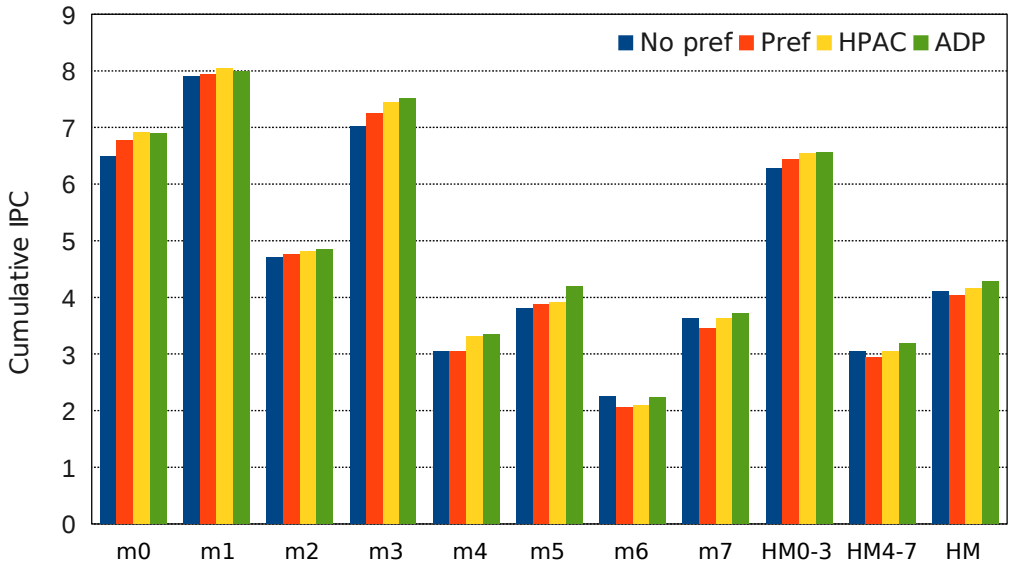


Figure 6.5: Cumulative IPC per Workload.

If performance is analyzed with Cumulative IPC then, in general, having the prefetcher always active brings performance improvements for combined mixes, but has no effect (m4 and m5) or even decreases performance (m6 and m7) for memory intensive mixes. This behavior is expected, as the huge memory contention in m4–m6 makes harder for the prefetcher to bring blocks in time, and the extra memory accesses delay demand requests.

Consequently, the adaptive approaches, which stress less the memory hierarchy, perform significantly better both for combined and memory intensive mixes.

Compared to HPAC, the proposed approach achieves better performance regardless the type of mix. ADP increases Cumulative IPC by 4.46% on average with respect to the non-prefetching scheme while HPAC by 1.25%. In combined mixes, ADP obtains the highest performance across all the mixes but for m1,

where performance is slightly lower. In memory intensive mixes, APD is always better than HPCA. Moreover, it is the only approach whose performance is equal or greater with respect to no prefetching in mixes m6 and m7.

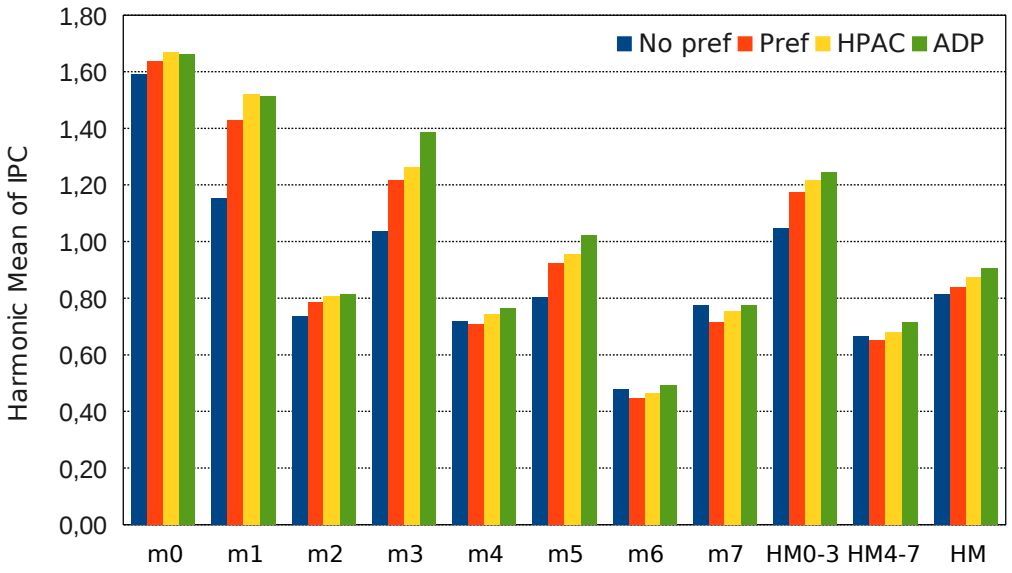


Figure 6.6: Harmonic Mean of IPC per Workload.

If performance is analyzed by Harmonic Mean of IPC (see Figure 6.6) the results are very similar. One significant difference between both figures is that the aggressive prefetcher obtains slightly better *relative* overall results if performance is measured using Harmonic Mean of IPC. Notice that this is caused by the low results obtained by the non prefetching approach for combined mixes. The difference in the relative results of figures 6.5 and 6.6, are because Cumulative IPC hides a lack of fairness that the Harmonic Mean of IPC, that tends to the least value averaged, exposes (see Section 5.3.1).

However, the conclusion is the same, ADP consistently obtains the highest performance, overcoming the problems of aggressively prefetching.

6.4.2 Prefetches Reduction Analysis

This section quantifies the reduction on the number of prefetches accomplished by the adaptive approaches with respect to the `Pref` approach.

Figure 6.7 shows the memory requests increase of the prefetching schemes compared to the non-prefetching approach. Keeping the aggressiveness high improves timeliness but consumes more bandwidth, and that can strangle the performance in memory intensive mixes. As observed, the adaptive approaches consis-

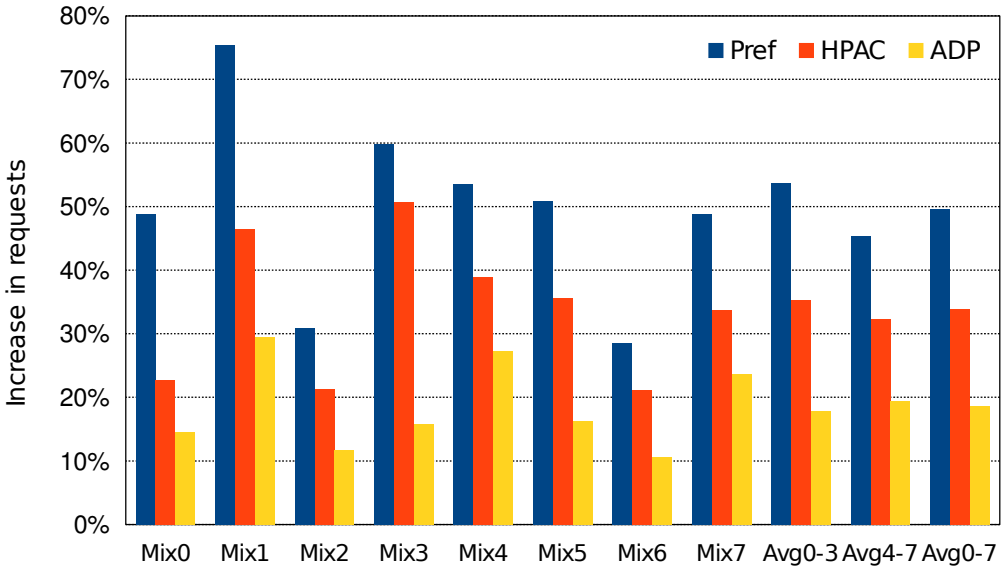
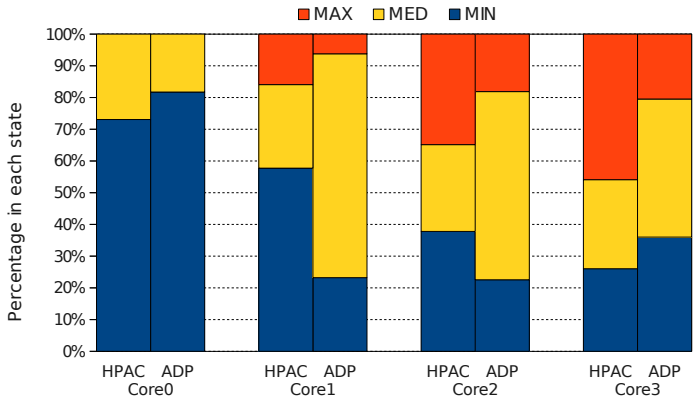


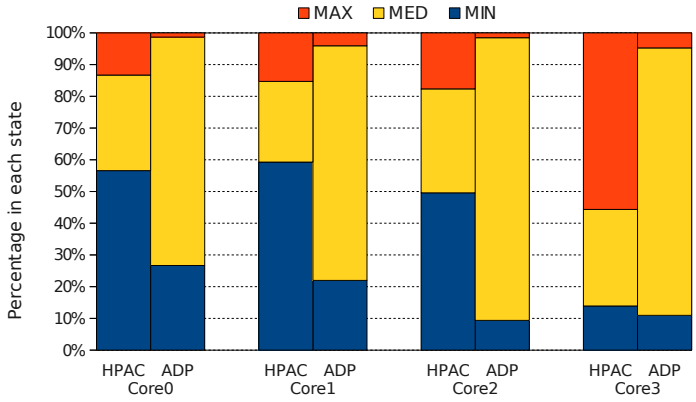
Figure 6.7: Requests increase of the studied prefetchers over no prefetching.

tently reduce the number of accesses across all the mixes with respect to always prefetching. Aggressive prefetching increases the amount of memory requests on average over no prefetching by 50%. In contrast, ADP and HPAC reduce that increase by one third and two thirds, respectively, with respect to aggressive prefetching.

To provide insights in this significant reduction, the percentage of time that each adaptive prefetching scheme spends in each state (i.e. aggressiveness level) has been analyzed. For illustrative purposes, Figure 6.8a presents the results for a combined mix (m2) and Figure 6.8b for a memory intensive mix (m7). Each bar presents the percentage of time the prefetcher of a core spends in each aggressiveness level (minimum, medium and maximum). Remember that the minimum aggressiveness is 1-block for HPAC and 0-block (i.e. deactivated) for ADP. In mix 2, a memory intensive application (category 2) runs in core 0 while applications running in the remaining cores belong to category 3 (scarcely benefited by prefetching). As observed, ADP disables the prefetcher by 82% of time in the memory intensive application (the one running in core 0). Notice that this time is even higher than the time HPAC keeps the prefetcher with 1-block aggressiveness (by 73%). Both approaches work well for this application since the highest aggressiveness is not triggered. Overall, memory bandwidth is a performance limiter in this mix, as it can be deduced by the relatively low achieved IPC. Therefore, it could be said that the lower percentage of time of ADP in maximum aggressiveness level in the remaining applications (core 1 to core 3) helps improve the



(a) Mix m2



(b) Mix m7

Figure 6.8: Fraction of time spent in each state for HPAC and ADP.

performance. Regarding the memory intensive mix m7, it can be observed that cores with ADP prefetcher scarcely trigger the highest 4-block aggressiveness. However, HPAC enables the highest aggressiveness always above 12% of the execution time in core 0 to core 2, and above 50% in core 3, in spite of the scarce bandwidth. Consequently, performance of HPAC is on pair with the non prefetching approach, and what is worse, the extra memory accesses will have a negative impact on the consumed DRAM energy as discussed below.

6.4.3 Energy Analysis

This section analyzes the main memory energy consumption of the studied schemes, which as mentioned above, represents almost half of the energy consumed

by the entire system. Figure 6.9 presents the energy results[5] broken down in four components depending on the memory activity that consumes the energy: i) activation and precharge, ii) background energy, iii) data bursts, and iv) refresh. The first component accounts for the energy consumed activating rows for reads and writes, plus the energy consumed due to precharging the bitlines. The second component refers to the energy consumed in background to keep memory devices powered on. Burst energy is consumed when data are being transferred by the memory bus in write and read operations. Finally, refresh energy is required to avoid capacitors loose the stored value as time passes.

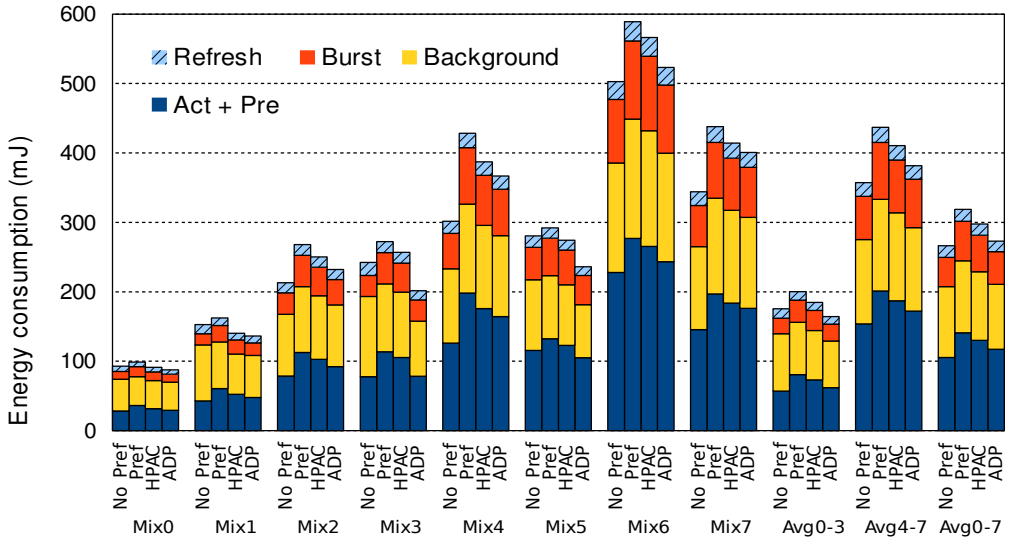


Figure 6.9: Energy consumption of the prefetching mechanisms.

On average, the aggressive prefetcher increases energy consumption by 20% over the non prefetching scheme. This expense in energy may be unacceptable, especially taking into account that aggressive prefetching can damage the performance in memory intensive mixes. An interesting observation is that ADP achieves the aforementioned performance gains with a minimal impact (by 3%) on energy, which is much lower (by 12%) than the adaptive HPAC prefetcher.

[5]Unlike IPC and memory requests which are gathered when the benchmark commits 300-million instructions (see Section 6.3), energy consumption is gathered at the end of the mix execution for simplification purposes. Therefore, this energy also considers those memory requests performed from individual statistics (IPC and memory requests) were gathered until the workload execution finishes.

6.5 Summary

This work has characterized the dynamic memory and prefetching behavior of the SPEC 2006 benchmarks across their execution time and analyzed their effect on performance. Results show that prefetching brings important performance improvements in some phases of their execution while in others it scarcely or negatively affects the performance.

This chapter has presented the adaptive ADP prefetching that implements deactivation and activation policies. The proposed prefetching mechanism has two main aims: to work in extreme conditions with scarce memory bandwidth, and to deal with main memory energy consumption. The main goal is to keep the prefetcher active when it benefits the global performance, and either throttling it down or deactivating it on the opposite case, so avoiding harmful prefetcher activity.

The key challenge of the proposal is the lack of information to turn on the prefetcher when it is deactivated. To this end, unlike previous approaches, external information to the prefetcher is also used, e.g. the percentage of ROB stall cycles. In addition, global information from the memory controller to check the interference among the running applications is also considered.

We have compared ADP both against an aggressive prefetcher and HPAC, a state-of-the-art throttling approach. Results show that ADP efficiently reduces memory traffic, which allows it to efficiently work in scarce memory bandwidth scenarios. ADP reduces the amount of memory requests by 68% with respect to an aggressive prefetcher while the reduction achieved by HPAC is only around 32%. Regarding performance, ADP speeds up memory intensive mixes by 5.9% while HPAC only improves the performance by 1.6% with respect to the non-prefetching scheme. Moreover, the proposal achieves important main memory energy savings by reducing the amount of main memory requests, and also due to performance improvements and therefore reducing the workload execution time. On average, main memory energy consumption increases only by 3% with respect to no prefetching while this percentage rises up to 12% and 20% in HPAC and aggressive approaches, respectively.

Chapter 7

Conclusions

The main focus of this Master's Thesis has been on hardware prefetchers for multicore processors. For this purpose, we have first characterized the dynamic memory and prefetching behavior of the SPEC 2006 benchmark suite across their execution time and analyzed their effect on performance. The study showed that not all the applications benefit from aggressive prefetching but it can hurt the system performance as well as rise energy consumption. These two issues become critical in multicores when prefetch requests from multiple applications compete among them for shared resources, especially main memory bandwidth.

Based on this study this Master's Thesis has proposed ADP adaptive prefetcher, which implements deactivation and activation policies for specific cores of the system. The proposed prefetching mechanisms allow the system i) to work in extreme conditions with scarce main memory bandwidth, and ii) to deal with main memory energy consumption while providing performance results on par with state-of-art adaptive prefetchers.

On the other hand, performance evaluation methodologies in multicore processors are still evolving are there is not a common trend among published papers in top conferences. To determine an adequate method for evaluation purposes, this work has presented an study of a sample of 28 papers published in 2013 in top computer architecture conferences and has shown that there exists wide disparity among evaluation research methodologies, performance and energy metrics, and simulator environment characteristics used in multicore research, providing some guidelines to help researchers to approach multicore research.

The proposed prefetching engines have been implemented and evaluated on top of extensively used simulation tools. In addition, important system components (e.g. coherence protocol, memory controller, etc.) have been properly adapted or added (e.g. prefetch buffers) to support the designed prefetchers. The devised memory controllers and prefetchers have been also used to carry out the study of performance methodologies.

7.1 Contributions

This Master Thesis has three main contributions regarding prefetching and multicore research.

The major contribution has been the ADP adaptive prefetcher for multicores. The proposal has been compared both against an aggressive prefetcher and HPAC, a state-of-the-art throttling approach. Results show that ADP efficiently reduces memory traffic, which allows it to efficiently work in scarce memory bandwidth scenarios. ADP reduces the amount of memory requests by 68% with respect to an aggressive prefetcher while the reduction achieved by HPAC is only around 32%. Regarding performance, ADP speeds up memory intensive mixes by 5.9% while HPAC only improves the performance by 1.6% with respect to the non-prefetching scheme. Moreover, the proposal achieves important main memory energy savings by reducing the amount of main memory requests, and also due to performance improvements and therefore reducing the workload execution time. On average, main memory energy consumption increases only by 3% with respect to no prefetching while this percentage rises up to 12% and 20% in HPAC and aggressive approaches, respectively.

The second contribution of this work has been the characterization of the SEC2006 benchmark suite with respect to prefetch sensitivity and memory usage. Results have shown that prefetching brings important performance improvements in some phases of their execution while in others it scarcely or negatively affects the performance. This allows benchmarks to be classified in four main categories attending to how prefetching affects the IPC and how memory intensive is the application.

Finally, the third contribution has focused on multicore evaluation methodologies. We have proved that results can differ in a significant percentage depending on the metric used. Taking into account that most proposals achieve benefits falling in between 3 and 5%, researchers should use the most representative methodology in order to obtain accurate results. On the other hand, we have demonstrated that when deciding what is the best performing machine, depending on the metrics used the results can vary. So an adequate metric is necessary. Regarding the simulation environment for obtaining experimental results, this work has demonstrated the importance of selecting not only a detailed performance simulator for the processor but also the selection of complementary tools for energy and/or performance of specific subsystems (e.g. main memory).

The research work presented in this Thesis has been published and submitted to the following conferences:

- V. Selfa, P. Navarro, C. Gómez, M. Gómez, J. Sahuquillo, “Diseño de mecanismos de prebúsqueda adaptativa bajo gestión eficiente de memoria

para procesadores multinúcleo", In *XXIV Jornadas de Paralelismo* (JP2013), pages 43–48, Madrid, Spain, 2013.

- V. Selfa, P. Navarro, C. Gómez, M. Gómez, J. Sahuquillo, "Methodologies and performance metrics to evaluate multiprogram workloads", *23rd Annual Euromicro International Conference on Parallel, Distributed and Network-based Processing* (PDP), Turku, Finland, March 4–7, 2015, Submitted.

- P. Navarro, V. Selfa, C. Gómez, M. Gómez, J. Sahuquillo, "Row Tables: Design Choices to Exploit Bank Locality in Multiprogram Workloads", *23rd Annual Euromicro International Conference on Parallel, Distributed and Network-based Processing* (PDP), Turku, Finland, March 4–7, 2015, Submitted.

- V. Selfa, P. Navarro, C. Gómez, M. Gómez, J. Sahuquillo, "Improving the use of memory bandwidth with adaptive prefetching", *21st IEEE International Symposium on High Performance Computer Architecture* (HPCA), Bay Area, CA, USA, February 2015. To be submitted.

7.2 Future Work

As for future work we plan to extend the prefetching engine presented in this work in several directions to reach the PhD Thesis. Below we summarize the main directions.

- Multithreaded workloads. Prefetching in multithreaded workloads present different behavior to multiprogrammed workloads mainly due to synchronization points like barriers and locks. We plan to characterize the prefetch and memory behavior of multithreaded applications such as SPLASH and PARSEC benchmarks suites and to apply the results for adapt our proposed prefetcher to this kind of applications.

- Prefetching in L2 caches instead of independent buffers. Bringing prefetch blocks into the cache implies some grade of cache pollution and therefore more chances that our adaptive algorithm could improve performance. Along the same line, using a shared L2 cache in place of a private one could also add pollution between cores, so it will be interesting to adapt our design to such architectures.

- Prefetching in SMT (simultaneous multithread) cores. Multithreaded cores share the L1 cache among the running cores. This cache becomes a scarce

resource that must be properly managed among the running threads; however, prefetching cannot be deactivated for all the threads due to its important benefits in performance for some of them. As for future work we plan to devise smart prefetchers for this kind of systems.

- Adaptive policies in other prefetcher engines. Adapt the proposed activation/deactivation policies to use other pattern-detection and prefetch algorithms, like PC/CS or PC/DC.

The prefetcher engine is driven by multiple threshold that must be properly tuned to reach the best performance even to minor changes in the system. In other words, most future work requires from significant tests to find the best threshold values. We plan to find the optimal values with a simple machine learning algorithm, that could be more flexible and be able to seamlessly adapt to unexpected workloads without needing manual tunning.

Finally, with respect to the methodological evaluation, we are currently extending the evaluation study of simulation methods by adding more case studies and the performance and energy results of the multiple Simpoints method in order to present a wider and deeper study.

Bibliography

[1] J. Doweck, "Inside intel core microarchitecture and smart memory access: An in-depth look at Intel innovations for accelerating execution of Memory-Related instructions, from Intel - white papers," Intel Corporation, Tech. Rep., 2006.

[2] J. Casazza, "First the tick, now the tock," Intel Corporation, Tech. Rep., 2009.

[3] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden, "Ibm POWER6 microarchitecture," *IBM J. Res. Dev.*, vol. 51, no. 6, pp. 639–662, Nov. 2007.

[4] J. Owen and M. Steinman, "Northbridge architecture of amd's griffin microprocessor family," *IEEE Micro*, vol. 28, no. 2, pp. 10–18, 2008.

[5] S. Srinath, O. Mutlu, H. Kim, and Y. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *International Symposium on High Performance Computer Architecture.*, 2007, pp. 63–74.

[6] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *International Symposium on Microarchitecture*, 2009, pp. 316–326.

[7] R. Ubal, J. Sahuquillo, S. Petit, and P. Lopez, "Multi2sim: A simulation framework to evaluate multicore-multithreaded processors," in *International Symposium on Computer Architecture and High Performance Computing.*, 2007, pp. 62–68.

[8] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 16–19, Jan. 2011.

[9] JEDEC website. [Online]. Available: http://www.jedec.org/

[10] D. G. Prez, G. Mouchard, and O. Temam, "Microlib: A case for the quantitative comparison of micro-architecture mechanisms." in *MICRO*. IEEE Computer Society, 2004, pp. 43–54.

[11] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, ser. HPCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 96–.

[12] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *International Symposium on Computer Architecture*, 1994, pp. 24–33.

[13] K. Nesbit, A. Dhodapkar, and J. Smith, "Ac/dc: an adaptive data cache prefetcher," in *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004*, 2004, pp. 135–145.

[14] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 364–373.

[15] SPEC website. [Online]. Available: http://www.spec.org/

[16] "4Gb DDR3 SDRAM MT41J512M8-64Meg x 8 x 8 banks," Micron Technology, Tech. Rep., 2011.

[17] A. Sharifi, E. Kultursay, M. T. Kandemir, and C. R. Das, "Addressing end-to-end memory access latency in NoC-based multicores," in *IEEE MICRO*, 2012, pp. 294–304.

[18] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. R. Das, "Exploring fault-tolerant network-on-chip architectures," in *International Conference on Dependable Systems and Networks*, 2006, pp. 93–104.

[19] Y. Hoskote, S. R. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-ghz mesh interconnect for a teraflops processor," *IEEE Micro*, vol. 27, no. 5, pp. 51–61, 2007.

[20] B. Jacob and D. Wang, *Principles and practices of interconnection networks*. Morgan Kaufmann, 2007.

[21] E. Ebrahimi, C. Joo, L. Onur, M. Yale, and N. Patt, "Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems," in *Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 335–346.

[22] R. Manikantan and R. Govindarajan, "Performance oriented prefetching enhancements using commit stalls," *J. Instruction-Level Parallelism*, vol. 13, 2011.

[23] S. Pugsley, Z. Chishti, C. Wilkerson, T. Chuang, R. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe, run-time evaluation of aggressive prefetchers," in *International Symposium on High Performance Computer Architecture*, 2014.

[24] N. Chidambaram Nachiappan, A. K. Mishra, M. Kademir, A. Sivasubramaniam, O. Mutlu, and C. R. Das, "Application-aware prefetch prioritization in on-chip networks," in *International Conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 441–442.

[25] J. Lee, M. Shin, H. Kim, J. Kim, and J. Huh, "Exploiting mutual awareness between prefetchers and on-chip networks in multi-cores," in *International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 177–178.

[26] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared resource management for multi-core systems," in *International Symposium on Computer Architecture*, 2011, pp. 141–152.

[27] F. Liu and Y. Solihin, "Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors," in *ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, 2011, pp. 37–48.

[28] F. Liu, X. Jiang, and Y. Solihin, "Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance," in *International Symposium on High Performance Computer Architecture*, 2010, pp. 1–12.

[29] S. Eyerman and L. Eeckhout, "Restating the case for weighted-ipc metrics to evaluate multiprogram workload performance," *IEEE Computer Architecture Letters*, vol. 99, no. 3, p. 1, Jul. 2013.

[30] P. Michaud, "Demystifying multicore throughput metrics," *IEEE Computer Architecture Letters*, vol. 12, no. 2, pp. 63–66, 2013.

[31] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, "Application-to-core mapping policies to reduce memory interference in multi-core systems," in *PACT*. ACM, 2012, pp. 455–456.

[32] K. V. Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-isa heterogeneous multi-cores," in *PACT*, 2013, pp. 177–187.

[33] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31. ACM, 2003, pp. 318–319.

[34] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman, "Ac-dimm: Associative computing with stt-mram," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 189–200, Jun. 2013.

[35] S. Navada, N. K. Choudhary, S. V. Wadhavkar, and E. Rotenberg, "A unified view of non-monotonic core selection and application steering in heterogeneous chip multiprocessors," in *PACT*, 2013, pp. 133–144.

[36] M. Shevgoor, J.-S. Kim, N. Chatterjee, R. Balasubramonian, A. Davis, and A. N. Udipi, "Quantifying the relationship between the power delivery network and architectural policies in a 3d-stacked memory device," in *MICRO*, 2013, pp. 198–209.

[37] A. Kolli, A. G. Saidi, and T. F. Wenisch, "Rdip: return-address-stack directed instruction prefetching," in *MICRO*, 2013, pp. 260–271.

[38] S. Padmanabha, A. Lukefahr, R. Das, and S. A. Mahlke, "Trace based phase prediction for tightly-coupled heterogeneous cores," in *MICRO*, 2013, pp. 445–456.

[39] V. Mekkat, A. Holey, P.-C. Yew, and A. Zhai, "Managing shared last-level cache in a heterogeneous multicore processor," in *PACT*, 2013, pp. 225–234.

[40] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *MICRO*, 2013, pp. 247–259.

[41] D. A. Jiménez, "Insertion and promotion for tree-based pseudolru last-level caches," in *MICRO*, 2013, pp. 284–296.

[42] J. Albericio, P. Ibáñez, V. Viñals, and J. M. Llabería, "The reuse cache: downsizing the shared last-level cache," in *MICRO*, 2013, pp. 310–321.

[43] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Row-clone: fast and energy-efficient in-dram bulk data copy and initialization," in *MICRO*, 2013, pp. 185–197.

[44] S. Eyerman and L. Eeckhout, "System-level performance metrics for multi-program workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, May 2008.

[45] J. R. Mashey, "War of the benchmark means: Time for a truce," *SIGARCH Comput. Archit. News*, vol. 32, no. 4, pp. 1–14, Sep. 2004.

[46] M. Zhou, Y. Du, B. R. Childers, R. G. Melhem, and D. Mossé, "Writeback-aware bandwidth partitioning for multi-core systems with pcm," in *PACT*, 2013, pp. 113–122.

[47] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 316–326.

[48] M. F. K. Luo, J. Gummaraju, "Balancing throughput and fairness in smt processors," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ser. MICRO 42. IEEE Computer Society Press, 200, pp. 164–171.

[49] S. Sardashti and D. A. Wood, "Decoupled compressed cache: exploiting spatial locality for energy-optimized compressed caching," in *MICRO*, 2013, pp. 62–73.

[50] L. G. Menezo, V. Puente, and J.-Á. Gregorio, "The case for a scalable coherence protocol for complex on-chip cache hierarchies in many-core systems," in *PACT*, 2013, pp. 279–288.

[51] M. N. Bojnordi and E. Ipek, "Programmable ddrx controllers," *IEEE Micro*, vol. 33, no. 3, pp. 106–115, 2013.

[52] T. J. Ham, B. K. Chelepalli, N. Xue, and B. C. Lee, "Disintegrated control for energy-efficient and heterogeneous memory systems," in *HPCA*, Feb 2013, pp. 424–435.

[53] S. M. Khan, A. R. Alameldeen, C. Wilkerson, J. Kulkarni, and D. A. Jimenez, "Improving multi-core performance using mixed-cell cache architecture," in *HPCA*, Feb 2013, pp. 119–130.

[54] P. Yedlapalli, J. Kotra, E. Kultursay, M. T. Kandemir, C. R. Das, and A. Sivasubramaniam, "Meeting midway: Improving cmp performance with memory-side prefetching," in *PACT*, 2013, pp. 289–298.

[55] R. Das, S. Narayanasamy, S. K. Satpathy, and R. G. Dreslinski, "Catnap: Energy proportional multiple network-on-chip," in *ISCA*. ACM, 2013, pp. 320–331.

[56] A. Sembrant, E. Hagersten, and D. Black-Schaffer, "Tlc: a tag-less cache for reducing dynamic first level cache energy," in *MICRO*, 2013, pp. 49–61.

[57] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *ISCA*. ACM, 2013, pp. 404–415.

[58] P. Petrica, A. M. Izraelevitz, D. H. Albonesi, and C. A. Shoemaker, "Flicker: A dynamically adaptive architecture for power limited multicore systems," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 13–23, Jun. 2013.

[59] W. Heirman, T. E. Carlson, I. Hur, and L. Eeckhout. The sniper multi-core simulator. [Online]. Available: http://snipersim.org

[60] A. Patel, F. Afram, S. Chen, and K. Ghose. Marssx86 micro-architectural and system simulator for x86-based systems. [Online]. Available: http://marss86.org

[61] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[62] H. M. G. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood, "Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip," in *ISCA*. ACM, 2013, pp. 583–594.

[63] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," in *HP Laboratories*, 2009.

[64] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration," in *DATE*. European Design and Automation Association, 2009, pp. 423–428.

[65] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "The mcpat framework for multicore and manycore architectures:

Simultaneously modeling power, area, and timing," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, pp. 5:1–5:29, Apr. 2013.

[66] P. Irelan and S. Kuo, "Performance monitoring unit sharing guide," Intel Corporation, Tech. Rep., 2009.