COMPUTER RESEARCH AND MODELING
2023 VOL. 15 NO. 3 P. 639–656
DOI: 10.20537/2076-7633-2023-15-3-639-656

UDC: 004.318

Reducing miss rate in a non-inclusive cache with inclusive directory of a chip multiprocessor

Yu. A. Nedbailo[1,2,a], A. V. Surchenko[1,b], I. N. Bychkov[1,2,c]

[1]MCST JSC,
108 Profsoyuznaya st., Moscow, 117437, Russia
[2]INEUM im. I. S. Bruka,
24 Vavilova st., Moscow, 119334, Russia

E-mail: [a] yuri.nedbailo@mail.ru, [b] Alexander.V.Surchenko@mcst.ru, [c] Ignat.N.Bychkov@mcst.ru

Received 14.04.2023.
Accepted for publication 03.05.2023.

Although the era of exponential performance growth in computer chips has ended, processor core numbers have reached 16 or more even in general-purpose desktop CPUs. As DRAM throughput is unable to keep pace with this computing power growth, CPU designers need to find ways of lowering memory traffic per instruction. The straightforward way to do this is to reduce the miss rate of the last-level cache. Assuming "non-inclusive cache, inclusive directory" (NCID) scheme already implemented, three ways of reducing the cache miss rate further were studied.

The first is to achieve more uniform usage of cache banks and sets by employing hash-based interleaving and indexing. In the experiments in SPEC CPU2017 refrate tests, even the simplest XOR-based hash functions demonstrated a performance increase of 3.2 %, 9.1 %, and 8.2 % for CPU configurations with 16, 32, and 64 cores and last-level cache banks, comparable to the results of more complex matrix-, division- and CRC-based functions.

The second optimisation is aimed at reducing replication at different cache levels by means of automatically switching to the exclusive scheme when it appears optimal. A known scheme of this type, FLEXclusion, was modified for use in NCID caches and showed an average performance gain of 3.8 %, 5.4 %, and 7.9 % for 16-, 32-, and 64-core configurations.

The third optimisation is to increase the effective cache capacity using compression. The compression rate of the inexpensive and fast BDI*-HL (Base-Delta-Immediate Modified, Half-Line) algorithm, designed for NCID, was measured, and the respective increase in cache capacity yielded roughly 1 % of the average performance increase.

All three optimisations can be combined and demonstrated a performance gain of 7.7 %, 16 % and 19 % for CPU configurations with 16, 32, and 64 cores and banks, respectively.

Keywords: multicore processor, memory subsystem, distributed shared cache, NCID, XOR-based hash function, data compression

Citation: *Computer Research and Modeling*, 2023, vol. 15, no. 3, pp. 639–656.

© 2023 Yuri A. Nedbailo, Alexander V. Surchenko, Ignat N. Bychkov
This work is licensed under the Creative Commons Attribution-NoDerivs 3.0 Unported License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/3.0/
or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

КОМПЬЮТЕРНЫЕ ИССЛЕДОВАНИЯ
И МОДЕЛИРОВАНИЕ 2023 Т. 15 № 3 С. 639–656
DOI: 10.20537/2076-7633-2023-15-3-639-656

УДК: 004.318

Снижение частоты промахов в неинклюзивный кэш с инклюзивным справочником многоядерного процессора

Ю. А. Недбайло[1,2,a], А. В. Сурченко[1,b], И. Н. Бычков[1,2,c]

[1]АО «МЦСТ»,
Россия, 117437, г. Москва, ул. Профсоюзная, д. 108
[2]ПАО «ИНЭУМ им. И. С. Брука»,
Россия, 119334, г. Москва, ул. Вавилова, д. 24

E-mail: [a] yuri.nedbailo@mail.ru, [b] Alexander.V.Surchenko@mcst.ru, [c] Ignat.N.Bychkov@mcst.ru

Получено 14.04.2023.
Принято к публикации 03.05.2023.

Хотя эпоха экспоненциального роста производительности компьютерных микросхем закончилась, даже настольные процессоры общего назначения сегодня имеют 16 и больше ядер. Поскольку пропускная способность памяти DRAM растет не с такой скоростью, как вычислительная мощность ядер, разработчики процессоров должны искать пути уменьшения частоты обменов с памятью на одну инструкцию. Непосредственным путем к этому является снижение частоты промахов в кэш последнего уровня. Предполагая уже реализованной схему «неинклюзивный кэш с инклюзивным справочником» (NCID), три способа дальнейшего снижения частоты промахов были исследованы.

Первый способ — это достижение более равномерного использования банков и наборов кэша применением хэш-функций для интерливинга и индексирования. В экспериментах в тестах SPEC CPU2017 refrate, даже простейшие хэш-функции на основе XOR показали увеличение производительности на 3,2 %, 9,1 % и 8,2 % в конфигурациях процессора с 16, 32 и 64 ядрами и банками общего кэша, сравнимое с результатами для более сложных функций на основе матриц, деления и CRC.

Вторая оптимизация нацелена на уменьшение дублирования на разных уровнях кэшей путем автоматического переключения на эксклюзивную схему, когда она выглядит оптимальной. Известная схема этого типа, FLEXclusion, была модифицирована для использования в NCID-кэшах и показала улучшение производительности в среднем на 3,8 %, 5,4 % и 7,9 % для 16-, 32- и 64-ядерных конфигураций.

Третьей оптимизацией является увеличение фактической емкости кэша использованием компрессии. Частота сжатия недорогим и быстрым алгоритмом BDI*-HL (Base-Delta-Immediate Modified, Half-Line), разработанным для NCID, была измерена, и соответствующее увеличение емкости кэша дало около 1 % среднего повышения производительности.

Все три оптимизации могут сочетаться и продемонстрировали прирост производительности в 7,7 %, 16 % и 19 % для конфигураций с 16, 32 и 64 ядрами и банками соответственно.

Ключевые слова: многоядерный процессор, подсистема памяти, распределенный общий кэш, NCID, хэш-функции на основе XOR, компрессия данных

© 2023 Юрий Александрович Недбайло, Александр Викторович Сурченко, Игнат Николаевич Бычков
Статья доступна по лицензии Creative Commons Attribution-NoDerivs 3.0 Unported License.
Чтобы получить текст лицензии, посетите веб-сайт http://creativecommons.org/licenses/by-nd/3.0/
или отправьте письмо в Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Introduction

Although the era of exponential performance growth in computer chips, formulated as Moore's law and Dennard scaling, is over, the industry still demonstrates notable progress. Even common desktop CPUs today possess up to 16 traditional general-purpose cores. The addition of energy- and area-efficient cores has proven an effective way to push the core number and their resulting performance even further. Another promising avenue for increasing computing power is the adoption of domain-specific architectures [Hennessy, Patterson, 2017].

While the computing power of CPUs grows, the throughput of standard DRAM memory modules they use does not keep pace. Doubling roughly every five years, it becomes increasingly insufficient and thus prompts CPU designers to find ways of reducing memory traffic per instruction. The obvious way is to reduce cache miss rates. With multiple cores and workloads utilising them unevenly, a shared cache is preferable as it dynamically allocates its space to the active cores [Iyer et al., 2021]. At the same time, private caches are still needed to keep the average latency low, which leads to cache hierarchy and the use of some coherence protocol, usually involving some degree of cache inclusion. The cache hierarchy design of a multicore CPU is, therefore, a problem with many variables [Balasubramonian, Jouppi, Muralimanohar, 2011].

In this paper, the "non-inclusive cache, inclusive directory" (NCID) cache configuration of a 16-core general-purpose VLIW CPU is examined, and several ways of reducing shared cache miss rates are studied and evaluated using trace-based simulation: hash-based interleaving and indexing, reduction of L2-L3 replication, and data compression.

Related work

Shared cache organisation

Although a distributed shared cache is usually organised using address-based interleaving, often referred to as NUCA, alternatives also exist. A notable one is cooperative caching [Herrero, González, Canal, 2008], where private caches can act like a distributed shared if a forwarding mechanism for replaced cache lines is implemented. Another is page colouring [Cho, Jin, 2006] which assigns each memory page a cache bank (or their group) near the core running the program and stores this information in the page table; this approach can be combined with NUCA [Hardavellas et al., 2009]. Traditional NUCA has little hardware overhead, is easy to manage (e. g., implement QoS) and needs no software support.

Cache coherence

Directory-based cache coherence is not the only option for many-core CPUs. While snoop-based coherence has a reputation of poorly scalable due to quadratic growth of snoop traffic, optical interconnect can mitigate this limitation [Wang et al., 2017]. A distributed directory can be organised in different ways, e. g., tagless [Zebchuk et al., 2009] or cuckoo [Ferdman et al., 2011]. NCID in its basic variant considered in this work is a relatively simple extension of NUCA.

Cache indexing

Hash functions designed for calculating cache index are discussed in [González et al., 1997; Vandierendonck, De Bosschere, 2005; Salwan, 2013]. The non-uniformity of cache set use was also addressed by skewed associativity [Seznec, 1997]. This approach can be extended to effectively increase cache associativity by adding a complex replacement mechanism [Sanchez, Kozyrakis, 2010]. Hash-based indexing implies traditional set associativity and is, therefore, easy to implement and combine with other optimisations.

Reducing L2-L3 replication

Cache line replication can be addressed by implementing cache-hierarchy aware replacement policies such as CHAR [Chaudhuri et al., 2012]. However, their aim is mainly reduction in access latency and data movement between the levels, not last-level cache miss rate. Some advanced multi-level caching policies, such as Hawkeye [Jain, Lin, 2016] and MPPPB [Jiménez, Teran, 2017], can reduce the last-level cache miss rate at the expense of increased traffic. A bloom filter-based scheme FILM [Wang et al., 2019] is the known solution without negative performance impacts. A different approach is the use of special instructions or instruction hints [Iyer et al., 2021], which requires assistance from the programmer, compiler or the OS. FLEXclusion scheme [Sim et al., 2012] considered in this work is effective, easy and inexpensive to implement in traditional caches and requires no software support.

Cache compression

A detailed survey of existing cache compression methods can be found in [Carvalho, Seznec, 2021]. The effectiveness of such methods comes at the expense of increased cache complexity, more information to store, and data access delays. The BDI*-HL method presented in this work is based on BDI [Pekhimenko et al., 2012] and was designed specifically for NCID caches, combining ease of implementation, low hardware cost and negligible latency impact.

Simulation methods

An acknowledged tool for evaluating CPU performance depending on memory subsystem details is Gem 5 [Binkert et al., 2011]. However, intended primarily for full-system execution-based simulation, it is not optimised for modelling many-core CPUs; the project has forks aimed at this [Qureshi et al., 2019]. Trace-based simulation methods offer better scalability, effectively decoupling the memory subsystem model from the rest of the system. One such technique, Elastic Traces [Jagtap et al., 2016], demonstrated a speed-up of 6–8 times and was included in Gem 5. The simulation framework used in this work belongs to the same trace-based type and was optimised further with trace sampling, which reduces the simulation time of CPU2017 refrate tests by several orders, and a simplified on-chip network model. With these optimisations, all the experiments in this work took roughly two months of Ryzen 5950X CPU time.

Baseline non-inclusive configuration

The object of this study will be a 16-core general-purpose CPU of VLIW architecture [Shilov, 2020; Nedbailo et al., 2021] and its trace-based model, depicted in Figure 1. The simulation framework is described in [Nedbailo, 2020]; it was updated and refined, and its accuracy against a real machine of the target configuration in SPEC CPU2017 tests is shown in Figure 2. For more broadly applicable results, the optimisations were simulated on the same model with 32 or 64 cores, and a respectively increased L3 bank number.

The simulated configurations are summarised in Table 1. The focus of this paper is the distributed shared L3 cache, split into 16/32/64 banks, 2 MB each, and its communication with private L2 caches, 1 MB each. As a result of former optimisations, the caches implement the “non-inclusive cache, inclusive directory” (NCID) scheme [Zhao et al., 2010]. The L3 cache itself is decoupled from other caches; however, it has a twofold margin of tags, the excess acting as a directory for private L1 and L2 caches. As in the inclusive scheme, the L3 cache creates a copy on L3 read miss, so L2 contents are partly replicated in L3; non-inclusiveness means that no back-invalidate is issued on L3 eviction, so a copy may still be present in L2 caches without a record in L3 (Figure 3).

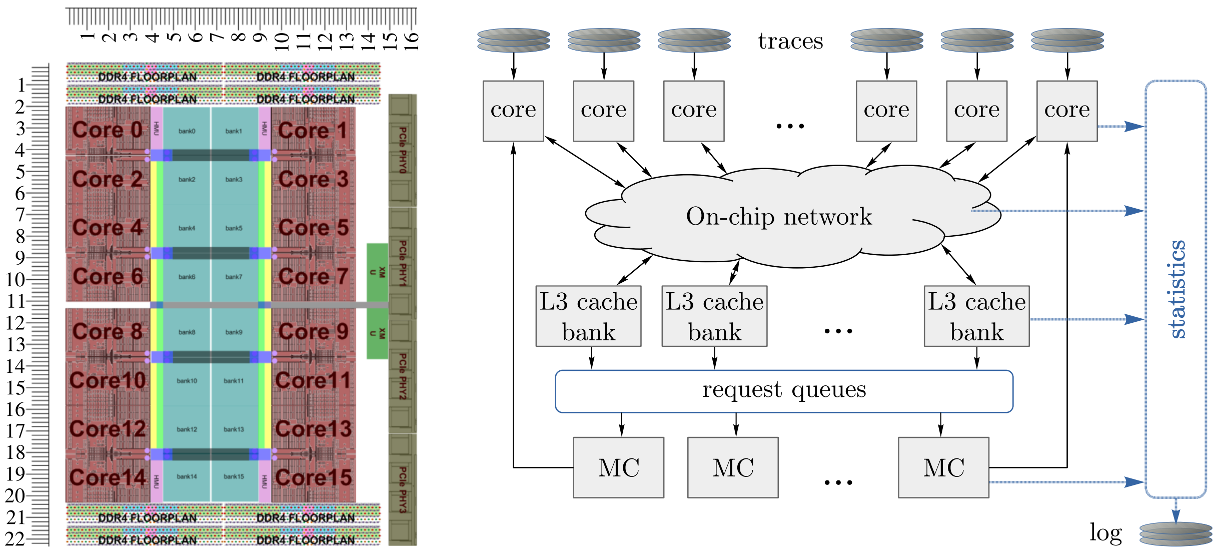

Figure 1. The floorplan (a) and the model (b) of the 16-core CPU simulated

Table 1. The CPU configurations used for simulation

Component	Configuration
Core	16/32/64 cores, VLIW @ 2000 MHz
L1i cache	Private, 128 KB / core, 4-way, 256 B line
L1d cache	Private, 64 KB / core, 4-way, 32 B line, write-through
L2 cache	Private, 1 MB / core, 4-way, 64 B line, non-inclusive
L3 cache	Shared, 2 MB / core, 16-way, 64 B line, NCID
Interconnect	4×4/8×8 mesh, 1 request + 32 B data per core cycle
RAM	8 channels, DDR4 @ 3200 MT/s

In the NCID scheme, only the directory is inclusive; thanks to its capacity margin — 2 MB of directory for 1 MB L2 per core — directory back-invalidates should hit L2 rarely enough in most cases. Another optimisation was implemented to make back-invalidation itself a rare event. On the eviction from an L2 cache, the L1 caches are checked and, in the case of a miss, a directory update request is issued. The same interconnection channels as for the normal requests from L2 to L3 are used, so any races are prevented by strict ordering. To prevent an excessive load on these channels, directory update requests are allowed to be dropped when request traffic is high. When traffic is not very high, which is true for most applications, this optimisation keeps some free space in the directory, so back-invalidates rarely occur, and the performance impact of the coherence mechanism is minimal [Martin, Hill, Sorin, 2012].

The optimality of this cache configuration was assessed on the 16-core model with 16 L3 banks. While L3 cache associativity of 32 yields a 1.2 % higher average IPC than 16, this advantage disappears when either the bank number increases to 32 or 64 (Figure 4) or hash-based indexing is used, discussed in the next section. NRU replacement policy, used by default, performed slightly better than Pseudo-LRU and 0.1 % worse than costly LRU. The average impact of directory back-invalidates, evaluated by simply disabling them in the simulator, is 0.7 % (Figure 5). As L2 caches are non-inclusive (with respect to L1), which might undermine the directory update optimisation, enforcing their inclusiveness was tested and brought no significant change. As L1i caches are included in the directory without any special protection, disabling their back-invalidation was tested and also showed a minimal effect (Figure 5).

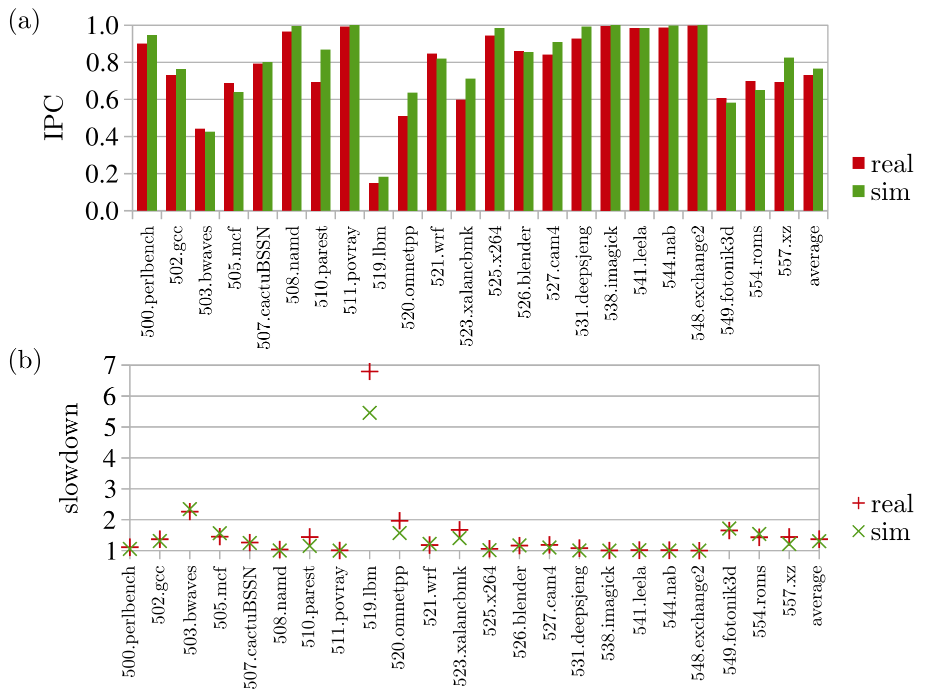

Figure 2. Instructons per cycle per core (IPC) in CPU2017 refrate tests on a real 16-core system and its trace-based model: a) single-core test; b) slowdown with 16-cores running the test

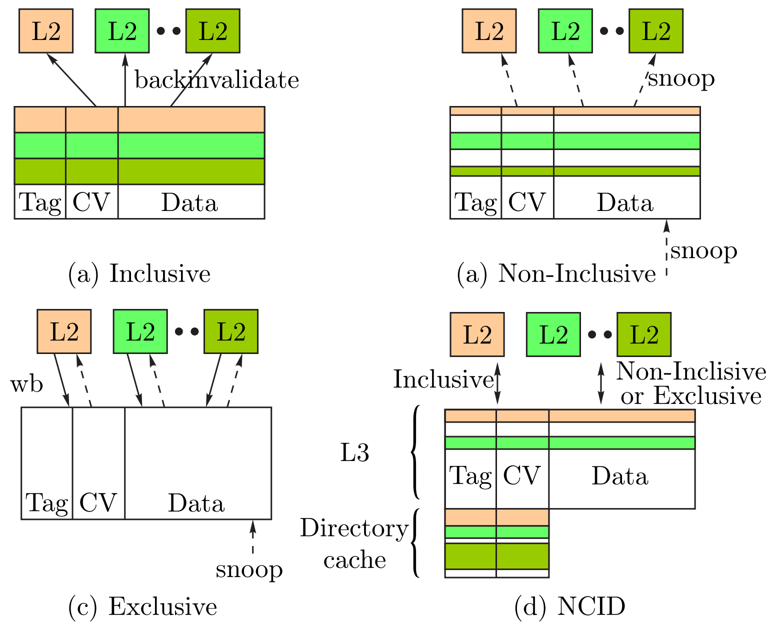

Figure 3. L2-L3 cache communication in the inclusive, non-inclusive, exclusive, and NCID schemes [Zhao et al., 2010]

Three ways of further optimisation of the shared L3 cache in this configuration were examined and evaluated.

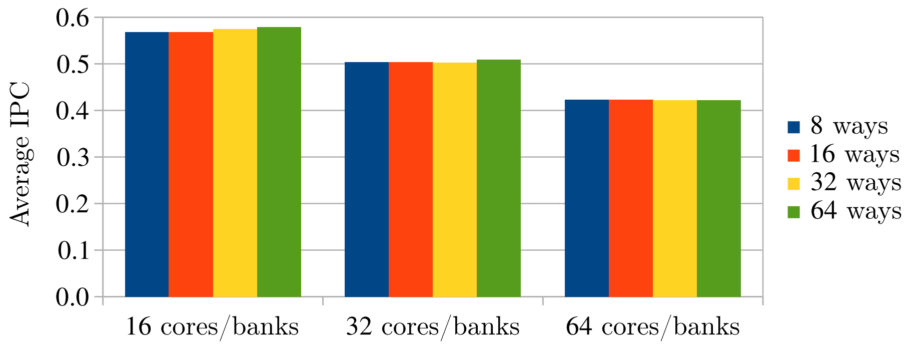

Figure 4. The average IPC per core in CPU2017 multi-threaded refrate tests for different configurations and L3 cache associativity

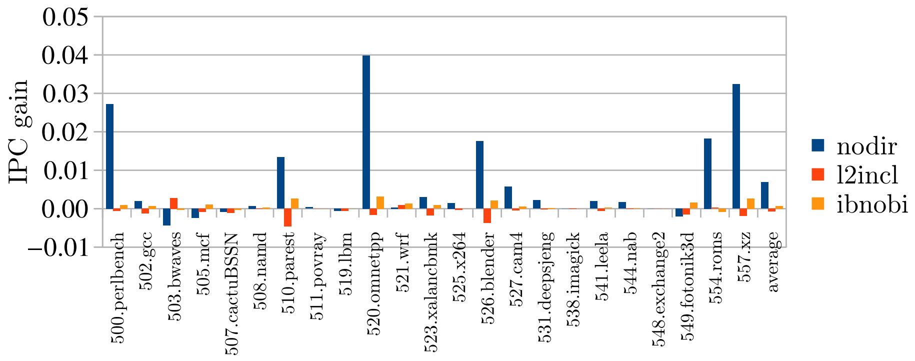

Figure 5. The relative increase in IPC in CPU2017 multi-threaded refrate tests without any back-invalidates (nodir), with L1-L2 inclusion (l2incl), and without L1i back-invalidates (ibnobi)

Hash-based interleaving and indexing

The first way of optimisation can be found by observing the uneven use of cache banks and sets when traditional sequential interleaving and indexing are employed. Bank number and index within the bank for an address are usually calculated by simply taking the corresponding least significant bits of the address; in a 16-bank cache with 2 MB banks, 16-way sets, and 64-byte lines, it means using 9th to 6th bits as the bank number and 20th to 10th as the index. Since virtual and physical addresses always share the least significant bits, executing the same program code on each core translates the unevenness of each thread's memory access pattern into uneven utilisation of cache banks and sets.

To mitigate this problem and, as a result, reduce the cache miss rate, more address bits can be used for bank number and index calculation. While numerous ways of doing so are known, e. g. by division [Kharbutli, Solihin, Lee, 2005], the most alluring are those having little logical complexity and not increasing the number of bits to store in the cache as "tags". A good example of such indexing was proposed along with skewed associativity [Seznec, 1997]:

$$f_0(A) = \phi(A_1) \oplus \phi'(A_2). \tag{1}$$

Each index bit is calculated here simply as XOR of two bits from the "index" (A_2) and "tag" (A_1) address parts, tag bits are stored in the cache just as in traditional indexing, and the lowest address bits are restored from the index and the tag using a similar formula. The same approach can be taken to calculate the bank number. To keep the same minimal size of tags to be stored while allowing more complex functions, the above formula can be generalised (leaving unnecessary permutations) to:

$$f(A) = A_2 \oplus hash(A_1). \tag{2}$$

Here, any function can be used as *hash*, the choice depending on its complexity and how uniformly its values spread with real-world address sequences. Unfortunately, the latter depends not only on the program behaviour, but also on memory allocation by the OS, which cannot be modelled accurately, so the optimal function can hardly be found analytically.

Virtual address mapping

The event traces, generated and used in the simulation framework to evaluate CPU performance, contain only virtual addresses of memory references. Two approaches to simulating memory allocation, producing physical addresses used by caches, were tested. The first, further denoted as *simple*, is mapping virtual addresses onto physical ones according to the formula:

$$PA(t) = (VA + psize \cdot t) \bmod 4GB + t \cdot 4GB,$$

where t is the thread (i. e., core) number, *psize* is the page size, which is 2 MB for the computer system considered here, VA and PA are virtual and physical addresses, respectively.

The second approach, denoted as *random*, involves a random address shift for each thread:

$$PA(t) = (VA + psize \cdot R(t)) \bmod 4GB + t \cdot 4GB,$$

where $R(t)$ is a random number with uneven distribution: bit 0 is 50/50 zero or one, bit 1 is 55/45, bit 2 is 60/40 etc., so the least significant bits are more likely to alter. To avoid the fluctuation of simulation results, an array of such numbers was generated, so $R(t)$ is a function of t, total core number and a test run parameter acting as a seed. Each test is run ten times with this seed parameter varying from 0 to 9, and the results of the runs are averaged.

The difference in the test results between the two address mappings depending on hash functions used for interleaving and indexing (explained further) is shown in Figure 6. Although the mapping method noticeably affects individual results, the overall picture remains nearly the same. The main difference is the performance of the non-optimised baseline configuration (*plain*) with 32 or 64 cores; the 16-core variant is not affected because 6-bit offset, 4-bit bank number and 11-bit index fit within 21 address bits not depending on address mapping. The *simple* mapping is, therefore, suitable for modelling hash-optimised configurations, while hash function comparison should be performed using *random* or some other realistic mapping method.

Hash functions tested

XOR-based hash functions tested in this work for indexing can be depicted as matrices shown in Figure 7 (assuming a five-bit index size instead of the actual 11-bit for simplicity). Each index bit is calculated as the parity of the bit-wise XOR of the ith column of the matrix with the cache line address (both A_1 and A_2), the upper bit of the column corresponding to the least significant address bit. If this matrix contains an identity or permutation submatrix, which is true for all four, such a hash-based indexing function is equivalent to Equation (2) (up to permutations) and thus does not require increasing cache tags. The number of 1's in a column determines hash computation delay, and the first two functions fall under the fastest variety expressed by Equation (1). Matrices c) and d) are H matrices proposed in [Cho et al., 2008] for skewed-associative caches and used in ZCache [Sanchez, Kozyrakis, 2010].

Other indexing functions tested were division-based [Kharbutli, Solihin, Lee, 2005] (effectively reducing the set number to the nearest prime number) and, as a reference, CRC-based (CRC32 of the address was computed and then processed with a *shift* function). Interleaving functions tested were the same shift and CRC, with the difference that more address parts were XOR'ed, as if the matrix a) was extended down to cover 32 address bits.

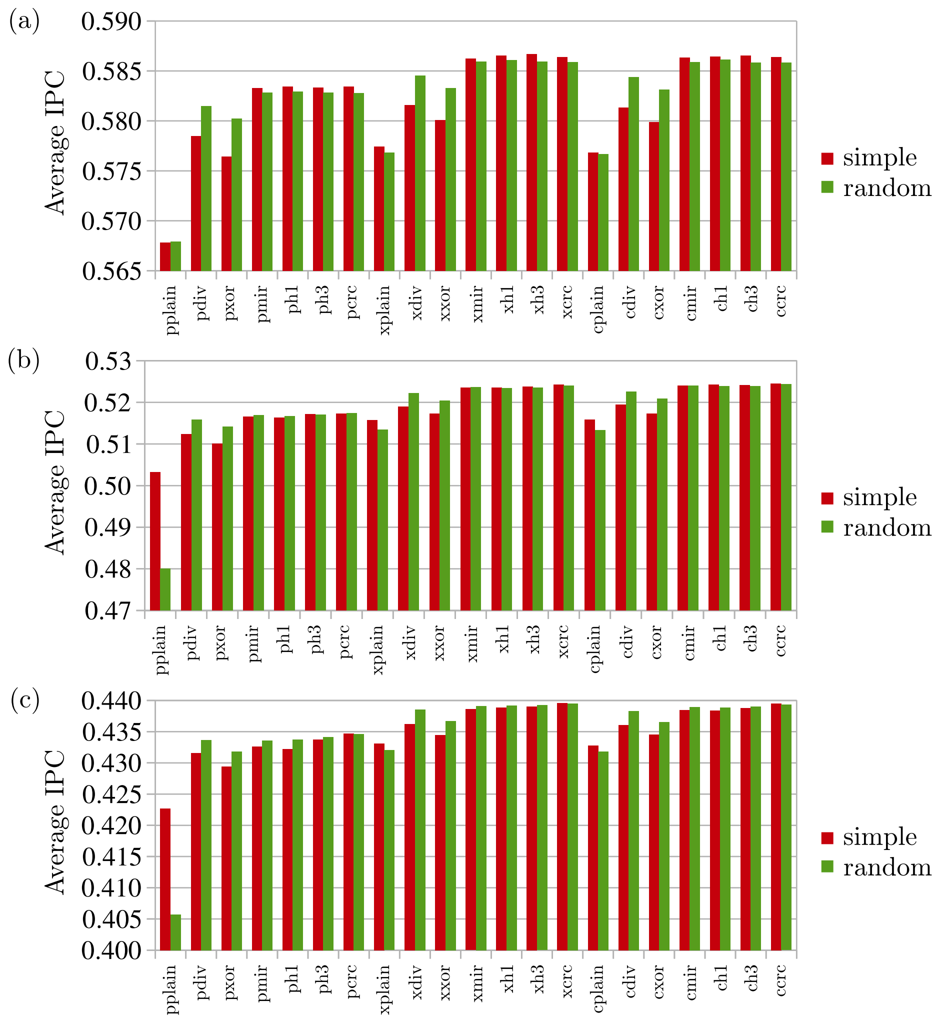

Figure 6. The avegage IPC in CPU2017 multi-threaded refrate tests for different interleaving, indexing, and address mapping with (a) 16 cores/banks, (b) 32 cores/banks, (c) 64 cores/banks

$$
\text{(a)}\ \begin{matrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\\1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{matrix}
\qquad
\text{(b)}\ \begin{matrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\\0&0&0&0&1\\0&0&0&1&0\\0&0&1&0&0\\0&1&0&0&0\\1&0&0&0&0\end{matrix}
\qquad
\text{(c)}\ \begin{matrix}1&1&1&0&1\\1&1&0&1&0\\1&0&1&0&0\\0&1&0&0&0\\1&0&0&0&0\\1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{matrix}
\qquad
\text{(d)}\ \begin{matrix}1&0&0&0&1\\0&0&0&1&0\\0&0&1&0&1\\0&1&0&1&1\\1&0&1&1&1\\1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\end{matrix}
$$

Figure 7. Hash matrices for 5-bit index size: a) shift; b) mirror; c) H_1; d) H_3

Table 2 summarises all the 21 indexing and interleaving combinations tested. The effect of hash use on the cache miss rate and IPC for individual tests is shown in Figure 8. It can be seen that only one test experiences a noticeable performance loss with the use of hashes, and miss rate reduction correlates with IPC increase even for tests with a moderate miss rate.

While the questions of optimal hash functions and appropriate address mapping are worth deeper study, several conclusions can be made as a result of this work:

Table 2. Indexing (first row) and interleaving (first column) combinations tested

	plain	division	shift	mirror	H_1	H_3	CRC
plain	plain	pdiv	pxor	pmir	ph1	ph3	pcrc
shift	xplain	xdiv	xor	xmir	xh1	xh3	xcrc
CRC	cplain	cdiv	cxor	cmir	ch1	ch3	crc

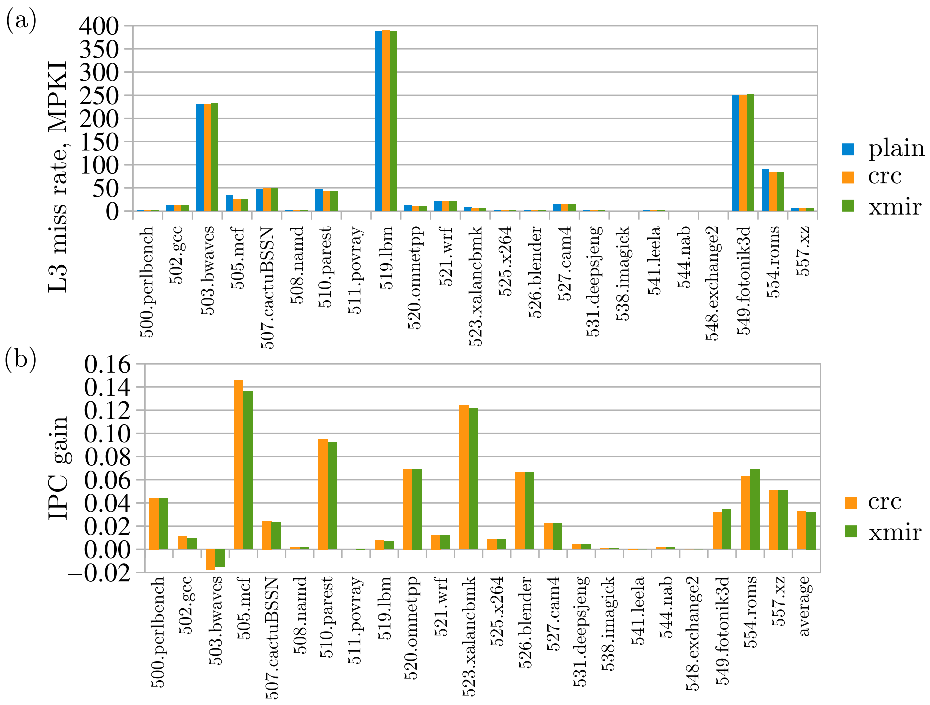

Figure 8. L3 cache miss rate (a) and relative increase in IPC (b) in CPU2017 multi-threaded refrate tests for simple (plain), XOR-based (xmir) or CRC-based (crc) interleaving and indexing (16 cores and banks)

- the method of mapping virtual addresses into physical ones can significantly affect simulation results when traditional cache indexing is used;
- any XOR-based indexing and interleaving considered yields a performance gain of several percent, comparable to division- and CRC-based;
- indexing and interleaving have a comparable effect and combine well;
- the cheapest and fastest XOR-based *mirror* indexing and *shift* interleaving are sufficient to achieve nearly the best results.

Although the effect on a real system might vary depending on page allocation, its very magnitude even with the simplest hash functions, along with negligible hardware cost, suggests that such an optimisation is worth primary attention.

Reducing L2-L3 replication

Another avenue of improvement can be found by noticing the comparable L2 and L3 cache capacity in the configuration considered — 1 MB and 2 MB per core, respectively — in combination with the read-allocate policy on both levels. Despite the non-inclusive scheme, in this case, up to a half of L3 cache contents is replicated in L2 and, therefore, unlikely to be used. The effective capacity of such a configuration will vary between 2 and 3 MB per core when all cores are active, while the broadly used exclusive scheme would give 3 MB exactly; this loss in capacity due to replication can be depicted as in Figure 9.

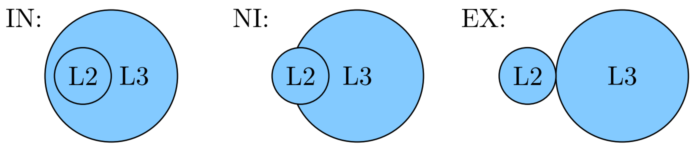

Figure 9. Data replication in the inclusive (IN), non-inclusive (NI), and exclusive (EX) cache schemes

Of course, read allocation could be regarded as the culprit and write allocation adopted instead. This would increase miss rates in many applications and thus is not considered an optimisation here. The switch to the exclusive scheme, implying write allocation, invalidation on hit, and writeback of unmodified lines, would also reduce performance in some cases due to the increased L2-L3 traffic and, less obviously, higher utilisation of the directory. While other remedies may exist in the field of promotion and replacement policies [Jaleel et al., 2010], a straightforward solution is to switch between the non-inclusive and the exclusive schemes dynamically depending on their effectiveness in the current scenario.

One such scheme called FLEXclusion was proposed in [Sim et al., 2012]. Based on statistics gathered in the L3 cache (set-duelling monitors), the optimal mode is periodically selected and assigned to each new cache entry. While the global mode can be changed at any moment, the mode of each cache line is fixed during its life in the caches. If the mode is non-inclusive, the line is read-allocated in L3 and then not written back unmodified from L2; if exclusive, both decisions are the opposite (Figure 10). Thus, the global mode can be switched at any time with no side effects, the overall performance drifting between that of the two basic modes as needed.

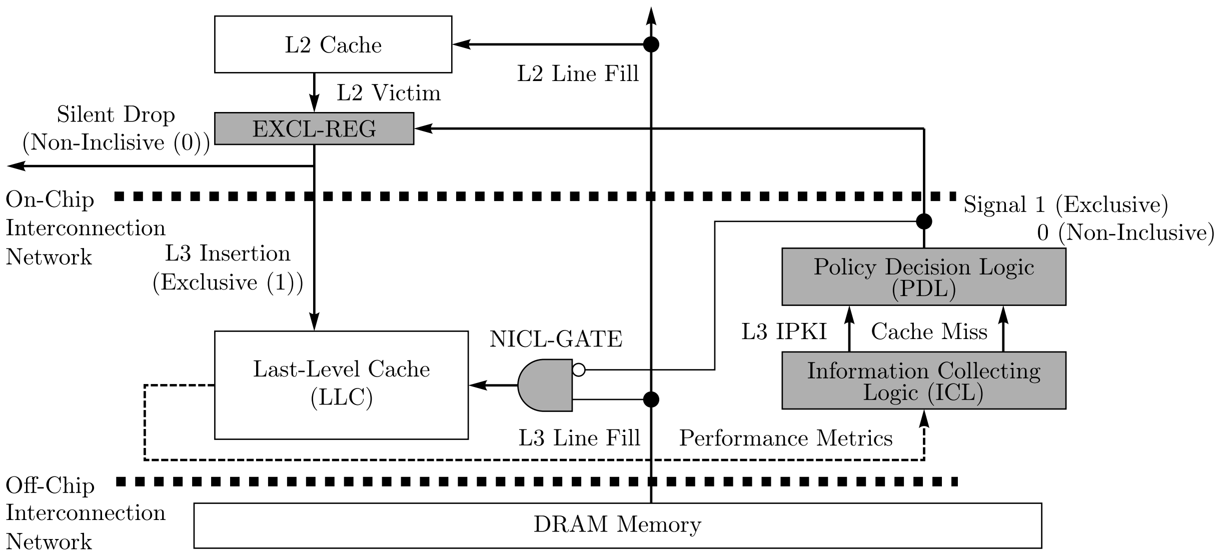

Figure 10. FLEXclusion cache scheme [Sim et al., 2012]

In the configuration of caches considered in this work, this scheme required two additional tweaks. Firstly, as the exclusive mode increases writeback frequency with negative effects on L2 cache performance, the mode switching should also be controlled by appropriate event counters at L2. Accordingly, L2 caches can inform the L3 cache about the preferred mode by using different request opcodes. Secondly, to allow for the increased utilisation of the directory in the exclusive mode, set-duelling monitors now treat directory misses like L3 misses: an L3 miss leading to directory allocation is counted as two misses.

The resulting reduction of the L3 miss rate and increase in IPC for exclusive (EX), original FLEXclusion (FC) and modified FLEXclusion (FL) schemes are shown in Figure 10. While the exclusive scheme yields mixed results, with an average IPC loss of 1.1 % due to poor directory performance, original FLEXclusion yields an average IPC gain of 1.4 %, and modified FLEXclusion performs with an average IPC increase of 2.2 %. When the same tests were run with hash-based L3 indexing discussed earlier, the negative effect was almost gone, and the exclusive scheme slightly outperformed both variants of FLEXclusion (Figure 12). In any case, the proposed modified FLEXclusion scheme appears to be a robust and flexible solution, which can be configured at any time to perform closer to either basic scheme by programming different switching thresholds. With narrower on-chip network channels, programs with high L2 and L3 cache miss rates like *503.bwaves* or *519.lbm* might benefit from such adaptivity.

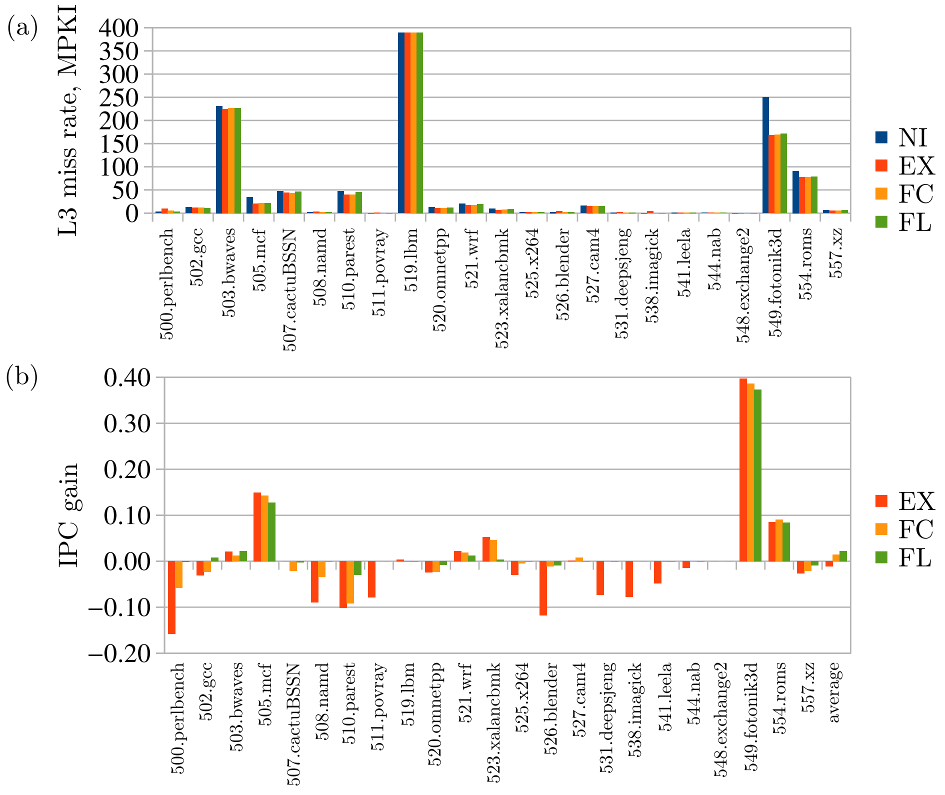

Figure 11. L3 cache miss rate (a) and relative increase in IPC (b) in CPU2017 refrate tests for non-inclusive (NI), exclusive (EX), original (FC) or modified (FL) FLEXclusion schemes (16 cores/banks)

More advanced optimisations of this type also exist [Wang et al., 2019] and are worth considering as further improvement.

Data compression

While the two previous approaches aim at more efficient use of the nominal cache capacity, there is a way of increasing the effective capacity by employing compression.

Cache compression methods can be divided into two categories. The first utilises the likeness of cache lines and encodes them together into fewer cache records [Sardashti, Seznec, Wood, 2016].

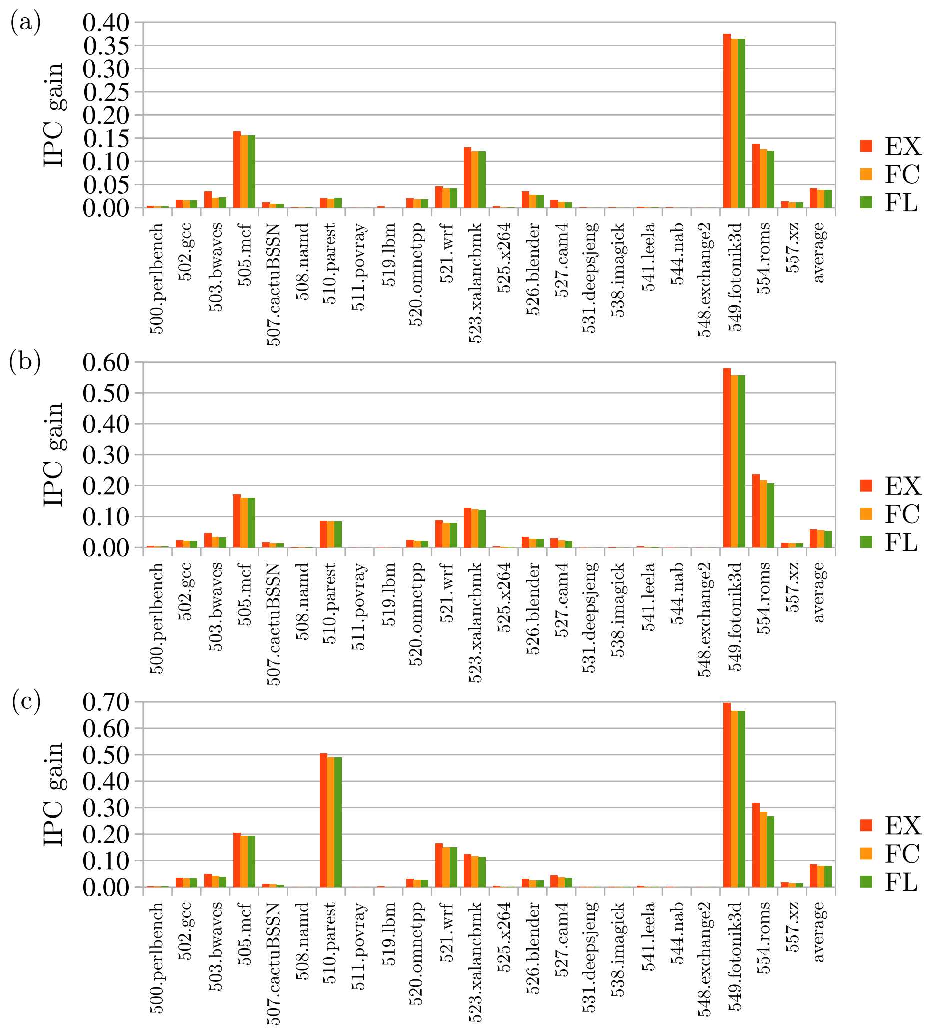

Figure 12. Relative increase in IPC in CPU2017 refrate tests for exclusive (EX), original (FC) or modified (FL) FLEXclusion schemes with XOR-based indexing and interleaving: a) 16 cores and banks; b) 32 cores and banks; c) 64 cores and banks

Another approach is to compress cache lines so multiple lines can fit into the space of one; this usually implies compressing the data and using a larger array of tags [Gaur, Alameldeen, Subramoney, 2016].

For the L3 cache with 64-byte lines and additional directory tags, considered in this work, the BDI*-HL (Base-Delta-Immediate Modified, Half-Line) algorithm of the second type was developed. With details discussed in [Kozhin, Surchenko, 2020], its basic idea is to organise the data array in half-lines, i. e. use 32-byte words to store 64-byte lines, compress cache line data twice where possible and thus store up to twice more cache lines in the cache using twice more tags (e. g. the extended tags implied by NCID). Figure 13 depicts how the compression works on 64-byte data. In this example, the data block is divided into 16 words, then each word is split into 3-byte left and one-byte right parts. The left part here is the same in all the words except one, where it equals zero, so the common left part, each right part, and the mask of words with zero left part can be stored as a 32-byte block. The decompression of the block consists of simple concatenation and multiplexing, adding negligible delay to data access.

In the NCID cache with an extended tag array, such an algorithm requires no additional storage space. This is depicted in Figure 14 for a 16-way cache set with 16 main and 16 directory tags. As the set is populated with valid data, they are compressed and occupy full (F), half (H) or no (Z) block, the

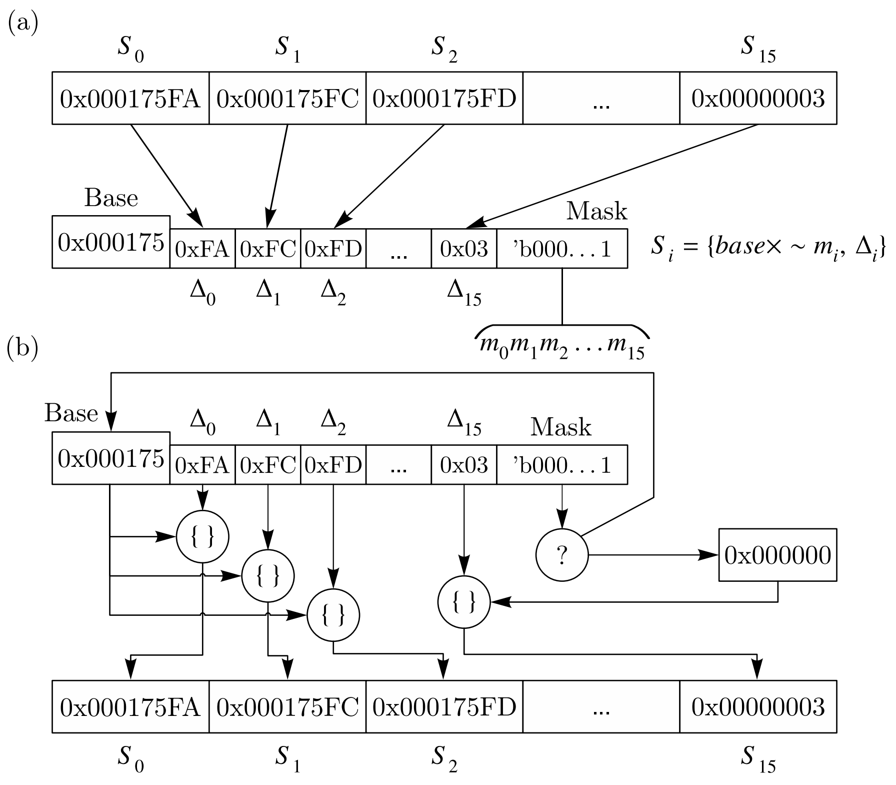

Figure 13. BDI*-HL cache line compression scheme: a) compression; b) decompression

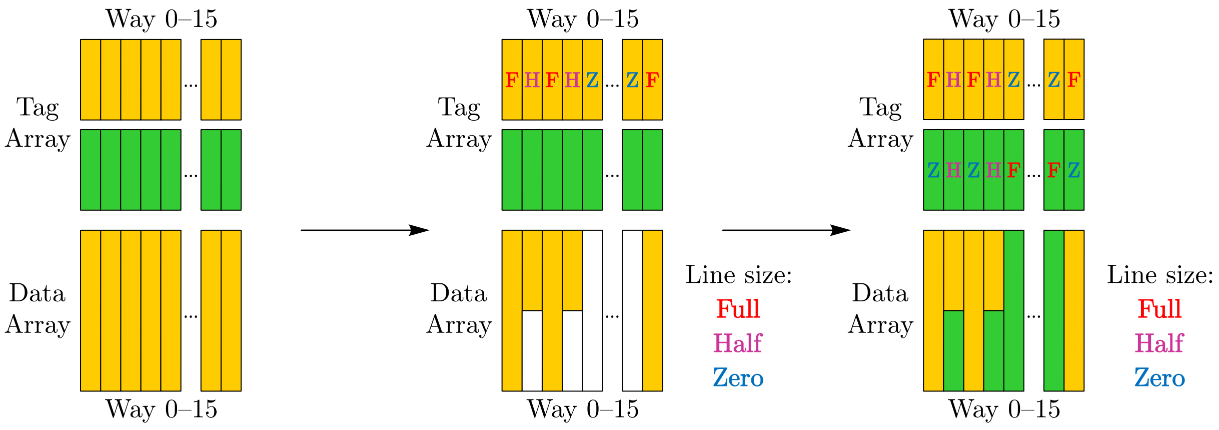

Figure 14. BDI*-HL compression in a NCID cache set

latter meaning 64 zero bytes. Empty spaces in the data array are then filled with new cache lines as long as their compressibility allows it, and directory tags are used.

Leaving the accurate simulation of the state machine implied by the algorithm as part of future work, the potential of such compression was evaluated in this study. First, the fraction of compressed cache lines in the L2-L3 traffic and in the L3 cache was measured; the results are shown in Figure 15, *a*. On average, it was 18 % and 17 %, respectively. Secondly, as the latter fraction for each test was known, the respective increase in the effective L3 cache capacity was emulated by proportionally increasing the number of L3 cache ways. The resulting reduction of the L3 miss rate is shown in Figure 15, *b* (three tests with the highest miss rate were unaffected and thus omitted). The increase in IPC is shown in Figure 15, *c*.

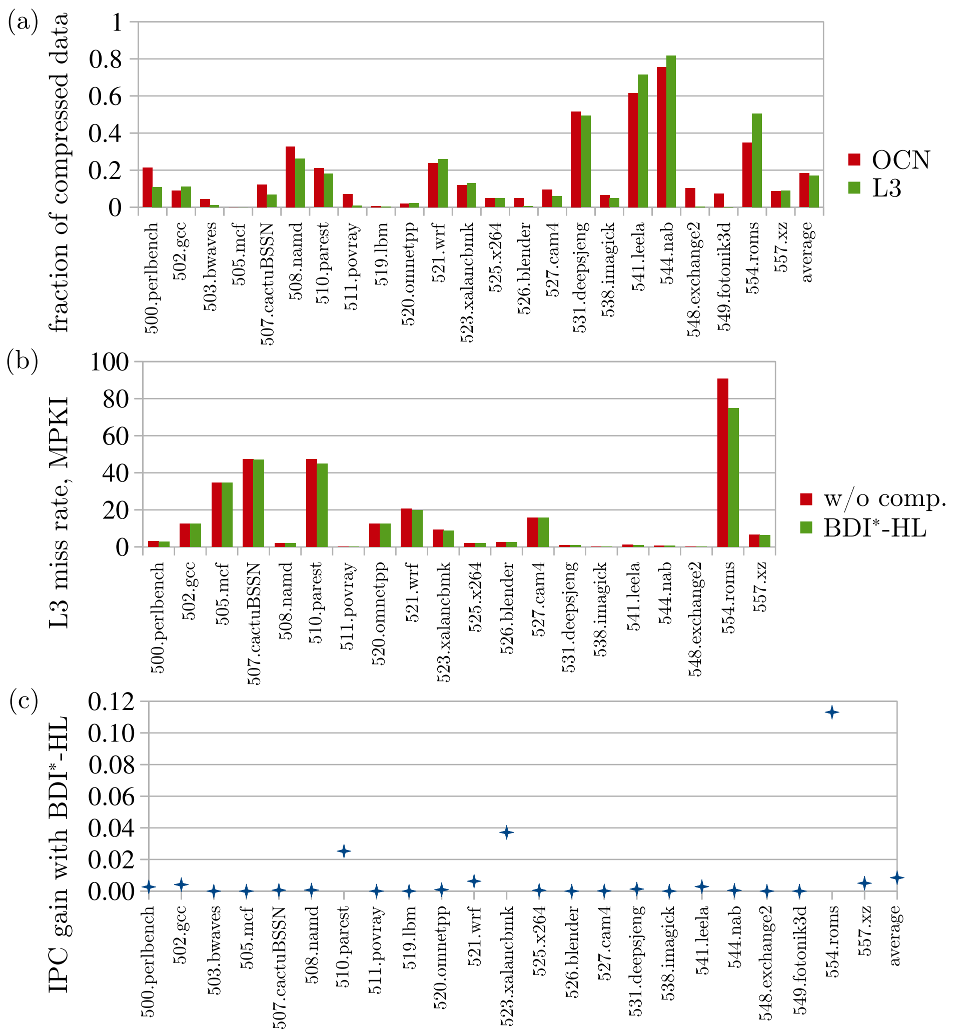

Figure 15. BDI*-HL (a) compression rate in the L2-L3 traffic (OCN) and the L3 cache, (b) L3 cache miss rate, (c) relative increase in IPC in CPU2017 refrate tests

Three tests (510.parest, 523.xalancbmk, 554.roms) show a performance gain from 2 % to 11 %. Interestingly, these have neither the highest L3 cache miss rate nor the highest data compressibility, as every test has its own dependence of the miss rate on available cache space.

On average, BDI*-HL compression yields around a 1 % increase in performance. As its cost is expected to be within 0.1 % of the total chip area, it could be a useful optimisation.

Results

The average CPU performance gain in SPEC CPU2017 refrate tests from all three optimisations discussed in this paper is summarised in Table 3. To show their cumulative effect, each optimisation was tested after the previous optimisation was applied.

Since all the configurations have the same memory bandwidth, the effect of each optimisation tends to increase with the core number.

Discussion

The cache miss rate substantially affects the performance of a multicore CPU, so even a large non-inclusive shared cache with inclusive directory has a potential for optimisations. Three types of optimisations were considered and demonstrated a significant effect.

Table 3. The average IPC gain with L3 cache optimisations

	16 cores/banks	32 cores/banks	64 cores/banks
XOR-based interleaving and indexing	3.2 %	9.1 %	8.2 %
Modified FLEXclusion	3.8 %	5.4 %	7.9 %
BDI*-HL compression	0.6 %	1.0 %	1.5 %
First two optimisations	7.1 %	15 %	17 %
All three optimisations	7.7 %	16 %	19 %

The first optimisation is aimed at more unifirm usage of cache banks and sets using hash-based interleaving and indexing. A XOR-based variety of hash functions was tested, and even the simplest functions, requiring no additional cache space and having minimal delay, demonstrated the average IPC increase in SPEC CPU2017 refrate tests of 3.2 %, 9.1 %, and 8.2 % for CPU configurations with 16, 32, and 64 cores and banks, comparable to the results of more complex matrix-, division- and CRC-based functions.

The second optimisation is to reduce replication at different cache levels by automatically switching to the exclusive scheme when it appears to be optimal according to cache hit and miss statistics. A known scheme of this type, FLEXclusion, was modified for NCID caches by accounting for the directory miss rate and L2 cache activity. In the experiments, it showed an additional performance gain of 3.8 %, 5.4 %, and 7.9 % for 16-, 32-, and 64-core configurations.

The third optimisation is to increase the effective cache capacity using compression. The compression rate of the inexpensive and fast BDI*-HL (Base-Delta-Immediate Modified, Half-Line) algorithm, designed for NCID, was measured, and the respective increase in cache capacity yielded roughly 1 % of average performance increase.

All three optimisations can be combined and demonstrated a performance gain of 7.7 %, 16 % and 19 % for CPU configurations with 16, 32, and 64 cores and banks, respectively.

References

Balasubramonian R., Jouppi N., Muralimanohar N. Multi-Core Cache Hierarchies. — Synthesis Lectures on Computer Architecture. — Morgan & Claypool Publishers, 2011.

Binkert N., Beckmann B., Black G., Reinhardt S. K., Saidi A., Basu A., Hestness J., Hower D. R., Krishna T., Sardashti S., Sen R., Sewell K., Shoaib M., Vaish N., Hill M. D., Wood D. A. The Gem5 Simulator // SIGARCH Comput. Archit. News. — 2011. — Vol. 39, No. 2. — P. 1–7.

Carvalho D. R., Seznec A. Understanding cache compression // ACM Trans. Archit. Code Optim. — 2021. — Vol. 18, No. 3. — 27 pp.

Chaudhuri M., Gaur J., Bashyam N., Subramoney S., Nuzman J. Introducing Hierarchy-awareness in replacement and bypass algorithms for last-level caches / 2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT). — 2012. — P. 293–304.

Cho S., Jin L. Managing distributed, shared L2 caches through OS-level page allocation / 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06). — 2006. — P. 455–468.

Cho S.-J., Choi U.-S., Hwang Y.-H., Kim H.-D. Design of new XOR-based hash functions for cache memories // Computers & Mathematics with Applications. — 2008. — Vol. 55, No. 9. — P. 2005–2011.

Ferdman M., Lotfi-Kamran P., Balet K., Falsafi B. Cuckoo directory: A scalable directory for many-core systems / 2011 IEEE 17th International Symposium on High Performance Computer Architecture. — 2011. — P. 169–180.

Gaur J., Alameldeen A. R., Subramoney S. Base-victim compression: An opportunistic cache compression architecture / 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). — 2016. — P. 317–328.

González A., Valero M., Topham N., Parcerisa J. M. Eliminating cache conflict misses through XOR-based placement functions / Proceedings of the 11th International Conference on Supercomputing. — ICS'97. — New York, NY, USA: Association for Computing Machinery, 1997. — P. 76–83.

Hardavellas N., Ferdman M., Falsafi B., Ailamaki A. Reactive NUCA: near-optimal block placement and replication in distributed caches // SIGARCH Comput. Archit. News. — 2009. — Vol. 37, No. 3. — P. 184–195.

Hennessy J. L., Patterson D. A. Computer architecture, sixth edition: A quantitative approach. — 6th edition. — San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

Herrero E., González J., Canal R. Distributed cooperative caching / Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. — PACT'08. — New York, NY, USA: ACM, 2008. — P. 134–143.

Iyer R., De V., Illikkal R., Koufaty D., Chitlur B., Herdrich A., Khellah M., Hamzaoglu F., Karl E. Advances in microprocessor cache architectures over the last 25 years // IEEE Micro. — 2021. — Vol. 41, No. 6. — P. 78–88.

Jagtap R., Diestelhorst S., Hansson A., Jung M., When N. Exploring system performance using elastic traces: Fast, accurate and portable / 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS). — 2016. — P. 96–105.

Jain A., Lin C. Back to the future: Leveraging Belady's algorithm for improved cache replacement / 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). — 2016. — P. 78–89.

Jaleel A., Borch E., Bhandaru M., Steely S. C. Jr., Emer J. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies / Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. — MICRO'43. — Washington, DC, USA: IEEE Computer Society, 2010. — P. 151–162.

Jiménez D. A., Teran E. Multiperspective reuse prediction / 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). — 2017. — P. 436–448.

Kharbutli M., Solihin Y., Lee J. Eliminating conflict misses using prime number-based cache indexing // IEEE Trans. Comput. — 2005. — Vol. 54, No. 5. — P. 573–586.

Kozhin A. S., Surchenko A. V. Design of data compression mechanism in cache memory of Elbrus processors / 2020 International Conference Engineering and Telecommunication (En&T). — 2020. — P. 1–5.

Martin M. M. K., Hill M. D., Sorin D. J. Why on-chip cache coherence is here to stay // Commun. ACM. — 2012. — Vol. 55, No. 7. — P. 78–89.

Nedbailo Yu. Fast and scalable simulation framework for large in-order chip multiprocessors / 2020 26th Conference of Open Innovations Association (FRUCT). — 2020. — P. 335–345.

Nedbailo Yu. A., Bychkov I. N., Chuchko P. A., Panchenko E. G., Slesarev M. V., Troosh A. I., Feldman V. M. Elbrus-2C3: a dual-core VLIW processor with integrated graphics / 2021 International Conference Engineering and Telecommunication (En T). — 2021. — P. 1–5.

Pekhimenko G., Seshadri V., Mutlu O., Kozuch M. A., Gibbons Ph. B., Mowry T. C. Base-delta-immediate compression: Practical data compression for on-chip caches / 2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT). — 2012. — P. 377–388.

Qureshi Ya. M., Simon W. A., Zapater M., Atienza D., Olcoz K. Gem5-X: A Gem5-based system level simulation framework to optimize many-core platforms / 2019 Spring Simulation Conference (SpringSim). — 2019. — P. 1–12.

Salwan H. Eliminating conflicts in a multilevel cache using XOR-based placement techniques / 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing. — 2013. — P. 198–203.

Sanchez D., Kozyrakis C. The ZCache: decoupling ways and associativity / Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. — MICRO'43. — Washington, DC, USA: IEEE Computer Society, 2010. — P. 187–198.

Sardashti S., Seznec A., Wood D. A. Yet another compressed cache: A low-cost yet effective compressed cache // ACM Trans. Archit. Code Optim. — 2016. — Vol. 13, No. 3. — 25 pp.

Seznec A. A new case for skewed-associativity: Research report RR-3208 / Seznec A.: INRIA, 1997.

Shilov A. Russian company tapes out 16-core Elbrus CPU: 2.0 GHz, 16 TB of RAM in 4-way system. — [Online]. — Available: https://www.tomshardware.com/news/russian-companytapes-out-16-core-elbrus-cpu-20-ghz-16-tb-of-ram-in-4-way-system.

Sim J., Lee J., Qureshi M. K., Kim H. FLEXclusion: balancing cache capacity and on-chip bandwidth via flexible exclusion // SIGARCH Comput. Archit. News. — 2012. — Vol. 40, No. 3. — P. 321–332.

Vandierendonck H., De Bosschere K. XOR-based hash functions // IEEE Transactions on Computers. — 2005. — Vol. 54, No. 7. — P. 800–812.

Wang J., Ramrakhyani P., Elsasser W., John L. K. Reducing data movement and energy in multilevel cache hierarchies without losing performance: Can you have it all? / 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT). — 2019. — P. 383–394.

Wang Z., Pang Z., Yang P., Xu J., Chen X., Maeda R. K. V., Wang Z., Duong L. H. K., Li H., Wang Z. MOCA: An inter/intra-chip optical network for memory / 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC). — 2017. — P. 1–6.

Zebchuk J., Qureshi M. K., Srinivasan V., Moshovos A. A tagless coherence directory / 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). — 2009. — P. 423–434.

Zhao L., Iyer R., Makineni S., Newell D., Cheng L. NCID: A non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies / Proceedings of the 7th ACM International Conference on Computing Frontiers. — CF'10. — New York, NY, USA: ACM, 2010. — P. 121–130.