# Minimizing Single-Usage Cache Pollution for Effective Cache Hierarchy Management

Thomas Piquet, Olivier Rochecouste, André Seznec

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1826

# MINIMIZING SINGLE-USAGE CACHE POLLUTION FOR EFFECTIVE CACHE HIERARCHY MANAGEMENT

THOMAS PIQUET, OLIVIER ROCHECOUSTE, ANDRÉ SEZNEC

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

# Minimizing Single-Usage Cache Pollution for Effective Cache Hierarchy Management *

Thomas Piquet, Olivier Rochecouste, André Seznec

Systèmes communicants
Projet CAPS

Publication interne n ° 1826 — Novembre 2006 — 19 pages

**Abstract:**

Efficient cache hierarchy management is of a paramount importance when designing high performance processors. Upon a miss, the conventional operation mode of a cache hierarchy is to retrieve back the missing block from higher levels and to store the block into all hierarchy levels. It is however difficult to assert that storing the block into intermediate levels will be really useful. In the literature, this phenomenon, referred to as cache pollution, is often associated with prefetching techniques, that is, a prefetched block could evict data that is more likely to be reused in a near future. Cache pollution could cause severe performance degradation. This paper is typically concerned with addressing this phenomenon in the highest level of cache hierarchy. Unlike past studies that treat polluting cache blocks as blocks that are never accessed (i.e. only due to prefetching), our proposal rather attempts to eliminate cache pollution that is inherent to the application. Our observations did indeed reveal that cache blocks that are only accessed once - single-usage blocks - are quite significant at runtime and especially in the highest level of cache hierarchy. In addition, most single-usage cache blocks are data that can be prefetched. We show that employing a simple prediction mechanism is sufficient to uncover most of the single-usage blocks. For a two-level cache hierarchy, these blocks are directly sent from main memory to L1 cache. Performing data bypassing on L2 cache maximizes memory hierarchy and allows hard-to-prefetch memory references to remain into this cache hierarchy level. Our experimental results show that minimizing single-usage cache pollution in the L2 cache leads to a significant decrease in its miss rate; resulting therefore in noticeable performance gains.

**Key-words:** Computer Architecture, Memory Hierarchy, Cache Pollution, Single-Usage Data, Block-Usage Prediction, Hardware Prefetching.

*(Résumé : tsvp)*

* This work was partially supported by an Intel research grant, an Intel research equipement donation and by the European Commission in the context of the SARC integraed project #27648 (FP6)

# Minimiser la pollution dans le cache pour une gestion efficace de la hiérarchie mémoire

**Résumé :** Cette étude propose de gérer plus efficacement la hiérarchie mémoire en minimisant la quantité de *données à usage unique* présentes au sein du cache. Nos analyses indiquent que cette forme de pollution est relativement importante à l'exécution et plus particulièrement dans le cache de second niveau. Nous proposons une solution matérielle permettant de prévenir l'insertion de ces données à usage unique dans le cache. Un mécanisme de prédiction simple est utilisé pour identifier ce phénomène.

**Mots clés :** Micro-Architecture, Hiérarchie Mémoire, Pollution de Cache, Données à Usage Unique, Prédition d'utilisation des données, Mécanisme de Préchargement Matériel.

# 1 Introduction

Processor performance is significantly impacted by the behavior of the memory hierarchy. Main memory access time accounts for several hundreds of cycles. To avoid such a huge penalty, modern processors feature a complete memory hierarchy including small L1 caches with 1–3 cycles access time, larger L2 caches with 8–15 cycles access time and often (shared) very large multi-megabytes L3 caches with 20–30 cycles access time. Moreover, processors also resort to prefetch mechanisms to avoid cache misses whenever possible.

On a cache miss, the conventional memory hierarchy management consists of propagating the missing block from the highest level in the memory hierarchy to the lowest level, each cache level getting a copy of the block. When this strategy is used, the cache hierarchy acts as a set of more and more efficient filters that retained the accesses. This strategy is in general quite efficient, since in case of a subsequent miss on the same block in a lower memory hierarchy level, the block remains accessible.

However, this strategy does not take in account that blocks have very different usages in applications. In particular, in some cases a block stored in the cache after a miss is not accessed again before eviction. We will call such a block, a *single-usage* block or SU-blocks. Storing a SU-blocks in the cache may have evicted another block which would have been otherwise useful, therefore generating an extra miss. In the remainder of the paper, we will refer to this phenomenon due to SU-blocks as *single-usage* pollution or SU-pollution.

Our first contribution in this paper is the characterization and analysis of SU-pollution phenomenon. Our analysis shows that, most applications only exhibit limited SU-pollution in the L1 data cache, but that many applications depict a significant amount of SU-pollution in the L2 cache. Our analysis also reveals that, in the L2 cache, the single-usage property of a block is often tight to the address of the load/store instruction that triggers the miss on the block.

Our second contribution is the proposal and the analysis of a hardware mechanism for reducing single-usage pollution. A single-usage block predictor is presented for the L2 cache. The address of the instruction that triggers the miss is used for predicting the single-usage property of the missing block. This prediction is highly accurate and has very high confidence on applications exhibiting significant amount of SU-pollution. Therefore on L2 misses, directly forwarding predicted single-usage blocks to the L1 cache without storing them in the L2 cache reduces L2 cache miss rate and improves the overall performance.

Our approach can be applied to all cache structures. In particular, it can be applied to cache structures that already target conflict misses, e.g. skewed associative caches [16] or prime number indexed caches [9]. Our approach can also take benefit from optimized replacement policies such as [14].

Our scheme is orthogonal with prefetching, since we try to retain blocks in the L2 cache while prefetching tries to hide memory latency. In practice, considering the popular stride prefetcher [5], we found that many SU-blocks are also blocks that can be prefetched. In addition, most of the logic of our single-usage predictor can be shared with the stride prefetcher. Therefore, the extra hardware complexity associated with our proposal is essentially limited to the addition of a few tag bits to each line in the L2 cache. Experiments showed that applying our guided L2 cache bypassing mechanism on top of a processor featuring stride prefetching results in a 15 % performance increase on the SPEC2000 FP applications and a 3% performance increase on the SPEC2000 INT applications.

The remainder of this paper is organized as follows. Section 2 discusses the related work and motivates this study. Section 3 presents the evaluation framework used for the experiments we conducted. Section 4 quantifies the amount of single-usage cache pollution in representative applications. Section 5 first provides evidence of the correlation of the single-usage property and instructions, then we present the block usage predictor. Section 6 presents our experimental results. Section 7 concludes this study.

# 2 Motivations and Related Work

In this section, we first recall the phenomenon of cache pollution that is generally associated with hardware prefetching techniques. We then introduce in more details the concept of single-usage pollution while reviewing a few related studies.

## 2.1 Cache pollution caused by hardware prefetching

We define cache pollution as storing an useless block in the cache. In the literature, the concept of cache pollution has often been associated with hardware prefetching [18, 20, 8, 11]. Hardware prefetching is an effective solution to improve the memory hierarchy behavior. The addresses of the memory blocks that the application will use in the near future are predicted, and blocks are brought back in the cache before their actual need or at least their latency access is partially hidden [2, 5, 7, 12]. Prefetched blocks are replacing other blocks in the cache. Hardware prefetching is speculative and therefore mispredictions can occur: when an incorrectly or too early prefetched block is written on the cache, it may evict some otherwise useful block from the cache.

Two approaches have been adopted to minimize cache pollution due to prefetching. One approach is to prohibit bad prefetch requests to be issued by means of a filter mechanism [18, 20]. Srinivasan et al. [18] discussed a feedback-directed technique to reduce the number of useless prefetches. These authors propose to selectively trigger prefetch requests based on information derived from profiling. Zhuang et al. [20] rather favored the usage of a hardware-based prediction mechanism to pro-actively control data prefetching. Besides reducing prefetching related cache pollution, these approaches also help decreasing the memory bandwidth usage. Due to their speculative nature, however, a number of useless prefetches might not be filtered out or, even worse, a significant fraction of the useful prefetches could be filtered; compromising thereby the prefetcher efficiency.

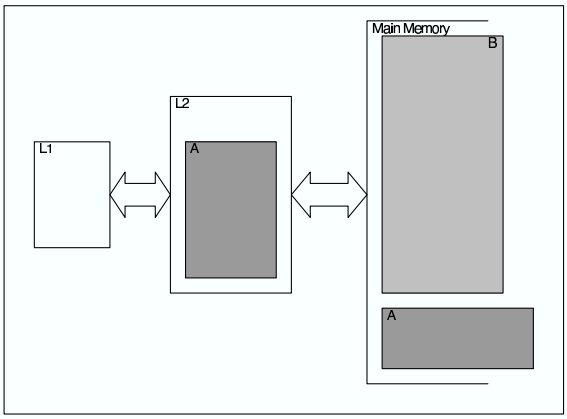
Another approach is to resort to a dedicated buffer to prevent ineffective prefetched data from entering the data cache. Jouppi [8] and Chen et al. [3] proposed to use small prefetch buffers to avoid unnecessary cache pollution. Prefetched data are first stored into these buffers instead of being directly stored into the cache. Upon a cache miss, a block is retrieved back from the prefetch buffer to the data cache when necessary. In our study, we rely upon such a prefetch buffer technique to filter out pollution associated with prefetching; thus allowing us to concentrate on eliminating single-usage cache pollution.

## 2.2 Single-usage cache pollution

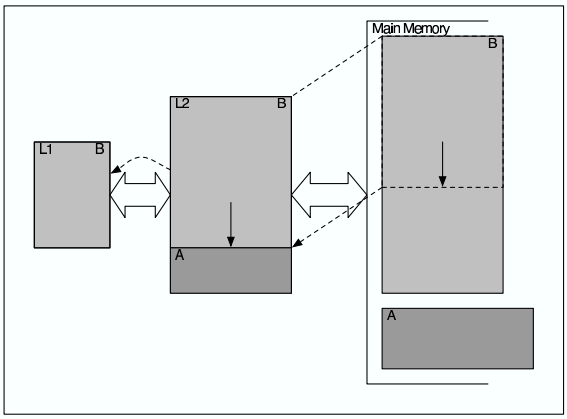
Our purpose also aims at reducing cache pollution for a more efficient utilization of memory structures. However, we are concerned with tackling a form of pollution that is inherent to the memory hierarchy usage.

Studies on prefetch pollution consider that a cache block is polluting when it is stored into the cache but never used before its eviction. In this study, we essentially address the blocks that are effectively accessed by the application. When a block is used once before its eviction, storing it in the cache can be considered as a form of pollution - a phenomenon we refer in this paper to as *single-usage cache pollution*. For instance, such an event occurs on the L1 cache when a scalar is read into a register and the data block is not accessed again before its eviction. Such cases on L1 cache are not so common. Single block usage before eviction occurs much more frequently on the highest cache levels (L2 or L3) since the L1 cache captures short term spatial and temporal locality. For instance, this can occur with an application where data structures featuring different sizes interact.

Let us illustrate this phenomenon with the scenario displayed in Figure 1. Let us assume that structure A fits into the L2 cache: after Phase 1, structure A lies in the L2 cache. Let us now suppose that during Phase 2, computation on structure B only involves temporal and spatial locality directly captured by the




(a) After Phase 1




(b) During Phase 2

Figure 1: Single-usage pollution example through execution phases.

L1 cache (i.e. no L1 cache miss is encountered apart the first access to the blocks). If structure B is larger than the L2 cache then the overall structure A has been ejected from the L2 cache at the end of Phase 2, and therefore has to be reloaded during Phase 3. The misses on the L2 cache during phase 3 could be avoided if the blocks from structure B were not loaded in the L2 cache, but simply loaded in the L1 cache.

## 2.3 Related work on single-usage pollution

Some researchers [4, 19, 6, 15] investigated methods to address the issues due to single-usage cache pollution. Chi and al. [4] proposed a software scheme to address single-usage cache pollution. The compiler determines for each memory reference its *cachability*. Since an architecture with a single cache level is considered, on a reference marked as *not cachable*, the data is not stored in the L1 cache. The

| Parameter | Configuration |
|---|---|
| Decode width | 4 |
| Issue width | 4 |
| Retire width | 4 |
| ROB size | 128-entry |
| LSQ size | 128-entry |
| Branch predictor | O-GEHL, 64-Kbit, 6-cycle mispred. penalty |
| Hardware prefetcher | 512-entry stride-prefetcher |
| Prefetch buffer | 32-entry, full-associative, FIFO |

Table 1: Simulated machine parameters.

most severe limitation of this software solution is that the proposed scheme does not take into account the spatial locality within a cache block.

Tyson et al. [19] observed that, on many applications only a few load instructions are responsible for the majority of data cache misses. They examined what would be the overall memory system performance if data referenced by these delinquent loads are not cached. Then they proposed two schemes, a static one and a dynamic one, to determine whether a load instruction should allocate space into cache or not. Our proposal is somewhat similar to theirs as we also suggest to determine which load instruction would cause single-usage cache pollution. Unlike these authors, we are not concerned with reducing cache pollution into the first level data caches as our experiments indicate that such a phenomenon is much more important into higher cache levels. Besides, we propose to integrate our mechanism into a hardware prefetch structure and analyze its impacts on the overall performance.

Johnson and al. [6] proposed a dynamic scheme to detect memory zones featuring short term usage on the L1 cache. When a cache miss occurs on a short term usage memory zone, the data is not stored in the main L1 cache, but in a small bypass buffer. Rivers and Davidson [15] suggested using a hardware mechanism aimed at capturing the same phenomenon on the L1 data cache, the non temporal streaming cache. Data blocks are classified in two categories : (1) Temporal block and (2) Non temporal (NT) block. A NT block is classified as a block where none of its words was re-referenced before its eviction. A NT bit is added to each block in the first and second level of cache. The main memory does not have the NT bit, therefore once a block evicted form the L2 cache, the information is lost. By contrast with these two proposals, our proposal addresses the L2 cache (or higher level). The blocks which exhibiting temporal locality and spatial locality that are completely captured by the L1 cache are not retained by the L2 cache. In our approach, we also associate the single-usage property to memory access instructions rather than to cache blocks in Rivers and Davidson proposal [15] .

# 3 Evaluation Framework

In this section, we describe the experimental environment used to evaluate our proposal.

## 3.1 Simulation methodology

Our experiments were performed using a modified version of MASE, an execution-driven simulator based on SimpleScalar [10]. Our baseline processor is a 4-way out-of-order superscalar with a 128-entry re-order buffer. It features a 64-Kbit O-GEHL branch predictor [17] with a 6-cycle misprediction penalty. The memory hierarchy configuration is derived after that of the PowerPC 970FX [1]. Tables 1 and 2 summarize the configuration we use as a reference. The main memory has a 140 cycles latency, and is able to deliver 16 bytes every 8 cycles. As our proposal is mainly concerned with memory hierarchy design, we modified MASE to model the memory bandwidth usage in details. The MASE model is augmented with the following components, a memory bus arbitrator, a write buffer and a prefetcher.

| Type | Size | Associativity | Block size | Replacement Policy | Latency |
|---|---|---|---|---|---|
| L1 instruction | 64-KByte | 1-way | 128 Bytes | LRU | 1 cycle |
| L1 data | 32-KByte | 2-way | 128 Bytes | LRU | 1 cycle |
| L2 (unified) | 512-KByte | 4-way | 128 Bytes | LRU | 10 cycles |
| Main memory | – | – | – | – | 140 cycles<br>16 bytes every 8 cycles |

Table 2: Memory hierarchy configuration.

The memory bus arbitrator deals with the following requests: (1) reads, (2) writes and (3) prefetch requests. Read requests are granted with the highest priority, if the bus is not occupied. In contrast, write and prefetch requests can only be serviced if the memory bus is free. Otherwise, if the memory bus is saturated, write requests are stored into write buffers while prefetch requests are simply rejected. The write buffer features 16 entries.

Modeling a prefetcher is essential to get a realistic view of the effective behavior of the memory hierarchy. Our study aims at addressing the single-usage pollution inherent to a program. Therefore, it was very important for the clarity of the evaluation to avoid prefetch related pollution of the L2 cache. Therefore, we model a prefetch buffer. This buffer is a 32-entry fully associative FIFO. We assume that the L2 cache and the prefetch buffer are probed in parallel. On a miss in the L2 cache, if the prefetch buffer hits, the requested block is allocated in the cache.

When using such a prefetch buffer, the structure of the prefetcher of course impacts on the overall performance of the processor, but has no influence on the observed single-usage pollution, apart marginally on blocks fetched on the wrong path. In this paper, we have considered a stride prefetcher [5] since many independent studies [13, 2] have shown that this prefetcher is quite efficient and easy to implement. The modeled stride prefetcher features a 512-entry direct-mapped prediction table (SPT). Entries in SPT are composed of three fields: (1) the instruction address, (2) the last memory address and (3) a valid bit. The stride prediction table is indexed with the instruction address. The data prefetcher is activated on each L2 cache access, as follows. On a L2 cache access, the stride prefetcher is accessed, it determines whether or not it should triggers a prefetch. The L2 cache is searched for the to-be-prefetched block. In case of a miss, the prefetch is activated. In order to allow in-time prefetch, the block to be prefetched is not the next block, but the next after the next block.

### 3.2 Benchmarks selection

As a benchmark set, we first use the whole set of SPEC CPU2000 benchmarks, apart *sixtrack* due to unsupported system calls. The applications were compiled for the Alpha ISA with the `-fast` and `-O4` optimization flags enabled. For every benchmark, the reference input data was used. The first billion instructions were skipped and the next billion instructions were simulated.

## 4 Characterizing single-usage pollution

In this section, we quantify the single-usage cache pollution that lies into applications. We will discriminate the dynamic occurrences of cache accesses between accesses to *single-usage blocks* (SU-blocks) - blocks that are not read back before cache eviction - and accesses to *multiple-usage blocks* (MU-block) - blocks that are read back before their cache eviction. In order to have a rough estimate of the number of single- and multiple-usage blocks processed at runtime, we gathered some profiling information for the SPEC2000 applications on the framework described in Section 3.

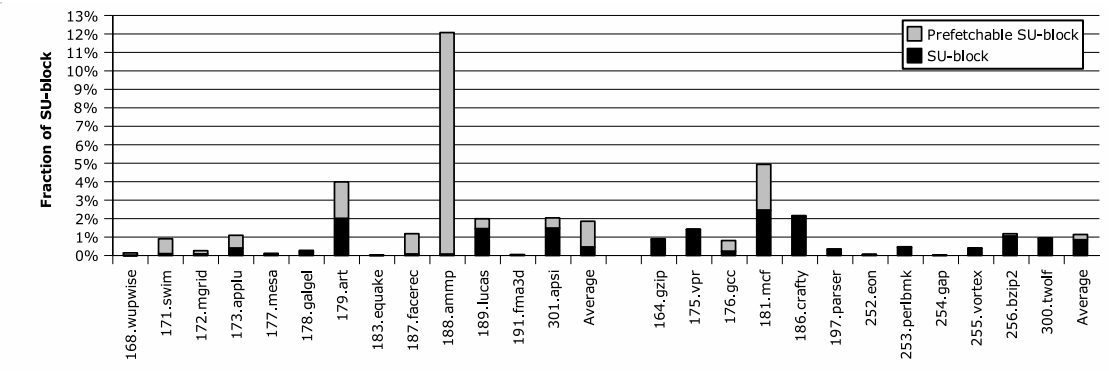



Figure 2: Single-usage pollution within L1 data cache.

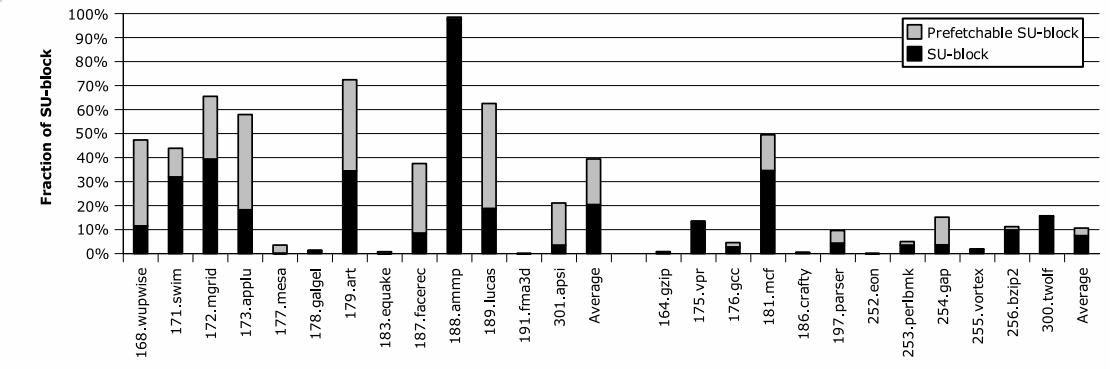



Figure 3: Single-usage pollution within L2 cache.

## 4.1 Quantifying SU-pollution in L1 cache

We quantified the number of accesses to single-usage blocks. A cache block is defined as single-usage if only one of its words is read before the block is evicted. Results are displayed in Figure 2. Note that this figure discriminates single-usage blocks that could be have been prefetched with a stride prefetcher (i.e. prefetchable SU-blocks). This property will be further discussed in Section 5.4. As depicted, the SU-pollution phenomenon is small in the lowest cache level since only around 2% on average of the dynamic accessed blocks are single-usage. This mainly stems from the fact that data contained in L1 cache exhibits high spatial and temporal localities. In our context, this means that attempting to reduce SU-pollution into the L1 cache would only have a small impact on overall performance.

## 4.2 Quantifying SU-pollution within L2 cache

We also measured the number of dynamic accesses to single-usage blocks for the L2 cache assuming a write back L2 cache. For the L2 cache, a block is considered to be single-usage even though it is written back just after its eviction from the L1 cache. The block is considered multiple usage only if it is read back before being evicted from L2 cache. Or in other words, we do not count a block write-back as a second access, since this write back could be directly forwarded to memory. Figure 3 reveals the number of SU-blocks contained into the L2 cache. We notice that the number of SU-blocks is strongly dependent upon the application type. For instance, for the floating-point application `ammp`, 98% of the dynamic

blocks accessed on the L2 cache are single-usage. In contrast, the integer application `crafty` depicts a poor level of SU-pollution. On average, our results also show that the SU-pollution phenomenon is much higher on floating-point programs that depict around 40% of dynamic accesses to SU-blocks on average. The phenomenon is much more limited on the integer benchmarks: only 10% of the dynamic accesses on the L2 cache constitute accesses to SU-blocks. The scenario presented in Section 2 is more likely to occur in scientific programs, and the majority of SPEC FP are scientific programs.

**Impact of cache parameters**

We modified the main L2 cache parameters - i.e. cache size, cache associativity and block size - to analyze their impact on SU-pollution. Figure 4 depicts the average amount of accesses to SU-blocks for the SPEC2000 applications. SU-pollution appears to be sensitive to cache size variation as the number of SU-blocks decreases with higher cache capacities. Increasing the cache size allows to preserve data longer into this structure and increases accordingly the data probability to get processed later on. On the other hand, varying the cache associativity has a only marginal effect on SU-pollution. This mainly means that address aliasing does not introduce significant extra SU-pollution. We also varied is the cache block size, but maintaining equal L1 and L2 block sizes. We observed for block sizes ranging from 64-byte to 256-byte, the ratio of accesses to SU-blocks remains in the same range.

### 4.3 Narrowing benchmarks selection

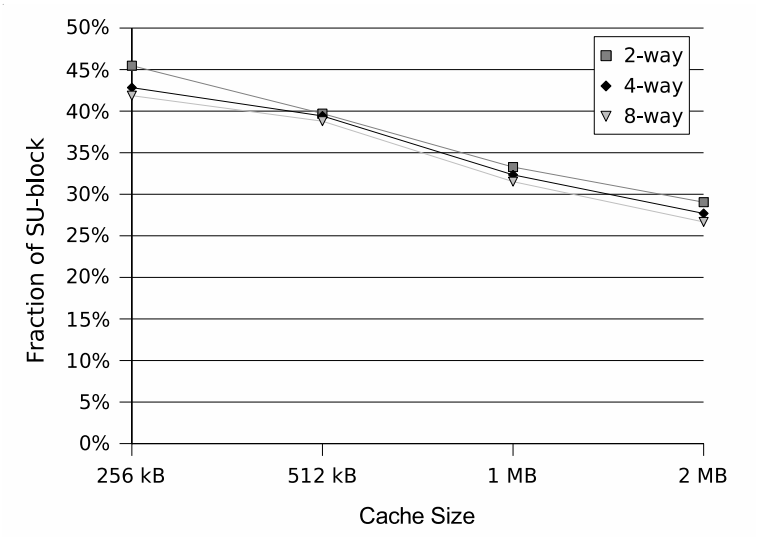
The analysis that we present above reveals that a significant fraction of the blocks found in L2 cache are single-usage. A number of applications exhibit, however, a negligible amount of single usage blocks - i.e. less than 2%. For readability sake, results pertained to these applications will not appear in most figures of the evaluation section. Let us note that these applications do not experience performance degradation and averages reported in Section 6 refer to all SPEC benchmarks. We will conduct our experiments using 9 floating-point applications: `wupwise, swim, mgrid, applu, art, facerec, ammp, lucas, apsi` and 8 integer applications : `vpr, gcc, mcf, parser, perlbmk, gap, bzip2, twolf`.

## 5 Predicting and bypassing single-usage blocks

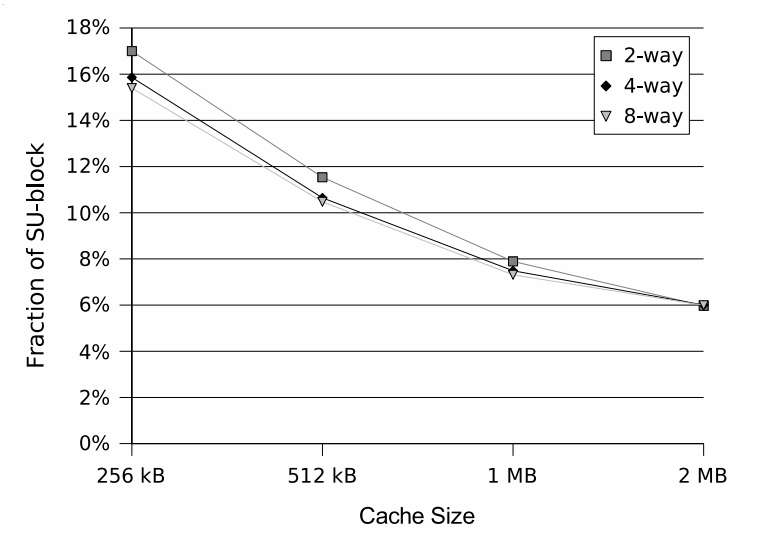
Being able to discriminate single-usage blocks from multiple-usage blocks for the L2 cache would enable us to bypass the SU-blocks directly from memory to the L1 cache without polluting the L2 cache; thus improving the overall performance. In this section, we first show that the single-usage property of a dynamic block in respect with L2 cache is strongly correlated with the instruction that triggers the L2 cache miss. This leads us to propose a simple block-usage prediction mechanism. We then show that a stride prefetcher could be adapted at a minimum extra hardware cost for predicting cache blocks usage.

### 5.1 Single-usage property is associated with the instruction

In order to speculate over the cache block usage property, one can consider the cache block address itself and quantify its usage over time. For instance, Johnson et al. used this approach in [6] for directing data placement. However, we show below that the cache block usage property is often tight to the address of the instruction that triggers the access to the block in main memory.




(a) SPEC FP




(b) SPEC INT

Figure 4: Ratio of SU-blocks on L2 cache for different cache configurations.

### 5.1.1 Quantifying single-usage I-sequences

We refer to the sequence of L2 cache accesses initiated by a single program instruction as *an I-sequences*. This section evaluates the correlation existing between the SU property of a data block and the instruction that triggers the access to the L2 cache. For this quantification, we define a single-usage I-sequences as

| SPEC FP | identified SU-blocks |
|---|---|
| 168.wupwise | 85.53% |
| 171.swim | 97.89% |
| 172.mgrid | 91.23% |
| 173.applu | 92.32% |
| 179.art | 88.53% |
| 187.facerec | 90.62% |
| 188.ammp | 94.48% |
| 189.lucas | 94.51% |
| 301.apsi | 91.55% |
| FP Average | 88.26% |

| SPEC INT | identified SU-blocks |
|---|---|
| 175.vpr | 72.94% |
| 176.gcc | 92.81% |
| 181.mcf | 90.75% |
| 197.parser | 94.13% |
| 253.perlbmk | 93.68% |
| 254.gap | 93.69% |
| 256.bzip2 | 93.25% |
| 300.twolf | 91.49% |
| INT Average | 90.03% |

Table 3: Ratio of SU-blocks identified by SU I-sequences.

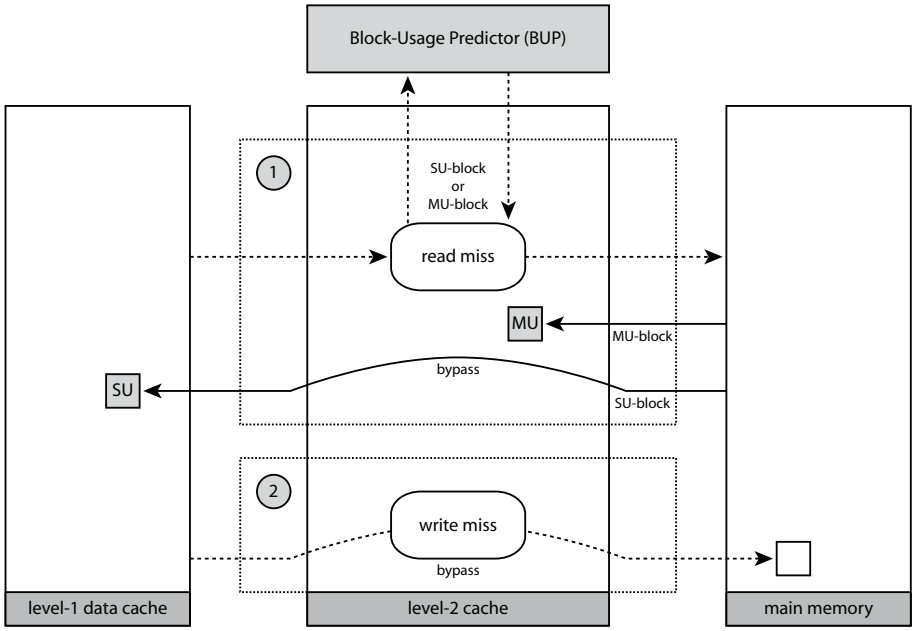



Figure 5: SU-blocks L2 cache bypassing

a I-sequences for which the amount of SU-blocks exceeds 95%. Note that the SU I-sequences property depends on the memory hierarchy configuration. Our measurements take into consideration only the I-sequencess that execute over 100 accesses into the L2 cache.

Table 3 reports the amount of single-usage blocks that are effectively accessed by these SU I-sequences. On average, more than 90% of SU-blocks are referenced by SU I-sequences. Hence, a mechanism able to detect SU I-sequences at runtime could therefore help to address SU-pollution. We propose such an hardware-based prediction mechanism in the next section.

## 5.2 SU-blocks bypassing principles

Figure 5 illustrates the principle we use for bypassing SU-blocks. A block usage predictor, or BU predictor is used to control this bypassing. On a miss on the L2 cache, the BU predictor predicts whether the

missing block will be single-usage or not. On a SU prediction, the block is directly sent to the L1 cache; bypassing thereby the L2 cache to prevent single-usage pollution. In contrast, the block is processed in a conventional fashion when predicted as multiple-usage.

The experiment presented in Section 5.1 shows that the single-usage property is very tight to the instruction triggering the memory access. We present below the principles of a simple block-usage prediction mechanism. The BU predictor tries to identify SU I-sequences. Blocks accessed in main memory by I-sequences identified as SU are stored only on the L1 cache. This prevents SU-pollution in the L2 cache.

**Structure** The BU predictor mainly consists of a block-usage prediction table and on two extra tags associated with each L2 cache line. Each entry in the block-usage prediction table consists of two fields: 1) the instruction address (IA) and 2) a saturated single-usage detection (SUD) counter. Two tags attached to a L2 cache line are the SU tag, a single bit that records whether or not the block has been reaccessed after being stored into the L2 cache, and the instruction address tag (or IA tag) that records the partial tag of the address of the instruction generating the miss.

The predictor operates in two main phases : 1) query and 2) update.

**Query** On a miss on the L2 cache, a query is sent to the BU predictor. The address of the instruction responsible for the miss is used as an index. If an entry matches, the predictor delivers a SU or non-SU verdict depending on SUD value. A SU verdict is only delivered upon a saturated SUD value. If there is no match then a non-SU verdict is delivered.

**Update** The BU table is updated whenever a block is evicted from L2 cache. The IA tag of the evicted block is used to get the corresponding I-sequences in the BU table. The SUD counter associated with the BU table entry is updated according to the SU tag of the block. If the tag indicates that the block is single usage, the counter is incremented, otherwise the counter is reset to zero.

The BU predictor can suffer from two misprediction types : 1) the block is predicted as non-SU, but was single usage and 2) the block is predicted as single usage but might have been accessed several times if it was stored into the L2 cache - this is referred to as SU misprediction.

In our proposal, a SU prediction results in bypassing the L2 cache: a SU misprediction automatically results in an extra L2 miss. On the other hand, falsely classifying a block as non-SU does not automatically result in an extra L2 miss. Therefore a block-usage predictor should favor the accuracy of the SU verdict against the accuracy of the non-SU verdict. Our design takes this aspect into account by delivering SU verdicts exclusively on saturated SUD states and by resetting the SUD counters on any non-SU update. However, when a I-sequences has become recognized as SU, the blocks are not loaded in the L2 cache: the SUD counter may remain saturated forever, while the the behavior of the application may change. To avoid such a scenario, all the SUD counters are periodically reset. We chose to base the reset frequency upon the number of updates done to the block-usage predictor. The influence of varying the reset frequency on the predictor performance is studied in Section 6.2.

## 5.3 Block usage prediction, cache inclusion and write policy

Due to bypassing, our scheme is not compatible with maintaining cache inclusion. When cache inclusion is enforced, write through L1 caches is generally preferred, since this design simplifies the coherence mechanism. But when inclusion is not enforced, maintaining cache coherence requires to check both the L1 and L2 caches. Using a *write-back* policy on the L1 cache is therefore possible.

We measured the fraction of SU-blocks assuming a L1 write-through cache. This number is significantly lower than when a L1 write-back cache is used: When a write-through L1 cache was used, a significant fraction of the multiple-usage blocks were due to burst of writes on a cache block.

Therefore in the remainder of the paper, we will consider a *write-back* L1 cache. On a write-back from the L1 cache, the L2 cache is first searched. On a miss on the L2 cache, the block is not written back into the L2 cache but directly written onto the main memory.

### 5.4 Extending a stride prefetcher for block-usage prediction

There exists a strong correlation between the SU property of a block occurrence and its prefetchability by a stride prefetcher. A stride prefetcher also exploits correlation of memory access address sequences with load/store addresses. In a simple experiment, we used a stride prefetcher to tag each single-usage block as prefetchable or not. The stride prefetcher was augmented with a prefetch buffer [8] to filter out most prefetch related pollution. Results are illustrated in Figure 3 (see Section 2). A majority of the SU blocks can be prefetched at runtime by a stride prefetcher.

Therefore, one can attempt to exploit existing structures of a stride prefetcher for implementing a BU predictor at a small hardware cost. The stride prefetch mechanism also relies on memory access I-sequences to identify strides and to initiate data prefetch. Besides, a stride prefetcher stores all memory access I-sequences and is not limited to those that generate a prefetch. So, the only addition necessary to augment a stride prefetcher with the block usage prediction functionality is to add to each stride prefetcher entry a SUD saturating counter.

**Storage overhead of the BU predictor** Augmenting the stride prefetcher with the block-usage predictor functionality can be done at a insignificant storage overhead: a 3-bit SUD counter is added to each entry. The main storage overhead induced by the BU predictor is the additional tags on the L2 cache. In our experiments, each cache line in L2 cache is augmented with the 1-bit SU tag and a 13-bit IA tag (9-bit are actually used to index the 512-entry stride prefetcher table, the purpose of the remaining 4-bit is to reduce aliasing). For the 128-byte cache blocks considered in the paper, these extra tags account for 2% of the cache storage budget.

## 6 Experimental evaluation

In this section, we provide the performance evaluation of the BU predictor associated with bypassing the L2 cache. We first evaluate the performance of the BU predictor itself in terms of accuracy and coverage. Then we analyzed the global impact of using block-usage prediction to enable bypassing the L2 cache.

### 6.1 SU coverage and accuracy

We define the SU prediction coverage as the fraction of the number of L2 misses on SU-blocks that are correctly classified as SU-blocks. We define the SU prediction accuracy is defined as the fraction of correct SU verdicts. Table 4 reports BU coverage and accuracy for the selected SPEC benchmarks. In our experiments, we used a 512-entry table comprised of 3-bit SUD counters with a reset frequency set to 500000 updates.

On average, our BU predictor performs much better on SPEC floating-point applications than on SPEC integer programs. On the floating-point applications the coverage of the block-usage predictor is generally high and its accuracy also. For some applications such as `vpr`, `twolf`, or `apsi`, the BU predictor exhibits poor coverage. Our dynamic BU predictor is much more conservative that the post-mortem classification presented in Section 5 since I-sequences are classified SU only when the 3-bit SUD counter is saturated. Such a conservative classification allows to avoid extra L2 misses due to incorrect SU verdicts. However, for `parser`, the accuracy of the SU-verdict is still relatively poor.

| SPEC FP | Coverage | Accuracy | SU-blocks ratio | # cache accesses (*M) |
|---|---|---|---|---|
| 168.wupwise | 76.58% | 98.57% | 47% | 1.96 |
| 171.swim | 97.55% | 99.92% | 44% | 14.35 |
| 172.mgrid | 90.77% | 98.35% | 66% | 6.82 |
| 173.applu | 95.14% | 99.05% | 58% | 11.74 |
| 179.art | 92.34% | 99.34% | 72% | 67.48 |
| 187.facerec | 94.51% | 95.71% | 38% | 9.62 |
| 188.ammp | 99.86% | 99.99% | 98% | 91.86 |
| 189.lucas | 96.15% | 99.53% | 63% | 8.49 |
| 301.apsi | 27.63% | 95.9% | 21% | 9.94 |
| FP Average | 78.48% | 90.94% | 20.39% | 28.43 |

| SPEC INT | Coverage | Accuracy | SU-blocks ratio | # cache accesses (*M) |
|---|---|---|---|---|
| 175.vpr | 2.04% | 77.63% | 14% | 21.38 |
| 176.gcc | 39.55% | 95.85% | 5% | 22.22 |
| 181.mcf | 94.23% | 99.86% | 50% | 46.54 |
| 197.parser | 44.59% | 65.19% | 10% | 7.79 |
| 253.perlbmk | 64.3% | 98.89% | 5% | 8.67 |
| 254.gap | 89.82% | 96.90% | 16% | 3.03 |
| 256.bzip2 | 17.08% | 92.44% | 11% | 7.18 |
| 300.twolf | 2.18% | 70.53% | 16% | 19.51 |
| INT Average | 36.97% | 79.14% | 7.46% | 20.27 |

Table 4: SU coverage and accuracy for a 512-entry direct-mapped BU table with 3-bit SUD counters and a 500,000 updates reset frequency.

## 6.2 Impact of various BU predictor parameters

We varied the main BU predictor parameters (number of predictor entries, with of the SUD counters and reset frequency) to study their influence on coverage and accuracy. With a prediction table size ranging from 128 to 1024 entries, the variation in accuracy is quite negligible on average. Increasing the number of entries has a positive, but still a small impact on coverage. Increasing the SUD counter width decreases the coverage of the BU predictor but increases the SU-verdict accuracy. respectively with 44 % and 65 % when using 3-bit counters. Adapting the reset frequency (from 5K to 5M updates) has almost no effect on BU accuracy. However, the predictor coverage is much more sensitive to reset frequency variation. Our results indicate that a suitable trade-off between the predictor efficiency and its hardware complexity is to use a 512-entry predictor table comprised of 3-bit saturating counters reset every 500000 updates.

## 6.3 Overall performance impact

In this section, we analyze the benefits of our proposal when added to the base architecture, and the base architecture augmented with a stride prefetcher respectively. All reported results are normalized to the baseline model. In Figures 6, 7 and 8, the first bar represents the baseline model augmented with block-usage prediction and L2 bypassing - referred to as base+BU implementation. The second bar is the baseline model augmented with stride prefetching,- referred to as SP implementation. The third bar represents the baseline model with a stride prefetcher augmented with block-usage prediction and L2

| SPEC FP | IPC | L2 Miss Rate | AMAT |
|---|---|---|---|
| 168.wupwise | 1.57 | 56% | 8.16 |
| 171.swim | 0.63 | 59% | 16.97 |
| 172.mgrid | 0.94 | 69% | 7.38 |
| 173.applu | 0.69 | 61% | 25.22 |
| 179.art | 0.31 | 73% | 159.11 |
| 187.facerec | 0.97 | 41% | 43.93 |
| 188.ammp | 0.06 | 99% | 83.17 |
| 189.lucas | 0.61 | 61% | 69.82 |
| 301.apsi | 1.34 | 33% | 17.42 |
| FP Average | 1.23 | 43.53% | 34.31 |

| SPEC INT | IPC | L2 Miss Rate | AMAT |
|---|---|---|---|
| 175.vpr | 0.57 | 23% | 10.32 |
| 176.gcc | 1.28 | 7% | 3.62 |
| 181.mcf | 0.24 | 53% | 138.5 |
| 197.parser | 1.14 | 18% | 7.98 |
| 253.perlbmk | 1.47 | 5% | 5.79 |
| 254.gap | 1.92 | 15% | 2.26 |
| 256.bzip2 | 1.54 | 22% | 4.92 |
| 300.twolf | 1.34 | 33% | 19.91 |
| INT Average | 1.43 | 14.54% | 16.65 |

Table 5: IPC, miss rate and AMAT for the baseline architecture.

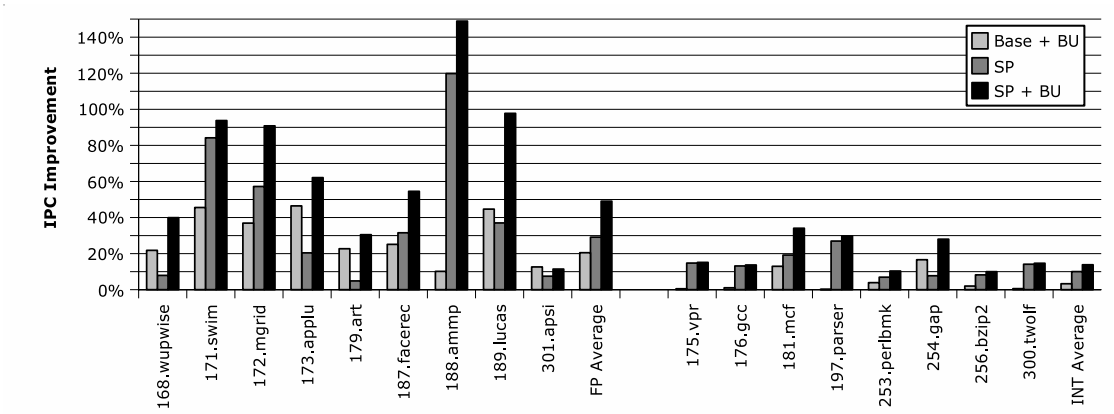



Figure 6: Relative performance gains.

bypassing - referred to as SP+BU implementation. Table 5 depicts the IPC, miss rate and AMAT, i.e., average memory access time, for the baseline architecture.

### 6.3.1 Performance impact

Figure 6 depicts the performance improvement - in terms of IPC - for the different processor configurations. Reducing single-usage cache pollution within the L2 cache has always a positive impact on overall performance. In particular, our mechanism significantly improves the average performance of the floating-point benchmarks, both as a standalone add on the baseline configuration and as an add to the SP configuration.

For the base+BU configuration, an average speed-up of 14 % is observed over the base architecture, some of the floating point benchmarks even experience speed-up higher than the speed-up obtained with using the stride prefetcher, e.g. `wupwise`, `applu`, `art`, `lucas` and `apsi`. On the other hand, on most integer benchmarks, the impact of using bypassing and BU prediction alone is more modest, 3% in average. However some significant speed-up are experienced on some applications, e.g., `gap` (18%) and `mcf` (13%).

It is noticeable that stride prefetching and BU prediction are orthogonal in most cases, i.e., the benefit of the SP+BU implementation over the base architecture is close to the sum of the performance gains resulting from using BU and a stride prefetcher respectively. For `wupwise` or `ammp`, the performance

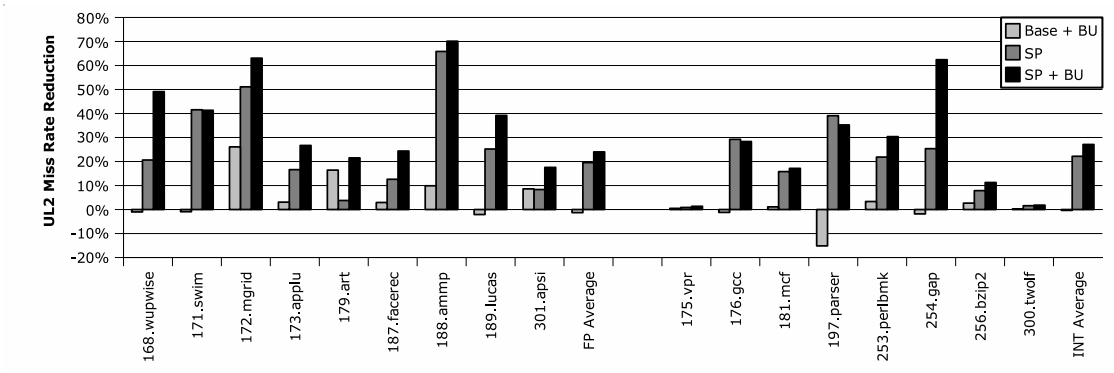



Figure 7: UL2 miss rate reduction.

improvement seems even higher than this sum. On the floating point applications, the average speed-up over the SP configuration is about **15 %**, on the integer benchmarks the average performance gain is around **3 %**.

In practice, when using the SP+BU configuration, several applications experience a speed-up higher than 20 % over the SP configuration, e.g. `wupwise` or `applu` while none of the benchmarks experienced slowdowns.

### 6.3.2 L2 cache utilization

Besides measuring IPC, we also analyzed the effects of reducing single-usage pollution in terms of miss rates on the L2 cache (Figure 7).

For base+BU, we notice that on some applications, the miss rate is decreased significantly. However the miss rate is not systematically decreased. Some applications expose only a marginal miss-rate decrease or even worse, increase the effective L2 cache miss-rate but still offers higher performance. For `parser` where the number of L2 cache misses is increased by 12 % but still remains limited around 1.5 misses per 1000 instructions, we already pointed out that the accuracy of the SU-verdict is relatively poor (only 65 %) thus leading to extra L2 cache misses.

With SP+BU, the reduction in percentages of L2 misses is in general close the sum of the reduction in percentages of L2 misses on base+SU and SP. In a few cases (`wupwise`, `facerec` and `lucas`), the benefit is higher than this sum. We found that this artifact is due to the structure of the prefetch buffer: when using only the stream buffer, some unprefetchable multiple-usage blocks are evicted from the L1 cache creating long latencies L2 misses. This creates new opportunities for initiating L2 prefetchs; prefetched blocks evict other blocks before they are used.

**Average Memory Access Time** In a memory hierarchy using two cache levels and implementing prefetching, the precise latency time of a memory access is also dependent on hit/miss on L1 cache, hit/miss on L2 cache, hit/miss on the prefetch buffer and on whether a prefetch access to the block has been initiated or not. In such a memory hierarchy, the *Average Memory Access Time* (AMAT) often better reflects performance than cache miss rates.

Figure 8 shows how the use of BU prediction and L2 bypassing reduces the AMAT. One will note the strong correlation between the IPC improvement and AMAT reduction on most applications. An AMAT reduction is observed on the base+BU configuration even when the L2 miss rate is not significantly reduced. This can be explained by the fact that the SU-blocks are statistically less critical than multiple-usage blocks: when a load misses on the L2 block B, the probability that extra loads can be executed after

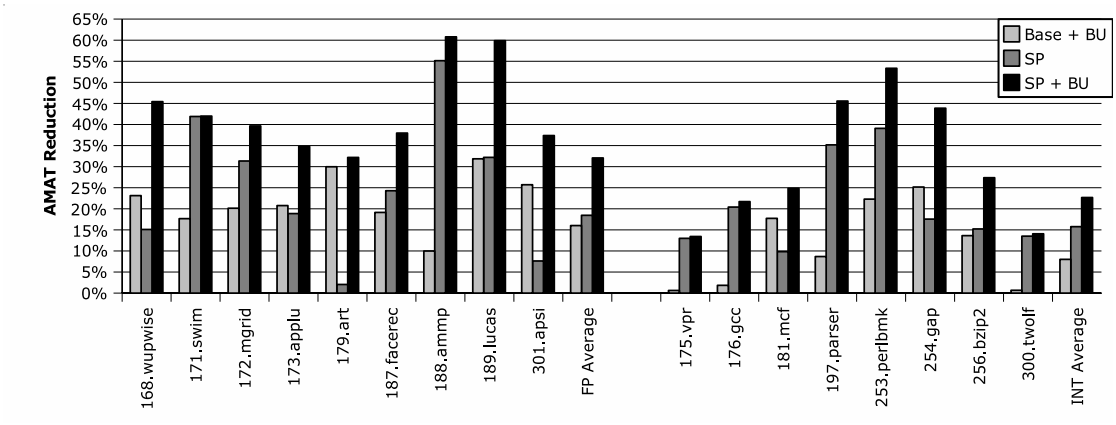



Figure 8: Average Memory Access Time reduction.

this load is higher if B is SU than if B is not SU. In other words, the average effective penalty on a SU block miss is lower than the one for a multiple usage block: by bypassing SU-blocks, we help to maintain in the L2 cache blocks for which the full penalty would be paid in case of a L2 miss.

However, for `swim`, we observe a negligible reduction of the AMAT for SP+BU configuration over the SP configuration, but still a 10% performance increase. Our measures indicated that our scheme allows a faster execution because the load/store queue stall time is reduced. In other words, the load instruction are sequenced earlier after their decode.

# 7 Conclusion

The relative main memory access time has been increasing during the last twenty years. Now, it represents several hundreds of cycles. For a long period, the trend has been to increase the total amount of caches associated with the processor. This has been enabled by the continuing advances in semi-conductor technology. However, this trend is likely to reach its limits as the processor industry seems now hungry to devote the extra transistors for designing CMPs. Therefore, it remains very important to manage the memory hierarchy as well as possible and to avoid wasting cache space.

In this paper, we have first identified the single-usage cache pollution phenomenon associated with the conventional mode of operations of the memory hierarchy: a missing block on a cache level is stored in all intermediate levels, this block is sometimes never touched back before its eviction. We have shown that on many applications, this phenomenon is particularly important on L2 caches. Significant performance and memory bandwidth may be wasted due to this non-optimal management of the memory hierarchy.

Then we have shown that, the dynamic single-usage property of a block is very correlated with the load or store instruction that causes the L2 miss on the block. We have also pointed out that this correlation allows to implement an instruction-indexed block-usage predictor for the L2 cache. This predictor delivers high coverage, i.e. recognizes most of the dynamic accesses to SU-blocks, and high accuracy on SU verdicts, i.e., when a block is predicted single-usage, it is very likely to be single-usage. Therefore, our block-usage predictor can be used to guide the decision of bypassing a block or not in the L2 cache.

We have also found that our approach is orthogonal with prefetching, in particular the popular stride prefetching [5], i.e., performance benefits are added. In addition, many SU-blocks are also blocks that can be prefetched by a stride prefetcher. Most of the logic of our single-usage predictor can be shared with the stride prefetcher. As a result, the extra hardware complexity associated with our proposal is essentially limited to the addition of a few tag bits to each line in the L2 cache.

Our experiments on a uniprocessor, indicated that, when applications exhibit a significant fraction of dynamic SU-blocks, our hardware proposal allows to increase the overall performance of the processor. On average,by augmenting an aggressive superscalar processor featuring a stride prefetcher with our L2 bypass hardware, we obtained a 15% IPC increase, and up to 44% on `lucas`, for the SPECINT, a more modest average 3% IPC increase was encountered, but 12% was observed on `mcf`. This performance improvement is achieved because our proposal general decreases the number of L2 misses, but also because it allows some critical blocks to remain in the L2 cache, thus reducing the average memory access time.

In the future, we intend to study the impact of reducing single-usage pollution in the context of multi-core architectures. In particular, we will propose to evaluate our design efficiency for both multi-programmed and parallel workloads.

# References

[1] *IBM PowerPC 970FX RISC Microprocessor User's Manual*. IBM, 2005.

[2] T.-F. Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. In *Proc. of the 21st Symp. on Computer Architecture (21st ISCA'94), Computer Architecture News*, Chicago, 1994.

[3] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. mei W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture*, 1991.

[4] C.-H. Chi and H. Dietz. Improving cache performance by selective cache bypass. In *Twenty-Second Annual Hawaii International Conference on System Sciences*, 1989.

[5] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, 1992.

[6] T. L. Johnson, D. A. Connors, M. C. Merten, and W. mei W. Hwu. Run-time cache bypassing. *IEEE Trans. Comput.*, 48(12), 1999.

[7] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, 1997.

[8] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, 1990.

[9] M. Kharbutli, Y. Solihin, and J. Lee. Eliminating conflict misses using prime number-based cache indexing. *IEEE Trans. Computers*, 54(5):573–586, 2005.

[10] E. Larson, S. Chatterjee, and T. Austin. Mase: A novel infrastructure for detailed microarchitectural modeling. In *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, 2001.

[11] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. Cache filtering techniques to reduce the negative impact of useless speculative memory references on processor performance. In *SBAC-PAD '04: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, 2004.

[12] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004.

[13] D. G. Pérez, G. Mouchard, and O. Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *MICRO*, pages 43–54. IEEE Computer Society, 2004.

[14] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-way cache: Demand based associativity via global replacement. In *ISCA*, pages 544–555. IEEE Computer Society, 2005.

[15] J. Rivers and E. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. *icpp*, 01, 1996.

[16] A. Seznec. A case for two-way skewed-associative caches. In *ISCA*, pages 169–178, 1993.

[17] A. Seznec. Analysis of the o-geometric history length branch predictor. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[18] V. Srinivasan, G. Tyson, and E. Davidson. A static filter for reducing prefetch traffic. Technical Report CSE-TR-400-99, University of Michigan, 1999.

[19] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture*, 1995.

[20] X. Zhuang and H.-H. S. Lee. A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *Proceedings of the 2003 International Conference on Parallel Processing (32th ICPP'03)*. IEEE Computer Society, Oct. 2003.