Exploiting Single-Usage for Effective Memory Management*

Thomas Piquet, Olivier Rochecouste, and André Seznec

IRISA, Campus Beaulieu, 35042 Rennes Cedex, France

Abstract. Efficient memory management is crucial when designing high performance processors. Upon a miss, the conventional operation mode of a cache hierarchy is to retrieve the missing block from lower levels and to store it into all hierarchy levels. It is however difficult to assert that storing the block into intermediate levels will be really useful. In particular, this is unnecessary if a cache block is accessed only once before getting evicted - i.e. a single-usage block. This paper is typically concerned with reducing the number of single-usage blocks. Our observations reveal that single-usage blocks are significant at runtime and especially in the lowest cache level. We show that using an address-based prediction mechanism is sufficient to identify this phenomenon. Two schemes are examined to remove pollution caused by single-usage blocks: a bypass scheme and a cache replacement policy. Our results show that leveraging single-usage pollution is beneficial to memory-intensive applications running on superscalar and multi-core architectures.

1 Introduction

Processor performance is strongly dependent on the memory hierarchy management. Access time to off-chip memory now represents several hundreds of cycles. In order to hide such huge latencies, modern processors feature a complete memory hierarchy composed of multiple cache levels with variable latencies. In addition, hardware prefetch mechanisms [1] are also often used to minimize the impact of main memory access time.

On a cache miss, the conventional memory hierarchy propagates the missing block from the lowest level in the memory hierarchy to the highest 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 retains different memory 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 into account that blocks have very disparate usages across applications. In particular, in some cases, a block stored in

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

the cache after a miss may not be accessed again before it is evicted. We call such a block, a *single-usage* block or a SU-block. Storing a SU-block in the cache may cause severe performance degradation as it could evict another block that could potentially be more useful. We refer to this phenomenon - storing SU-blocks in a cache - as *single-usage pollution* or SU-pollution.

Our first contribution in this paper is the characterization and analysis of the SU-pollution phenomenon. For a 2-level cache hierarchy, we show that most applications only exhibit a limited amount of SU-pollution in the L1 data cache, while some applications exhibit a high SU-pollution rate in lowest cache level. Our analysis also reveals that the single-usage property of a block is closely related to the memory instruction that triggers the L2 miss on this block.

Our second contribution is the proposal of a hardware mechanism for predicting single-usage pollution. Two schemes are presented to exploit SU-pollution: (1) a bypass scheme that prevents SU-blocks from entering the cache and (2) a SU-based cache replacement policy. Experiments show that our proposal is beneficial to both superscalar and multi-core architectures where the memory subsystem is a bottleneck.

The remainder of this paper is organized as follows. Section 2 quantifies single-usage pollution. In Section 3, we propose a single-usage prediction scheme and two techniques to exploit single-usage blocks. Section 4 presents our experimental results. Section 5 discusses the related work. Section 6 concludes this study.

2 Characterizing Single-Usage Pollution

We quantified the number of SU-blocks that are accessed at runtime for a subset of the SPEC2000 applications. Our data has been collected on a 4-way superscalar architecture featuring a 2-level cache hierarchy (32KB 4-way L1 data cache and 512KB 4-way L2 cache). A complete description of our baseline configuration is available in Section 4.1.

2.1 Quantifying SU-Pollution Within L1 and L2 Caches

We measured the number of dynamic accesses to SU-blocks within both the L1 and L2 caches. For a given cache level, *a cache block is defined as single-usage if it is accessed only once before getting evicted from this level.* Figure 1 reports the fraction of memory accesses that are single-usage at execution. We notice that SU-pollution is quite negligible in the highest cache level as only 6%, on average, of the dynamically accessed blocks are single-usage. The high usage behavior of cache blocks, mainly stems from the fact that data contained in L1 data cache exhibits high spatial and temporal localities. In our context, this means that attempting to reduce SU-pollution in the L1 cache would only have a small impact on overall performance. In contrast, the SU-pollution amount is much more significant in the lowest cache level. On average, 33% of memory accesses in the L2 cache are single-usage. We can even observe for some applications (*wupwise, swim, mgrid, applu, art, ammp*), mostly based on memory-intensive scientific

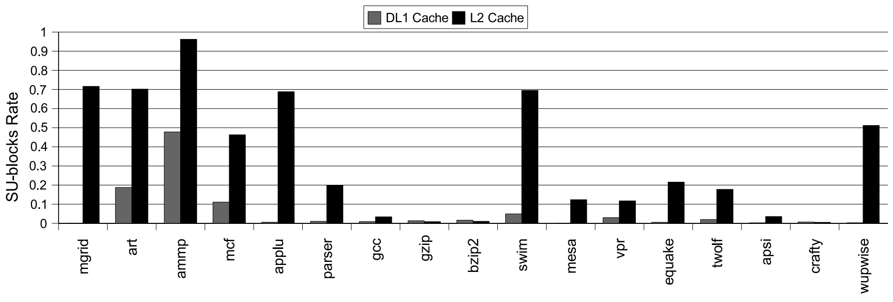

Fig. 1. Single-usage pollution within L1 and L2 caches

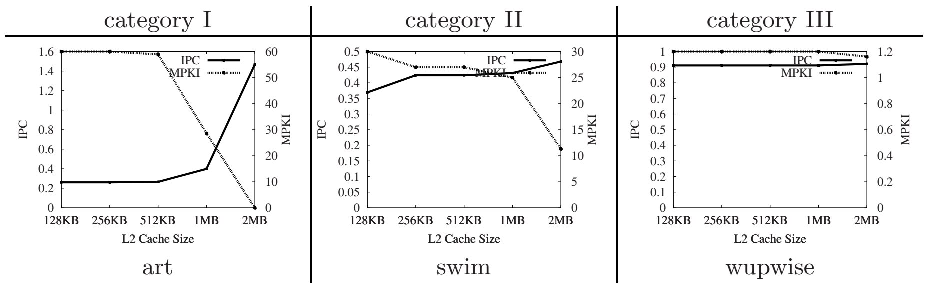

Fig. 2. Applications sensitivity to varying L2 cache size in terms of IPC and MPKI

kernels, that SU-blocks are quite prominent at runtime. Other applications such as *gzip, gcc, crafty, bzip2* depict a small SU-pollution level and are unlikely to benefit from our scheme.

2.2 Categorizing SPEC2000 Applications

Albeit some applications exhibit a high SU-pollution rate, minimizing this quantity would not necessarily translate into speed improvement, especially if the program performance is not dependent on the memory hierarchy behavior. We studied the applications sensitivity by varying the L2 cache size (from 128KB up to 2MB), using IPC and MPKI (miss per kilo-instruction) as metrics. Due to space constrains, we only report results for a few programs.

Figure 2 classifies the SPEC2000 applications into three distinct categories. The first category encompasses benchmarks in which performance and miss rate are very sensitive to increasing the L2 cache size. In our context, these applications are very likely to benefit from a scheme that reduces SU-pollution. For *art*, increasing L2 cache size from 512KB to 1MB has a significant impact on IPC and MPKI. In addition, since *art* exhibits a considerable SU-pollution amount, minimizing this quantity would also allow a substantial increase in the available cache space; and hence, in a potential performance improvement. The second category comprises memory-intensive workloads that do not benefit from resizing the L2 cache from a performance viewpoint. Nonetheless, we observe that doubling the L2 cache size does still lead to a noticeable reduction in terms of miss rate. This would therefore allow to reduce the main memory bandwidth usage, which could be used to trigger prefetch requests instead. The last category gathers applications in which performance and miss-rate are not dependent on cache size. For these applications, attempting to reduce SU-pollution will have only a marginal impact on performance, or worse, in case of a SU misprediction, this could even result a performance loss.

3 Predicting and Exploiting Single-Usage Blocks

Minimizing the pollution due to SU-blocks can help improve the whole memory hierarchy behavior as well as the processor performance. In this section, we first show that the single-usage property of a dynamic L2 cache block is closely related to the instruction that triggers the L2 cache miss. This observation makes a PC based block-usage prediction mechanism viable. We also describe how a stride prefetcher could be adapted at a minimum extra hardware cost for predicting cache blocks usage. To decrease SU-pollution, we propose two distinct schemes: (1) a bypass solution and (2) a SU-based cache replacement policy.

3.1 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. Johnson et al. used this approach in [2]. However, we show below that a cache block usage property is tied to the program instruction that triggers the memory access.

Quantifying Single-Usage I-Sequences. We refer to the sequence of L2 cache accesses initiated by a single program instruction as *an I-sequence*. Let us also define a single-usage I-sequence as a I-sequence for which the amount of SU-blocks exceeds 95%. Note that the SU I-sequences property depends on the memory hierarchy configuration. On average, we found that over 90% of SU-blocks are referenced by SU I-sequences across SPEC2000 benchmarks. Hence, a mechanism capable of detecing SU I-sequences at runtime could help decrease SU-pollution.

3.2 Hardware Support to Predict Block Usage

The block-usage (BU) predictor (see Figure 3), 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 re-accessed after being stored into the L2 cache, and the instruction address tag (or IA tag) that records the address of the instruction

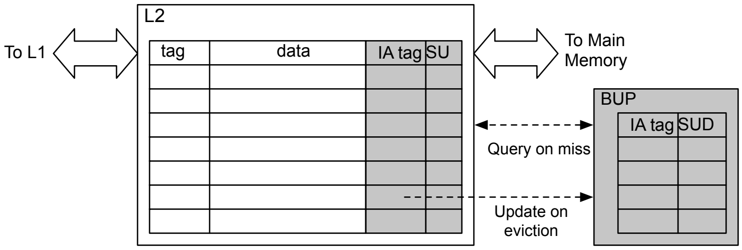

Fig. 3. Block-usage predictor (BUP)

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 that incurred 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 counter state, else a non-SU verdict is returned.

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-sequence 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 is single usage and 2) the block is predicted as single usage but might have been accessed several times if it was stored into L2 cache - this is referred to as a SU misprediction. Due to the potential performance loss induced by a SU misprediction, our BU predictor favors accuracy over coverage. This is done through delivering SU verdicts only upon a saturated SUD counter state and by resetting counters on non-SU updates.

Adapting a Stride Prefetcher for Block-Usage Prediction. One can easily extend a stride prefetcher [1] for block-usage prediction as this mechanism also uses the instruction address to initiate prefetch requests. This would enable to mitigate the hardware overhead due to the BU predictor. To do so, each entry in the stride prefetcher table has to be augmented with a SUD counter. The main overhead induced by the BU predictor is the additional tags on the L2 cache. In our experiments, each L2 cache line is augmented with a 1-bit SU tag and a 13-bit partial 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.

3.3 Reducing SU-Pollution

We propose two distinct techniques to exploit SU-pollution reduction in the L2 cache: (1) a bypass scheme and (2) a SU-based cache replacement policy.

Bypass Scheme. In order to prevent SU-pollution, we suggest to directly forward missing SU-blocks - identified by means of the BU predictor - from memory to the L1 data cache; thus bypassing the L2 cache to enable more useful data to remain cached. Note that non-SU blocks are still processed in a standard way. Performing bypassing could however have a detrimental effect on the predictor accuracy. Once an I-sequence is marked as single-usage, its corresponding SUD counter could remain saturated forever. To overcome this issue, we propose to re-inject a SU-block into the L2 cache once in a while. This allows to update the SUD counter. If the I-sequence behavior changes, the SUD counter will be reset. The decision of re-injecting a SU-block is taken using a low probability.

SU-Based Replacement Policy. Another way to reduce SU-pollution is to use the BU predictor for a cache replacement purpose. Let us suppose a set-associative cache featuring a LRU replacement policy. Our proposal is to augment the LRU algorithm with block usage information. Upon selecting a block for eviction, our technique favors the replacement of least-recently-used blocks marked as single-usage instead of solely using the recency information. If there are no SU-block in the current set, the LRU block is selected. The architectural support needed for this scheme consists of extending each L2 cache block with a single bit that reflects whether or not this block is single-usage. This prediction bit is updated each time a cache block is loaded from memory. As for the bypass scheme, however, SUD counters could remain saturated. To avoid this scenario, we take an arbitrary decision using a low probability to decide if we should select the LRU-block as a victim instead of the SU-block.

4 Evaluation

This sections evaluates the performance of the BU predictor in terms of accuracy and coverage. It also examines the impact on performance, miss-rate and memory traffic induced by the SU-based cache replacement policy and the bypass scheme.

4.1 Experimental Setup

Our experiments were performed on SESC, an execution-driven simulator developed by [3]. Our baseline processor is a 4-way out-of-order superscalar architecture. Table 1 summarizes the configuration we used as a reference. Our memory subsystem models a 512-entry stride prefetcher [1] that is coupled with a 32-entry prefetch buffer [4, 5] to filter pollution related to aggressive prefetching.

Benchmarks. We evaluate our proposal on a subset of SPEC2000 benchmarks that run on SESC: *wupwise, swim, mgrid, applu, mesa, art, equake, ammp, apsi,*

Table 1. Simulated machine parameters

Parameter	Configuration
Decode / Issue / width	4
Retire width	5
ROB size	36 Issue + 32 entries
LSQ size	20 Issue + 32 entries
Branch predictor	O-GEHL [6], 64-Kbit, 6-cycle mispred. penalty
L1 inst.	64kB, direct-map, 128B/block, LRU, 1-cycle
L1 data	32kB, 4-way, 128B/block, LRU, 1-cycle
L2 unified	512kB, 4-way, 128B/block, LRU, 11-cycle
Main Memory latency	500-cycle

Table 2. BU predictor coverage and accuracy (512-entry table + 3-bit SUD counters)

	coverage	accuracy	SU-rate	#cache accesses (*M)		coverage	accuracy	SU-rate	#cache accesses (*M)
mgrid	96.4%	99.53%	71.64%	4.96	swim	87.74%	99.64%	69.52%	31.3
art	83.72%	99.8%	70.18%	69.91	mesa	88.23%	99.98%	12.35%	1.54
ammp	99.63%	99.98%	96.23%	86.76	vpr	1.39%	80.2%	11.77%	19.45
mcf	71.71%	98.16%	46.31%	88.41	equake	59.16%	99.77%	21.58%	4.57
applu	98.01%	99.84%	68.84%	6.34	twolf	0.81%	74.08%	17.79%	25.43
parser	4.81%	81.65%	19.87%	9.45	apsi	67.19%	98.06%	3.6%	2.58
gcc	49.04%	97.92%	3.42%	13.47	crafty	0.02%	91.67%	0.54%	11.78
gzip	67.05%	97.2%	0.87%	7.85	wupwise	97.65%	99.74%	51.19%	2.22
bzip2	41.61%	89.26%	1.05%	9.75					

gzip, vpr, gcc, mcf, crafty, parser, bzip2, twolf. All applications were compiled for the MIPS ISA with the `-O3` optimization flag enabled. We used the reference data as an input. The first billion instructions were skipped and the next billion instructions were simulated.

4.2 Block-Usage Predictor Accuracy and Coverage

We define the BU predictor coverage as the fraction of the number of SU-blocks that are correctly predicted. The BU predictor accuracy as the fraction of SU verdicts that are correct. Table 2 reports the coverage and accuracy of a 512-entry BU predictor table featuring 3-bit SUD counters. Overall, the BU predictor provides high accuracy on most benchmarks. The rationale is that we deliver SU verdicts exclusively on a counter saturated state while resetting the count value on non-SU verdicts. Although this slightly impairs the predictor coverage, we observe that we still identify a large fraction of SU-blocks for memory-intensive applications. For other applications such as *crafty, vpr, twolf*, our predictor is quite inefficient. This is due to the fact that these applications only exhibit a small SU-pollution rate as mentioned in Section 2.1.

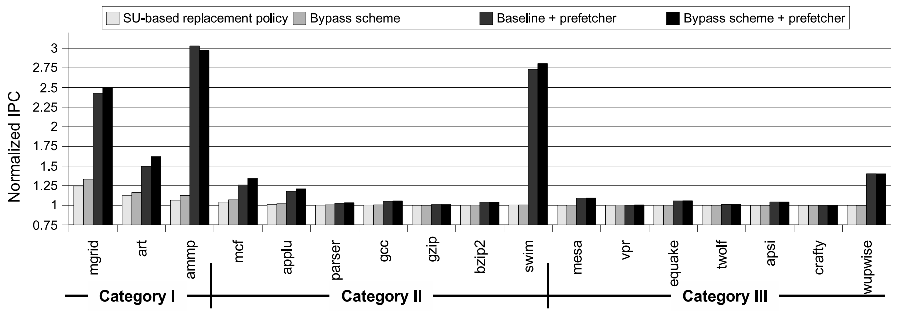

Fig. 4. IPC normalized to baseline for our different schemes

Varying BU Predictor Parameters. We varied the main BU predictor parameters (number of predictor entries and SUD counter width) to study their influence on coverage and accuracy. Increasing the number of entries from 128 to 1024 has a positive, but small impact on coverage. Increasing the SUD counter width decreases the coverage of the BU predictor but increases the SU-verdict accuracy. 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.

4.3 Impact on Performance, Miss-Rate and Memory Traffic

Figure 4, Figure 5 and Figure 6 compare our cache management policies using three metrics, namely the IPC, the L2 cache miss rate (in MPKI) and the bus traffic (number of accesses) between L2 cache and main memory. These results are normalized to our baseline architecture described in Table 1. The first bar corresponds to our SU-based replacement policy described in Section 3.3. The second bar represents the bypass scheme (see Section 3.3). The following bar is the baseline architecture enhanced with a stride prefetcher. The last bar corresponds to the stride prefetching scheme adapted for block-usage prediction.

Figure 4 points out that our proposal performs well with workloads from the first category (see Section 2.2) whereas applications from other categories show little or no performance gains. *mgrid* performs well on both schemes by achieving a speed-up close to 30% along with a noticeable decrease in the L2 cache miss-rate. This is consistent with our analysis as we observed that *mgrid* performance is very sensitive to adapting the L2 cache size - especially from 512KB to 1MB.

When a performance gain is observed, the bypass scheme usually performs better than the SU-based cache replacement policy. This is somewhat coherent as bypassing SU-blocks allows existing multi-usage data to remain cached in L2. In contrast, with the SU-based replacement policy, a few SU-blocks can still reside in L2 cache; hence the lower performance gain.

For most applications, using a stride prefetcher for block-usage prediction is beneficial to performance and miss rate. Due to a reduced memory traffic - see

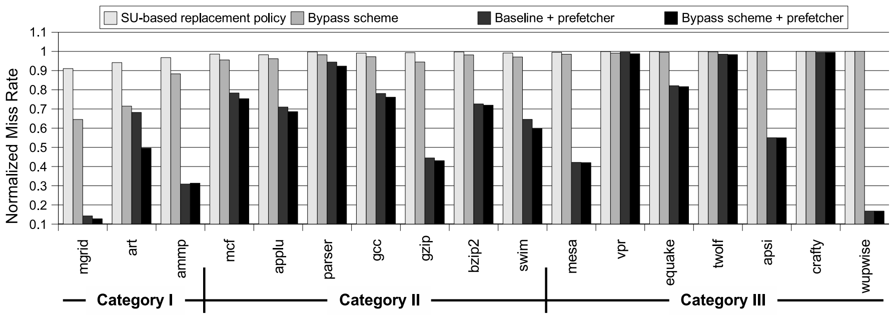

Fig. 5. Normalized L2 cache miss-rate

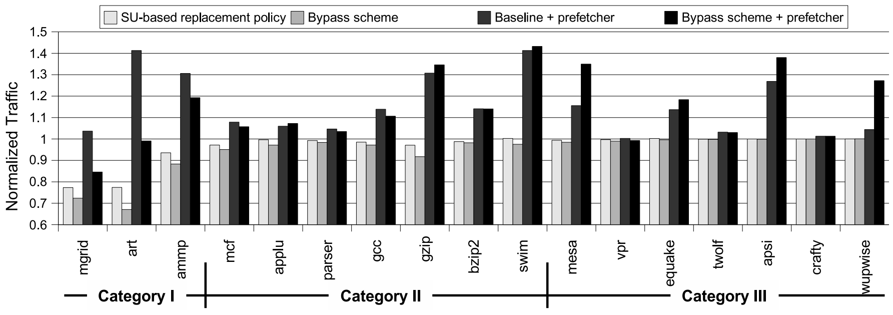

Fig. 6. Normalized memory traffic between L2 and main memory

Figure 6 - the number of prefetching opportunities is accordingly increased; hence an extra performance gain as compared to a basic bypass scheme. On some applications such as *gzip*, no performance gain is observed as this program has a small miss rate. Note that the performance improvement obtained with the prefetcher-based BU predictor is not as significant as that of a basic BU predictor scheme - e.g. see *mgrid*. This stems from the fact that the usage of prefetching overrides part of the benefits achieved by our scheme.

Our proposal slightly degrades performance of *ammp* when used with a stride prefetcher - from a 0.0732 IPC to 0.0718. This is due to the management of the memory bus. Prefetches are initiated only when the bus is free. Bypassing the L2 cache on write-backs of SU-block tends to create a burst of traffic that could prevent prefetches. When data are first stored in the L2 cache, the write-back traffic is smoothened, creating new opportunities for triggering prefetch requests.

4.4 Exploiting SU-Pollution Reduction in Multi-core Systems

Managing the memory hierarchy is a crucial issue in multi-core architectures where processing cores often share the lowest cache level. We examined the potential gains that could be achieved by our bypass scheme in this context. To do

Table 3. Weighted IPC, L2 cache miss-rate and SU-pollution for a dual-core system

	2-core - 1MB L2			2-core - 2MB L2			2-core - 1MB L2 + bypass		
	W_{IPC}	Miss Rate	SU rate	W_{IPC}	Miss Rate	SU rate	W_{IPC}	Miss Rate	SU rate
wupwise	1.00	0.50	98.52	1.00	0.50	97.20	1.00	0.50	4.14
mgrid	0.87	3.48	44.99	1.00	2.96	35.12	1.00	2.98	3.51
applu	1.00	4.47	90.90	1.00	4.47	90.83	1.00	4.47	3.05
mcf	0.98	51.39	68.42	1.08	45.95	57.51	1.21	43.17	18.78
mgrid	0.71	4.54	72.59	1.00	2.97	35.38	0.94	3.53	4.59
ammp	0.57	9.34	4.37	1.00	0.19	0.07	0.93	1.14	0.17
art	0.94	35.23	44.62	2.73	0.26	0.11	1.09	23.81	12.35
gzip	0.79	1.68	11.08	0.99	0.21	1.65	0.81	1.55	3.93

so, we modeled with SESC a dual-core system that features private 32KB 4-way L1 data caches and a shared 4-way 1MB L2 cache. A 1k-entry BU predictor table is considered. We mixed together applications from distinct categories (see Section 2.2) to study the performance impact on our scheme. As a performance guide, we use the weighted speed-up metric [7, 8]. For each benchmark, we simulated 250M instructions. If a benchmark completes 250M instructions before the other, we keep on executing the finished benchmark till the second benchmark finishes its processing.

Results. Table 3 reports the weighted IPC, the L2 cache miss-rate and the associated SU-pollution rate for different memory configurations of the baseline dual-core system. For each programs mix, we report the contribution of individual programs for the considered metrics. For instance, running *wupwise-mgrid* on the baseline CMP shows that *wupwise* does not suffer from cache sharing ($W_{IPC} = 1$) while *mgrid* ($W_{IPC} < 1$) does. Table 3 shows that executing our mixed applications with a larger L2 cache often improves the considered metrics. Overall, our bypass scheme applied to a multi-core architecture provides noticeable performance gains. It does even outperform a CMP system featuring a twice as large L2 cache when executing *applu-mcf*. While *applu* by itself does not benefit from reducing SU-pollution, it does however makes room for *mcf*, thus allowing substantial performance gain on this latter application. The same phenomenon occurs with *mgrid/ammp*. Reducing *mgrid* SU-pollution essentially reduces *ammp* miss rate.

5 Related Work

Tyson et al. [9] observed that, on many applications only a few load instructions are responsible for the majority of data cache misses. They proposed a scheme to decide whether or not a load instruction should allocate data in the L1 cache. The authors suggest using PC-indexed counters that are incremented on a miss and decremented on a hit. In practice, a load instruction is classified as a "to be bypassed" if in general it is the first to touch a memory block and if further instances of the same load do not touch back the same block in the near future. This scheme is able to capture instructions exhibiting no spatial locality on the L1 cache, such as loads exhibiting a stride longer than a cache block. However it is not able to capture future reuse of the cache block by other loads or writes. This may sometimes lead to dramatically poor behavior, particularly on optimized code. For example, on a streaming application, unrolling a loop with an unrolling factor larger then the cache line size may push the hardware to classify the first access to each data in a cache block as a "bypass access".

Dybdahl et al. studied block bypassing in the last cache level in [10]. They extended the dynamic scheme for the L1 cache proposed by Tyson [9] to the lowest cache level. They noticed that this extension sometimes leads to a severe performance loss. They proposed a new hardware scheme to address this issue. The result is mitigated: performance losses are reduced on some applications, performance benefits are also reduced on other applications. The hardware cost of their scheme is relatively high, since each block in the last-level cache is augmented with voluminous information (shadow address tag, instruction address, status). Moreover the management algorithm is quite complex.

Chi et al. [11] 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 main limitation of this software solution is that it does not take into account the spatial locality within a cache block.

Rivers and Davidson [12] proposed a hardware mechanism to capture the temporality of a data block. Data blocks are classified as temporal or non temporal (NT). A block is classified as NT if none of its words is re-referenced before its eviction. A NT bit is added to each block in the first and second levels of the cache. The main memory does not have the NT bit, therefore once a block is evicted from the L2 cache, the information is lost. In contrast to this proposal, we associate the single-usage property to memory access instructions rather than to cache blocks, and we address the L2 cache.

Wong and Baer [13] described a cache replacement policy enhanced with temporal locality information to guide block replacement. Instead of systematically evicting LRU blocks, their scheme favors replacing non-temporal blocks instead. The temporal information is obtained through profiling or by means of a hardware predictor.

6 Conclusion

This paper proposes to exploit reduction in single-usage cache pollution for a better memory hierarchy management. We observed that the single-usage property of a cache block is very tied to the load/store instruction that causes a cache miss (on this block). Hence, we suggest using a PC-based hardware predictor to uncover SU-blocks at runtime. Our experiments show that our predictor provides high coverage and accuracy on most programs. We evaluate two schemes to reduce SU-pollution: (1) a bypass technique and (2) a SU-based cache replacement policy. Our results point out that using either technique is beneficial to a superscalar architecture. Extra gain is further observed when adapting a stride prefetcher for block-usage prediction - while mitigating the storage overhead due to our predictor. Our proposal is also evaluated in a multi-core environment using multi-programmed workloads. Exploring the benefits on multi-threaded programs is part of our future work.

References

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

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

[3] Renau, J., Fraguela, B., Tuck, J., Liu, W., Prvulovic, M., Ceze, L., Sarangi, S., Sack, P., Strauss, K., Montesinos, P.: SESC simulator (2005), `http://sesc.sourceforge.net`

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

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

[6] Seznec, A.: Analysis of the o-geometric history length branch predictor. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture (2005)

[7] Snavely, A., Tullsen, D.M., Voelker, G.: Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In: Proceedings of the 2002 international conference on Measurement and modeling of computer systems (2002)

[8] Hsu, L.R., Reinhardt, S.K., Iyer, R., Makineni, S.: Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In: Proceedings of the 15th international conference on Parallel architectures and compilation techniques (2006)

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

[10] Dybdahl, H., Stenström, P.: Enhancing last-level cache performance by block bypassing and early miss determination. In: Asia-Pacific Computer Systems Architecture Conference (2006)

[11] Chi, C. H., Dietz, H.: Improving cache performance by selective cache bypass. In: 22nd Hawaii International Conference on System Sciences (1989)

[12] Rivers, J., Davidson, E.: Reducing conflicts in direct-mapped caches with a temporality-based design. icpp 01 (1996)

[13] Wong, W.A., Baer, J.L.: Modified lru policies for improving second-level cache behavior. In: HPCA (2000)