A hybrid timing-address oriented LSQ filtering for an x86 architecture

Abstract

In the last few years, many researchers have focused their efforts on the field of low-power processor design. Several jobs in this area have dealt with the logic that enforces correct memory-based dependences –the Load-Store Queue–, a pretty energy-consuming structure since many accesses are performed in an associative fashion. Among these proposals, some of them manage to reduce this resource's energy consumption by avoiding unnecessary lookups. In this context, we introduce a straightforward filtering mechanism, which results in a more energy-efficient design than past techniques, using less and simpler hardware. Besides, both the new scheme and some previous approaches are tested in the widespread x86 architecture. This microarchitectural model provides new opportunities for extra types of filtering, which lead to higher energy savings. On average, our proposal filters up to 75% of the associative accesses to the LQ, 56% to the SQ, and 42% to the Dependence Predictor with a reduced amount of hardware –less than 100 bytes–. According to our energy model, this means a dynamic energy saving of more than 39% over a conventional LSQ.

Key words: LSQ, Age Registers, Age-based Filtering, Bloom filters, Energy-efficiency

1. Introduction

Until some years ago, the only concern of a microprocessor researcher was to increase performance, and for this reason, the management of all the different structures in the processor was just oriented to optimize this factor. However, a lot of research in the field of low-power processor design has been developed recently, due to the well-known problems that are arising associated with the increasing energy/power consumption. The general trend is to focus on a processor's component and to propose a new management scheme that, maintaining performance –or slightly decreasing it– leads to a lower energy/power consumption.

One such part is the logic that enforces correct memory-based dependences, commonly referred to as the load-store queue (LSQ), and typically implemented as two separated queues: the load queue (LQ) and the store queue (SQ). Conventional implementations of this resource contain complete addresses, with the corresponding entries allocated in program order. To enable early execution of loads without compromising program correctness, memory instructions are tracked by the two queues and associative searches are used to find the correct producer or to detect dependence violations.

These search operations are a major concern for the scalability and consumption of the structure. As such, a range of implementations that reduce the number of associative searches have been explored recently [1] [2] [3] [4]. It can be argue that this research problem is not a major concern now due to the clear shift towards multi-core architectures made by the industry. However homogeneous multi-manycore architectures will only provide substantial benefits for scalable applications/workloads and some researchers have recently highlighted that future design will benefit

January 22, 2010

from asymmetric architectures that combine simple and power-efficient cores with a few complex and power-hungry cores [5]. The local inefficiencies of a complex core can translate into global performance/per-watt improvements since a complex core could accelerate the serial phases of applications when the power-efficient cores are idle. This way, a single chip will be able to provide good scalability for parallel applications as well as ensure high serial performance. In summary, as promoted in [6], researchers should still investigate methods of improving sequential performance despite we have entered into the multicore era.

In this paper, we make the following contributions:

1. We introduce a new simple hybrid mechanism which, using less and simpler hardware than previous proposals, manages to increase the filtering capability to the LSQ. This technique, as previous ones based on filtering, has no impact on performance.
2. We test the mechanisms from [1], [2] and our new scheme in a different microarchitectural model –an x86 architecture–, that results from the merging of different features of an Intel Pentium 4 [7] [8], an AMD K8 and an Intel Core 2 architecture [9]. Clearly, this model results more appealing than previous ones, since most current domestic computers use an Intel or an AMD processor.
3. Due to the different LSQ management employed in this microarchitectural model compared with that of [1] and [2], new types of filtering arise, providing the opportunity for extra energy savings.
4. We provide a complete energy model for the LSQ hardware and include it on the PTLSim simulator [10]. We evaluate the LSQ energy savings achieved in [1], [2] and in our hybrid mechanism.
5. We instrument the SPEC CPU2006 benchmark suite for using it with the PTLSim simulator.

The rest of the paper is organized as follows. Section 2 summaries the conventional design of the LQ and SQ in an x86 architecture. Section 3 recaps related work. Section 4 presents our new filtering mechanisms. Section 5 details our experimental environment, the SPEC'06 instrumentation, and the proposed energy model. Section 6 outlines experimental results and analyses. Finally, Section 7 concludes.

2. Conventional LSQ Design in an x86 Architecture

In modern microprocessors the load-store queue is the structure responsible for keeping memory operations, enforcing memory dependences and detecting memory order violations through associative searches. In this section, we detail the LSQ management [11] that we employ [1] in this paper.

2.1. Load instructions

Most modern processor implementations, looking for an increase of performance, allow aggressive issue of loads, *e.g.*, they issue as soon as their physical addresses are computed, even if there exist previous dependent or unresolved stores. Furthermore, an associative search to the SQ is performed at load issue, checking if previous matching stores exist. The following situations can occur as result

[1] Compared to RISC architectures, the x86 architecture is infamous for its relatively widespread use of unaligned memory operations. This concern must be efficiently handled by any implementation.

of this search: (1) If a prior matching store is found and its data is ready, the load gets it through a *store to load forwarding* [2]; (2) If a previous matching store is found, but its data is not available, or if a prior unresolved store is detected and it is predicted to overlap the load (see section 2.2 for more details about this predictor), a dependency is created on the store, and the load is re-issued some cycles later when the store is resolved. When the load is re-issued, the SQ is scanned again to make sure no intervening stores arrived in the meantime; (3) Finally, if a previous unresolved store is found, but it is not predicted to overlap, the load ignores it and the search continues.

2.2. Dependence Predictor

This structure, called LSAP (Load Store Alias Predictor), consists on a small associative table –accessed by every load PC– that predicts weather the current load will alias with a previous store. If a load finds a previous unresolved store during the SQ search, and it is predicted to alias, it is not allowed to issue until the store gets resolved.

2.3. Store instructions

As loads can execute despite the presence of prior unresolved stores, the data returned by such speculative loads may be incorrect (*premature loads*), so the processor must detect and repair this mis-speculation. Thus, stores search the load queue in an associative fashion to find any subsequent matching load which has already issued. If a hit occurs (*aliasing*), the intructions dependent on the store are re-executed, with the corresponding performance degradation.

In an x86 architecture, due to the use of unaligned memory operations, the data needed by a load in a store to load forwarding may come from more than one store instruction. To handle this situation efficiently, it is desirable that loads which may need to forward data from a store reference exactly one store queue entry, rather than having to merge data from multiple small prior stores to cover the entire byte range being loaded. For this purpose, stores merge their data with that of previous matching ones. In order to check if this mixing is feasible store instructions must check the SQ too.

During this associative access, the following situations can occur: (1) If a previous matching or a previous unresolved store is found, a dependency is created, and the current store is re-issued when everything it needs gets ready (its own data and the address and data of the store it depends on); (2) If no matching or unresolved store is detected, the store waits for its own data, and then it is re-issued.

When the store is re-issued, it repeats the scan of the load and store queue –to catch any instruction that may have issued between the first and second phase– and performs the data merging if neccessary (*store to store forwarding*).

The first and second phases may be combined into a single issue without re-issue if both the address and data operands of the store are all ready at the same time and the prior store (if any) the current store inherits from has already successfully issued.

---

[2]Since loads and stores may have unaligned addresses, the store to load forwarding process can be complicated. This way, some bytes in the region accessed by the load could come from the data cache while other bytes may be forwarded from a prior store. To solve this situation, a temporary buffer to extract the correct data is required.

3. Background

The LSQ exhibits two main problems: 1) its logic is complex as it involves associative comparison of wide operands (addresses), which implies a high energy consumption, and 2) the scaling of the structure increases its access latency, which makes it hard to integrate it in high-frequency designs. Based on the observed behavior of memory instructions (memory order violations and forwardings are infrequent), many researches have proposed filtering techniques to reduce the number of associative searches.

Sethumadhavan et al. [2] propose a type of filtering scheme named *search filtering*, which uses hashing to reduce the number of lookups to the LSQ. For this purpose two Bloom Filters [12] are employed, and based only in address information they are able to significantly reduce the associative searches needed while maintaining the program semantics.

Castro et al. [1] introduced two filtering mechanisms for the LSQ that avoid many of the associative searches that a conventional processor performs unconditionally. The design is based on two age registers: one that filters lookups to the LQ, and another that carries out the same operation over the SQ. Upon execution, just based on straightforward age comparisons between memory instructions and the corresponding register, the mechanism is able to deliver high filtering efficiency with a negligible hardware cost.

In section 3 we will review again these two proposals in more detail.

Park et al. [3] propose to extend the *store set predictor* [13] to predict the matches between loads and stores. They called their scheme *store-load pair predictor*. A load will search the store queue only when the store-load pair predictor says that there is a potentially-dependent store in the queue and tells the load to obtain its value from the SQ. To detect potential mispredictions, stores search the load queue for matching loads at commit.

Sha et al. [4] propose the improvement of the SQ scalability by implementing store-load forwarding using speculative indexed access –*store queue index prediction* (SQIP)–, rather than associative search. This technique uses prediction to identify the single SQ entry from which each dynamic load is most likely to forward. A forwarding mis-prediction, detected by pre-commit filtered load re-execution, results in a pipeline flush.

Sethumadhavan et al. [14] propose a new LSQ implementation to reduce area and search bandwith by allocating entries when instructions are issued rather than when they are dispatched. Thus the instructions life time inside the structure decreases and therefore the requiered LSQ size and the searches power consumption reduce too.

Cain and Lipasti [15] propose to solve the associative load queue scalability problem by completely eliminating the associative load queue. Instead, data dependences and memory consistency constraints are enforced by simply re-executing load instructions in program order prior to retirement. Authors propose a mechanism named *value-based replay*, that performs re-execution of load operations in program order prior to commit. If both loads read the same value, then the load correctly resolved its memory dependences; otherwise a violation occured either due to an incorrect reordering with respect a prior store or a potential violation of the memory consistency model. Instructions that have already consumed the load's incorrect value are squashed.

Roth [16] proposes a new mechanism to significantly reduce the re-execution requirements in those schemes similar to [15] that allow a first speculative load access and perform another access prior to commit to compare values obtained and verify the correctness of the first execution. The author introduces the *store vulnerability window* (SVW), a filter that reduces the number of loads that must re-execute to support a given load optimization. SVW assigns each dynamic store a

monotonically increasing sequence number. This number is used to associate with each dynamic load a dynamic window of stores to which that load is made vulnerable (potentially dependent).

Subramaniam et al. [17] propose a mechanism in which for each load the SQ index of a sourcing store is predicted. The reason why it works is that a store that forwards data to a load usually forwards to the same load each time. To implement this technique, that results in the complete elimination of the SQ, any store that forwards data to a load will use a predicted LQ index to directly write the value to the LQ entry without any associative logic. Any mispredictions/misforwardings are detected by a low-overhead pre-commit re-execution mechanism.

4. Hybrid Filtering Mechanism

We present in this work a full LSQ filtering HW, built upon proposals [1] and [2]. In both papers, the main idea is to add simple HW capable of ruling out some memory ordering violations and store to load forwardings. For that purpose, in the first scheme timing information was mainly used, while the second one employed just address information of memory instructions. We combine both in a new hybrid design that provides more filtering capability. Besides, in the new architecture, new types of filtering arise.

4.1. LQ Filtering

Regarding stores searching the LQ looking for premature loads, we can take advantage of a register that contains information about loads' age (timing information). More specifically, we add a register that records the age of the youngest load that has issued. This register is referred as YLA (Youngest issued Load Age) [18] [1] [19]. A single YLA only takes into account timing information. As it is desirable to combine both timing and address information, we can just use a handful of registers, each of them covering a different range of addresses (**multi-YLA scheme**).

The operation of such implementation is the following: when a store instruction resolves its address, instead of immediately perform an associative search to the LQ, it first compares its own age with the YLA that corresponds to the store's address (Fig. 1, first stage). If the store is younger, we can guarantee that regarding the current store no premature loads exist, and more important, the corresponding lookup can be avoided. Otherwise, the temporal ordering is such that a potential memory dependence violation exists, and thus the search is conservatively carried out (Fig. 1, second stage). In [1] more details about the YLA operation are given, the age management is explained, and the complete YLA updating process is described.

As we will show in the experimental results section, the multi-YLA by itself provides really good results in terms of filtering capability, since it employs both timing and address information. Therefore, combining this scheme with the Bloom Filter from [2] would report no significant improvements.

4.2. SQ and Predictor Filtering

Before explaining our technique, let us summarize the main ideas we borrow from [1] and [2], and how we apply them to the new architecture. In [1], authors propose a register similar to YLA, called OFS (Oldest in Flight Store), which records the age of the oldest store in the processor. When a load instruction resolves its address, instead of performing an unconditional associative access to the store queue, it first consults the OFS. If the load is older than the value recorded in the register, we can assure that a store to load forwarding is not required for the current load, so the corresponding lookup is avoided. Otherwise, the associative search to the SQ is carried out

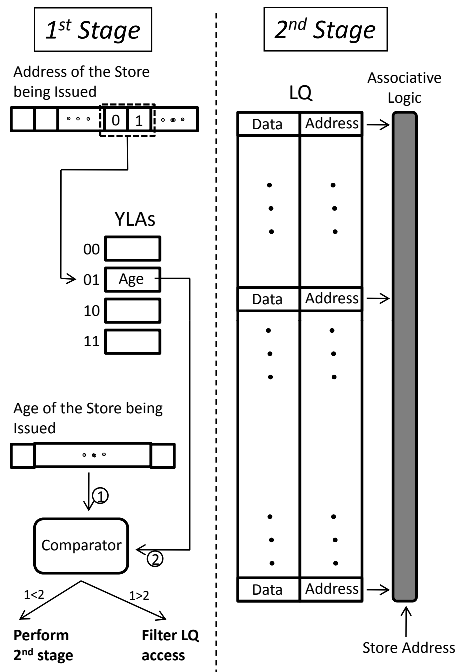


Figure 1: New LQ implementation.

in order to compare address information. The OFS –originally proposed to be used in an Alpha architecture– can be directly employed in the new x86 environment to filter both load and store accesses to the SQ.

The filtering capability of a single OFS is poor. For improving it, and similarly to the YLA management, in [1] authors propose to add some address information, increasing the number of OFSs (**multi-OFS scheme**). Besides this set of OFSs, the mechanism also requires a Main-OFS (identical to the OFS explained above) and a PAS register (*Pending Address Stores*, that records the number of unresolved stores). Depending on the PAS value –equal or bigger than zero– the multi-OFS scheme tries to filter based on the OFSs set or on the main-OFS respectively. Thus, the single-OFS performance is improved. However, the OFSs set updating process turns very complex: when a store commits, a sequential search over the SQ is required in order to find the next older in flight store to the same address. This fact significantly increases the amount of accesses to the SQ –and therefore the store queue energy consumption– which in [1] was not considered.

We can use the multi-OFS scheme in the new architecture for filtering both store and load accesses to the SQ. In the architecture employed in [1], only loads accessed the SQ, however, in the new one, the technique can be applied analogously to the stores. On the other hand, in the case of loads, we have to perform some changes, to take into account the LSAP information as follows: If PAS is bigger than zero at load issue and LSAP predicts aliasing, we can not trust the multi-OFS, and the main-OFS is checked. However, if LSAP does not predict aliasing, we can trust the multi-OFS even though unresolved stores exist (PAS bigger than zero).

The second technique which our design is based on is described in [2], where the authors propose

to use two **Bloom Filters** (BF) [12] –one for loads and another for stores– to filter associative accesses to LQ and SQ. Let us pay attention to the Bloom Filter for avoiding accesses to the SQ. The BF is made up of a table of counters. When a store issues, it indexes (address-based) an entry of the filter, incrementing the corresponding counter. When a store commits, the correspondig counter is decremented. Then, when a load issues, the BF is consulted in order to check the correct memory dependence enforcement. If the entry contains a value bigger than 0, a potential memory dependence exists and therefore the load conservatively performs an associative search to the SQ. Otherwise we know that no matching between memory addresses exists, so the SQ scan is avoided.

As Bloom Filters only provide information about resolved stores, and in the x86 environment –due to special features of st-st forwarding– information about unresolved ones is also required, some changes are neccessary to employ BF in our new architecture:

1. When a load issues, if the LSAP predicts aliasing with a prior unresolved store, the SQ search can not be filtered although the BF advices it (the architecture from [2] did not use a dependence predictor). In this case, the load must access the SQ to find the closest earlier unresolved store and create a dependence on it. Since we know for sure that a dependent resolved store does not exist, this checking is cheaper than a common associative search.
2. In the new architecture, when a store issues, it accesses the SQ to find a previous store to the same address or an unresolved one. Since the BF only reports information about resolved stores, it can not filter this kind of accesses (in our hybrid mechanism we will include an extra register that allows the BF usage in this case).

Due to the very complex updating process of the OFSs set in the multi-OFS scheme its usage is inappropriate. On the other hand, using only a Bloom Filter loses the opportunity of filtering more accesses based on timing information. Hence, we propose to combine a single OFS, which reports timing information and requires a simple updating process, and a BF, which provides address-based information. Besides, we include an additional register, called PAS (Pending Address Stores), to know if all in flight stores in the processor are resolved. In the next subsections we describe in detail our proposed mechanism.

4.2.1. Filtering of Loads accessing the SQ

When a load instruction issues, it consults the OFS and checks the PAS (Fig. 2, first stage). If the load is older than OFS, we can assure, based just on timing information, that a store to load forwarding is not needed for that load, and both the BF access and the SQ associative search can be avoided. Otherwise, the load goes to the second stage of our mechanism, the Bloom Filter (Fig. 2, second stage, top). If the corresponding BF entry and the PAS register are both zero [3], or if the BF entry is zero and the LSAP does not predict a dependence with a previous store, we can guarantee, based now on address information, that the load can not receive its data via a forwarding, and again the SQ scan can be avoided. If the BF entry is zero, but PAS is bigger than zero and the LSAP predicts a dependence, an SQ search must be performed to find the closest unresolved store. However, as we said before, this is a simpler and cheaper access than a common associative one, since we do not need to compare addresses. Finally, if the value recorded in the corresponding BF entry is bigger than zero, the normal SQ associative access is carried out (Fig. 2, third stage).

[3] PAS being zero means that we do not need to pay attention to LSAP since no unresolved stores exist

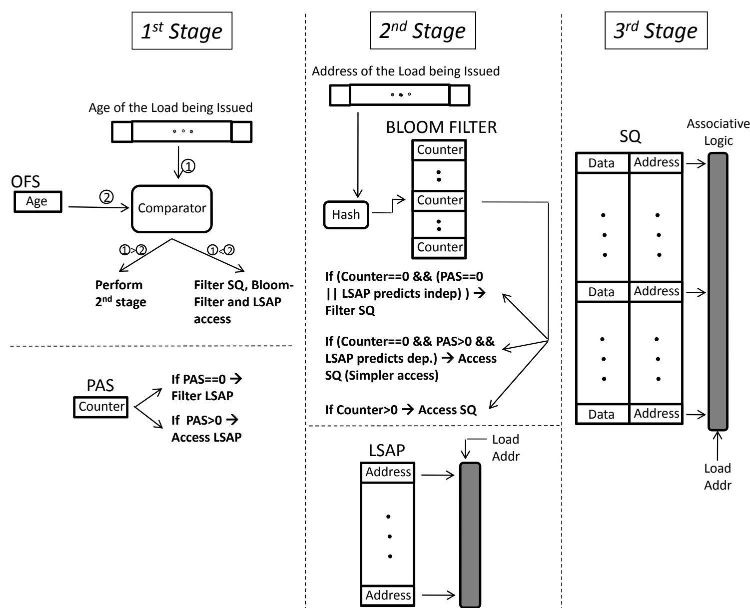


Figure 2: New SQ implementation: LOAD ISSUE.

4.2.2. Filtering of Loads accessing the Predictor

The new architecture we are using allows for new filtering opportunities. One of them is the chance to filter some accesses of load instructions to the LSAP. The first opportunity to avoid such lookups happens when a load checks the OFS (Fig. 2, first stage, top): If the load is older it means that no previous stores exist; hence the LSAP information is irrelevant and the predictor search turns unnecessary.

In order to increase the LSAP filtering capability, we can take advantage of the PAS register (Fig. 2, first stage, bottom): If it is zero, we can also aviod the LSAP access, since all stores are resolved and therefore there is nothing to predict. Otherwise, the LSAP search is required (Fig. 2, second stage, bottom).

4.2.3. Filtering of Stores accessing the SQ

Recall that in the new architecture each store checks the SQ in order to find previous stores to the same address. Once again, we can filter some of these searches. When a store instruction issues, it compares its age with the OFS value (Fig. 3, first stage). If they are equal, we can guarantee that no prior stores exist, and therefore both the BF access and the SQ lookup can be avoided. Otherwise, the store consults the BF, hashing its address. If the corresponding entry and PAS [4] are zero, we can assure, based now on address information, that this is the only one store to that address, and the SQ search can be avoided. If the BF entry is zero but PAS is bigger than zero,

[4]Note that we use the PAS register in combination with the BF for being able to filter some of the SQ accesses. The BF provides information about resolved stores, while the PAS register reports information about unresolved ones. As we mentioned before, without this register the BF can not be trusted in this context.

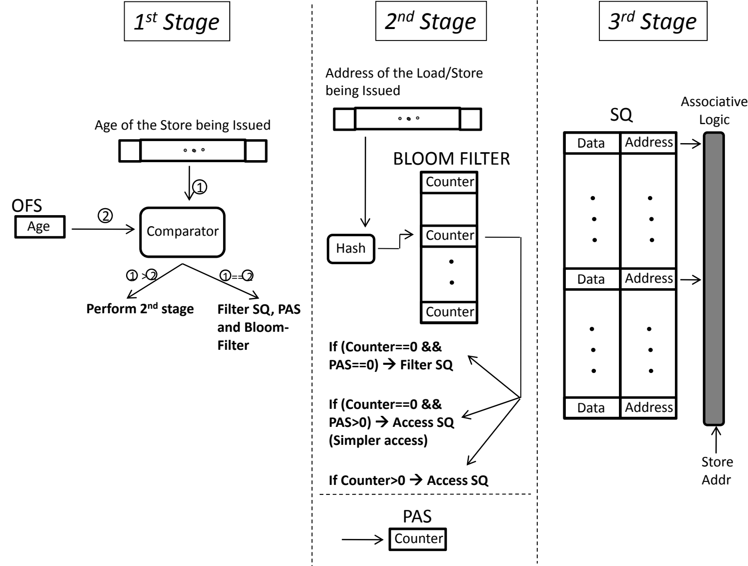


Figure 3: New SQ implementation: STORE ISSUE.

we have to perform an SQ lookup to find the closest unresolved store. Again, this is a simpler and cheaper access than a normal associative search. Finally, if the entry is bigger than zero, a normal SQ lookup is carried out (Fig. 3, third stage).

This proposed hybrid mechanism exhibits several advantages compared to the multi-OFS [1] and the Bloom Filter [2] schemes. First, thanks to the combination of timing and address information, the number of filterings grows significantly, as we will demonstrate in the evaluation section. Second, unlike the multi-OFS scheme, the updating process for our single-OFS is very simple and incurs no energy cost: when a store instruction commits, the OFS is just updated with the age of the store accommodated in the contiguous SQ entry. Third, in our approach the Bloom Filter access is avoided when the 1st stage filters the SQ search based on the OFS (note that the OFS access is much cheaper than the BF one). On the contrary, in the BF scheme, the filter is always accessed at load/store issue.

5. Experimental Framework

We have evaluated our proposed design using the PTLsim simulator [10]. The microarchitecture models the default PTLsim configuration that results from the merging of different features of an Intel Pentium 4 [7] [8], an AMD K8 and an Intel Core 2 [9]. Some of the main simulation parameters are listed in Table 1.

The evaluation of our proposal has been performed using 24 benchmarks from the SPEC CPU2006 suite, compiled for the x86 instruction set. The technology parameters correspond to $0.1\mu m$, with a 1.9V V_{dd} and 1.25GHz clock. We simulate regions of 100M instructions after reach-

Table 1: Simulation parameters for default configuration

Branch predictor	Bimodal: 2 bits, 16K BHT, 2K BTAC
Instruction Fetch queue size	32
ROB size	128
LSQ size	80 (LQ: 48, SQ: 32)
LSAP size	16
Physical Registers	256
Fuctional Units (INT)	8: 4 ALU (2 INT, 2 FP), 2 Load, 2 Store
Fetch/Decode/Issue/Commit width	4/4/4
L1 Instruction Cache L1 Data Cache	32KB (4 way, 64B line) 16KB (4 way, 64B line, 2 cycles latency)
L2 Data Cache L3 Data Cache	256KB (16 way, 64B line, 6 cycles latency) 4MB (32 way, 64B line, 16 cycles latency)
Main memory latency	140 cycles

ing the triggering point, that marks the beginning of code area in which the application behavior is representative of the overall execution. In the next section, we explain how we select these points.

5.1. Benchmarks instrumentation for the PTLSim

As we mentioned above, in order to use a benchmark in the PTLSim environment, some previous instrumentation is necessary, in which we set a representative point in the program for beginning the simulation. The PTLSim framework already provides these triggering points for the SPEC CPU2000 suite. However, due to the relatively novelty of the SPEC CPU2006 suite, this work is not done yet for this set of applications. We introduce here a first attempt to this instrumentation –that we employ to generate the results shown in the experimental section– that can be used by any interested reader. Table 2 details some parameters utilized in the benchmarks' instrumentation.

5.2. Energy Model

For dynamic energy modeling, the PTLsim is modified to reflect the common approach of using a split LQ-SQ organization. A memory instruction incurs two types of accesses: reading or writing data/address in a particular entry (called *direct access* in our model), and associatively searching in a whole queue (called *associative access* in our model). We used the CACTI 100nm model [20] to quantify the energy consumption of this kind of accesses. According with our experiments, the energy cost of an LQ direct access accounts for 32% of an associative one, while in the SQ this percentage reaches the 40%. Next, we summarize the main actions considered in order to measaure the energy consumption derived from LSQ management:

1. **Base processor - Store Issue** (Fig. 4): Recall that store issue is split into two phases. During the first one, two associative searches are performed unconditionally (one to the LQ and another to the SQ), and the store address is written into the corresponding SQ entry. Besides, if a dependent previous store is found, we have to read its data for merging. During the second phase, the SQ and LQ are scanned again, and the data of the store is possibly merged and written into the SQ. Note that if the address and data of a store are generated at the same time, we can combine their writting to the SQ in a single and cheaper access.
2. **Base processor - Load Issue** (Fig. 5): When a load instruction issues, it performs two associative searches (one to the SQ and another to the LSAP), and writes its address and data into the corresponding LQ entry. Furthermore, if a dependent previous store is found, its data must be read and forwarded to the load. In those cases where the load has to be re-issued, the SQ is scanned again.

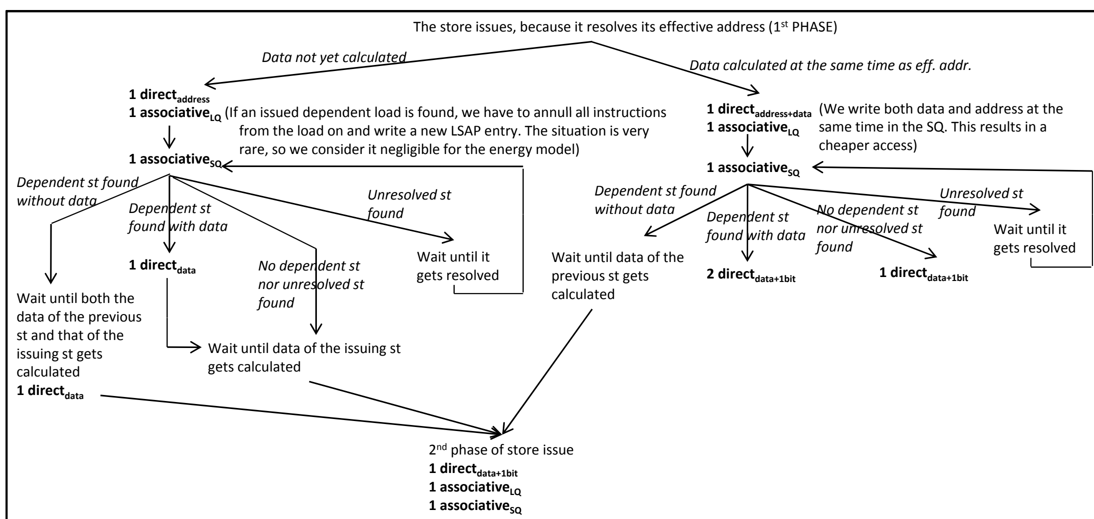


Figure 4: Store's accesses in the Baseline.

3. **Hybrid approach - Store and Load Issue** (Fig. 6 and Fig. 7): Our approach tries to filter as many associative searches as possible. For doing that, it consults the multi-YLA, the OFS, the PAS, and the Bloom Filter. If the filtering does not succeed, the accesses are performed like in the baseline (see above).
4. **BF and Multi-OFS - Store and Load Issue**: They can be easily deduced from the previous one.

6. Evaluation

6.1. LQ Filtering

We begin our analysis examining the results concerning the LQ filtering. Figure 8 illustrates the amount of LQ associative searches filtered in an x86 architecture using the **multi-YLA** and the **Bloom Filter** schemes, compared to the conventional design –baseline– in which every store accesses the LQ unconditionally. As shown in Figure 8(b) with even a single YLA register, an average of 35 percent of stores can be marked as safe after YLA checking, and their LQ searches filtered out. With 32 registers, the filtering efficiency is a remarkable 69 percent –only 31 percent of stores remains unfiltered– which highly overcomes the results obtained by the BF approach –49% on average for 256 entries–. Recall that this increase in filtering capability is obtained adding extremely simple hardware (just a few registers).

Figure 8(a) shows the detailed results of individual applications, trying to compare the multi-YLA scheme and BF approach for a similar amount [5] of hardware (8 YLAs vs 32-entry BF and 64 YLAs vs 256-entry BF). We see that there is variation across different applications as expected,

[5]The Bloom filter only needs 3 bits per entry to achieve a good performance. On the contrary, both the YLA or the OFS must save the ROB ID (age), so in our case they need 7 bits (ROB=128 entries) plus 2 more bits of control. Therefore, with the same number of bits, we can have a BF with 3x entries the number of YLAs or OFSs. Since a

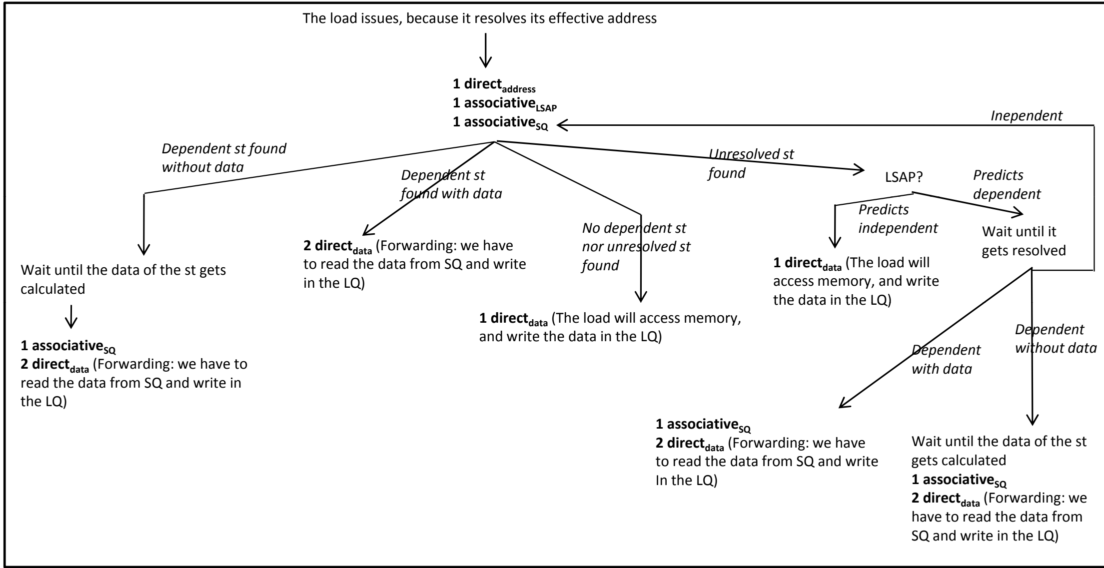


Figure 5: Load's accesses in the Baseline.

but the variation is not too significant. For 8 YLA registers the amount of searches filtered range between 98% for *sjeng* and 13% for *GemsFDTD*. Anyway, we confirm the higher filtering efficiency of multi-YLA respect BF.

6.2. SQ Filtering

In this subsection, we evaluate our new **Hybrid sheme** (explained in section 4.2), measuring the amount of SQ searches that our scheme is able to filter out, and comparing it against that of the two previous approaches: **Multi-OFS scheme** from [1] and **Bloom Filter scheme** from [2] but both adapted to the new architecture as explained in section 4.2. Recall that SQ accesses come from two different sources: loads checking if a st-ld-forwarding is feasible, and stores looking for a st-st-forwarding.

Figure 9 shows the results of this comparison. All the bars are computed compared to the conventional design. Figure 9(a) shows the SQ associative accesses filtered in each application, using the three previously explained schemes, with different structures sizes. The results illustrate how our proposal ovecomes multi-OFS and BF schemes. For example, in *mcf* application 46% of SQ searches are avoided when 8 OFS (multi-OFS scheme) are employed and 55% if a 16-entry BF is used (BF scheme), whereas our hybrid approach –1 OFS, 16-entry BF and a PAS register– manages to filter up to 66% of SQ lookups. In the same figure we can observe that the same tendency keeps across different applications (excluding a couple of benchamarks).

In Figure 9(b) we illustrate the average SQ accesses filtered which, of course, follow the trend showed in Figure 9(a). Thus, for a similar hardware amount, 32 OFS filter 32% of SQ lookups, a

---

factor of 3 should not be used, in the following comparisons we will always approximate to the closest power of two factor that is opposite to our interests.

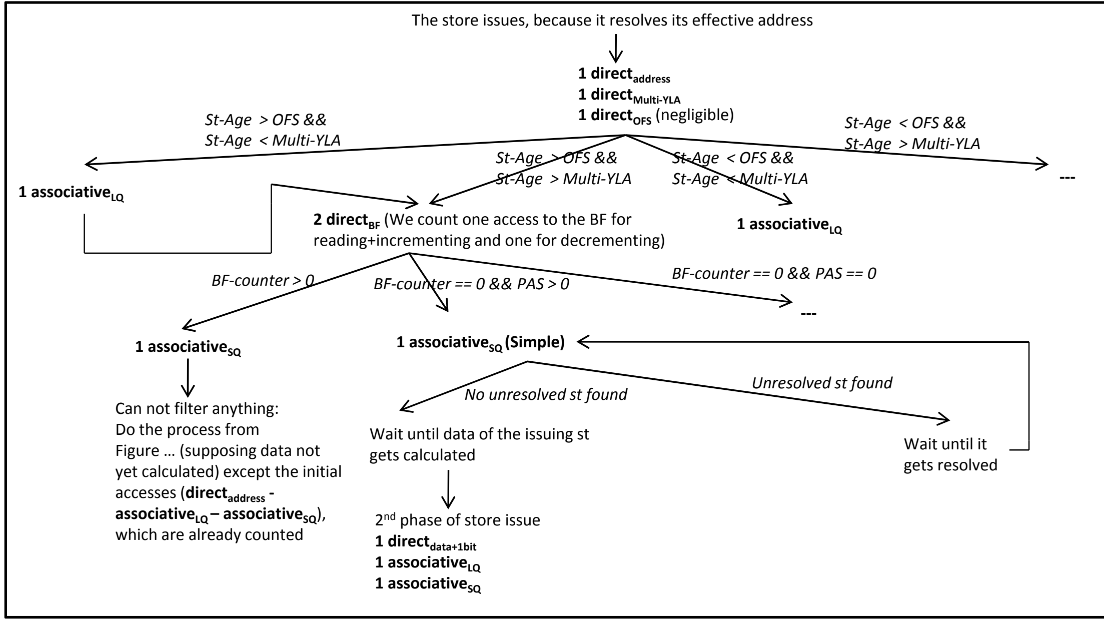


Figure 6: Store's accesses in our Hybrid Approach.

64-entry BF avoids 48% searches and our proposal achieves a 55% filtering rate. Figure 9(c) shows again the same results but organized in a slightly different way. Each group of bars represents a similar –not exact– HW amount (number of bits) for each of three schemes evaluated. The mentioned figure illustrates that the filtering capability of our design is significantly higher than that of the two other mechanisms. On average for all the sizes considered, our hybrid approach is able to filter around 6% more accesses than the original BF design, and around 24% more accesses than the multi-OFS scheme. For example, with the largest BF size employed –256 entries– our scheme filters more than 56% of original SQ associative accesses (18% with the OFS and 38% with the BF), while a BF with the same HW filters 49% and a multi-OFS (128 registers) only 34%.

One important issue is related with the complex updating process requiered in the Multi-OFS scheme: As we explained before, when a store commits, a sequential access over the SQ is carried out in order to update the corresponding OFS register. According to our experiments, the energy cost of each of these accesses is higher than that of an associative one. In our evaluation, we have considered the energy consumption of these two accesses to be identical (even though it is opposite to our interests). Thus, Figure 9 considers one extra associative access to the SQ for every committing store.

Compared to the BF approach, our hybrid proposal has an additional advantage: When we filter an SQ access employing the OFS (which happens for around 18% of the accesses), we avoid accessing the Bloom Filter, which is much more energy consuming than the OFS itself. On the contrary, the BF approach accesses the filter for every issued store.

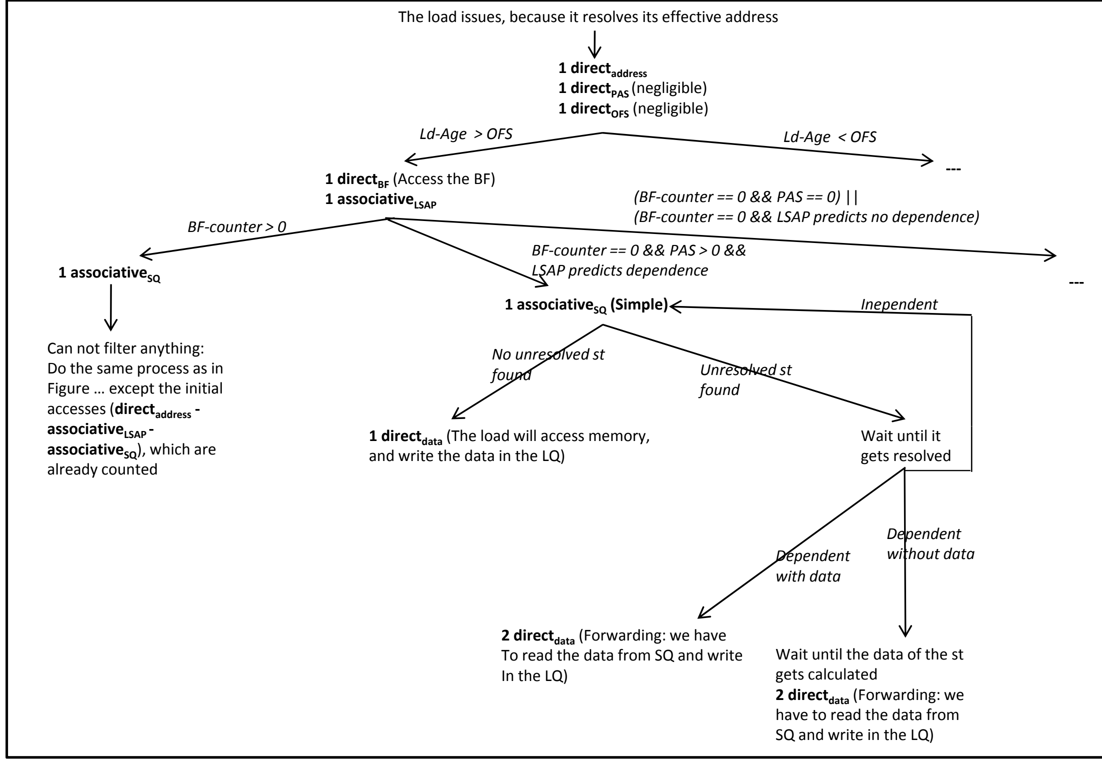


Figure 7: Load's accesses in our Hybrid Approach.

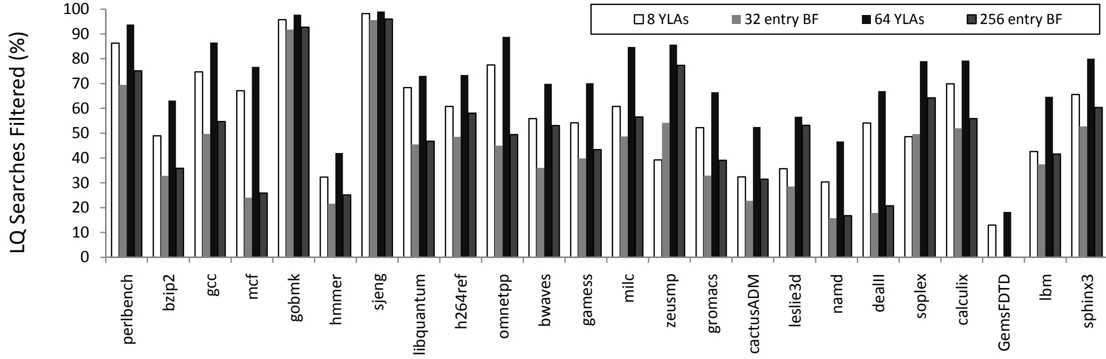
6.3. LSAP Filtering

In this section we analyze the filtering capability of our mechanism, evaluated for the dependence predictor. The energy cost of an LSAP access can be modeled similarly to an LQ/SQ access, since the predictor is implemented as a fully associative table. According to our results, 42% of LSAP accesses are avoided with our hybrid approach. Values obtained for each appliacation range between 17% in *hmmer* and 81% in *sphinx3*. This filtering delivers extra energy savings with just a very limited –PAS and OFS registers– hardware cost.

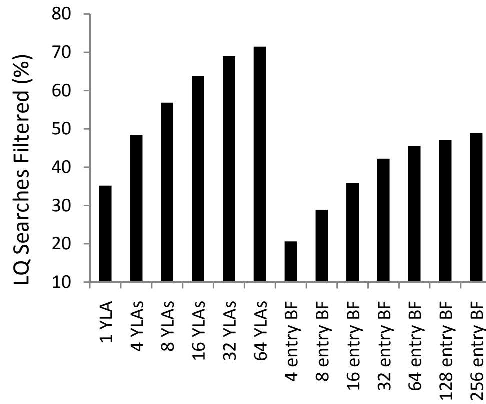
6.4. Energy comparision

In this section we report results for the energy savings obtained with the three schemes under evaluation over the baseline, according with the LSQ Energy Model explained in Section 5.2. Figure 10(a) shows the LSQ dynamic energy savings per application with roughly the same amount of hardware in each approach. We can observe that our hybrid scheme overcomes the energy reduction obtained by the other two mechanisms. Furthermore, the percentage achieved with our proposal ranges between significant 28% for *GemsFDTD* and 52% for *soplex* when 1 OFS plus 128-entry BF and a PAS register are used.

Figure 10(b) compares again the three approaches for a similar HW amount. Clearly, the hybrid scheme reports the highest dynamic energy savings. For example, the Bloom filter (with 256 entries)



(a)



(b)

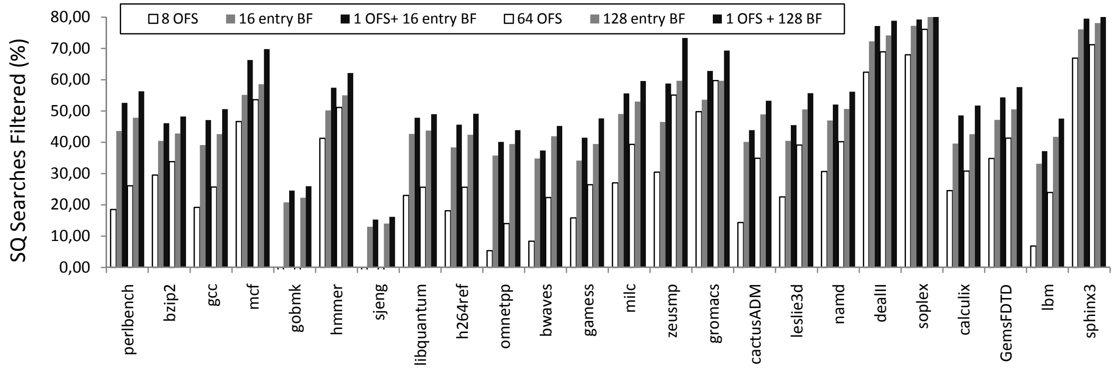
Figure 8: **(a)** LQ filtered accesses per application. **(b)** LQ filtered accesses on average. Both compared with the conventional LSQ explained in Section 2.

saves around 25%, the Multi-OFS (128 registers) 22%, and our design (with 256-entry BF) 38% of the dynamic energy consumption of a conventional LSQ. Note that we are including in the LSQ energy consumption the LSAP energy cost –of course, apart from LQ/SQ energy waste–.

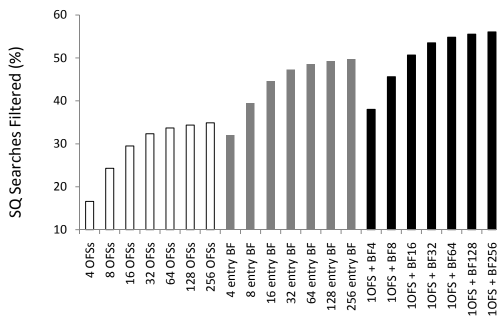
6.5. Other experimental results

In this section, we extend our analysis to clarify some important aspects of our approach. On average, among all issued loads, only a negligible 0.08% become premature loads, demonstrating that memory dependence violations are relatively infrequent (thanks in part to the usage of a dependence predictor). Besides, around 11.5% of the loads receive their data via st-ld forwarding, and around 14.5% of the stores must combine their data with a previous one (st-st forwarding), which suggests that the complex disambiguation hardware in modern micropocessors is underutilized and therefore it exists a wide room for improvement.

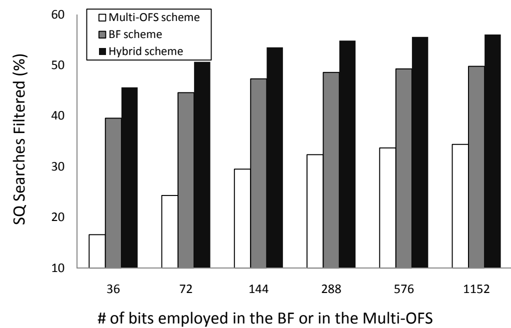
Concerning the Dependence Predictor (LSAP), recall that every issued load checks it to know weather the load will depend on a previous store, whereas issued stores update it when a dependence



(a)



(b)



(c)

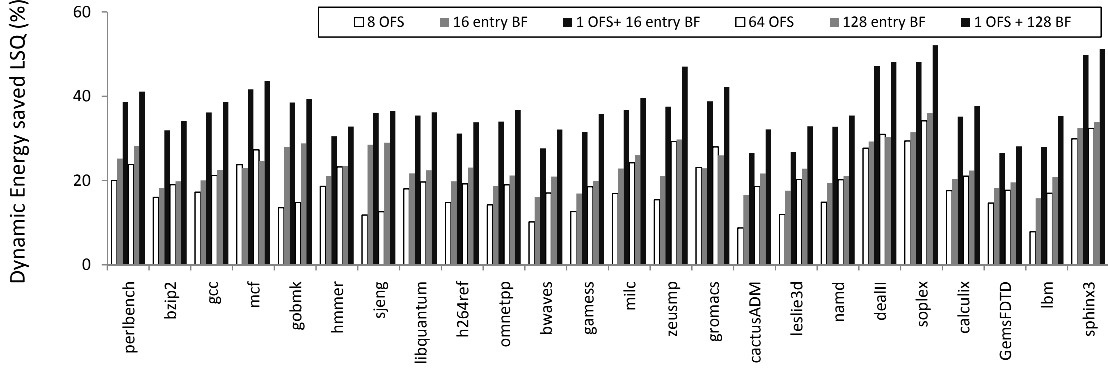
Figure 9: **(a)** SQ filtered accesses per application. **(b)** SQ filtered accesses on average for several number of entries. **(c)** SQ filtered accesses for the same HW amount: each group of 3 bars uses the number of bits. All figures compared with the conventional LSQ.

violation is encountered. Among all issued loads, around 6.5% on average are predicted to be dependent on a previous store (70% of these predictions hit, while 30% miss). This demonstrates the utility of the LSAP, since without it, the number of premature loads would increase from 0.08% to around 5.5%, degrading performance significantly. Besides, as memory dependece violations are rare, only 0.17% of the stores need to update the predictor.

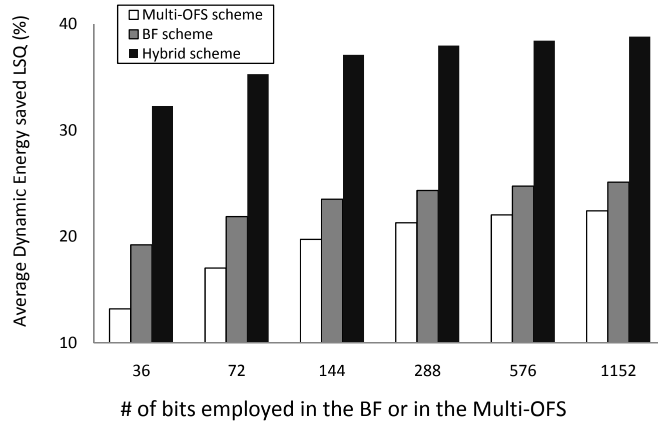
Finally, we have also taken some IPC and Micro-IPC results, just to provide an idea of the range in which we move (recall that performance is not affected by any of the techniques studied). The average IPC is 1.3 and the Micro-IPC is 1.9.

7. Conclusions

In this paper we have proposed a straightforward memory disambiguation scheme which, compared with previous approaches, reports higher efficiency –in terms of number of accesses to the LSQ structure– and uses less and simpler hardware. With just a small set of age registers (YLAs) and based on simple timing comparisons, our proposal is able to avoid most of LQ associative



(a)



(b)

Figure 10: **(a)** Dynamic energy saved over the conventional LSQ, per application. **(b)** Dynamic energy saved over the conventional LSQ, on average.

searches performed in a conventional design. Similarly, for SQ filtering we use an age register (OFS) and a tiny Bloom Filter, containing timing and address information respectively, which also filter a significant portion of SQ lookups. Finally, the OFS –aided by an extra register (PAS)– also rules out a moderate amount of accesses to the LSAP. Note that our hybrid approach is a filtering technique, so it has no performance impact.

We have analyzed our approach and the most relevant previous ones in a different microarchitectural model than that of the prior works. This model, besides of resulting more appealing, enables for new types of filtering which lead to extra energy savings. Overall, on average, our design manages to avoid around 64% accesses to LQ (using 16 YLAs), 55% to SQ (employing an OFS plus a small BF of 64 entries) and 42% to LSAP, with respect to a conventional design in which all associative searches are performed unconditionally. Besides, on average, our design saves up to 39% of the dynamic energy of a conventional LSQ, while the Bloom Filter only saves 25% and the

Multi-OFS 23%.

References

[1] F. Castro, D. Chaver, L. Pinuel, M. Prieto, F. Tirado, Using Age Registers for a simple Load Store Queue Filtering, Journal of Systems Architecture 55 (2) (2009) 79–89.

[2] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, S. Keckler, Scalable Hardware Memory Disambiguation for High ILP Processors, in: Intl. Symp. on Microarch., San Diego, California, 2003, pp. 399–410.

[3] I. Park, C. Ooi, T. Vijaykumar, Reducing Design Complexity of the Load/Store Queue, in: Intl. Symp. on Microarch., San Diego, California, 2003, pp. 411–422.

[4] T. Sha, M. Martin, A. Roth, Scalable Store-Load Forwarding via Store Queue Index Prediction, in: Intl. Symp. on Microarch., Barcelona, Spain, 2005, pp. 159–170.

[5] F. Bower, D. Sorin, L. Cox, The impact of dynamically heterogeneous multicore processors on thread scheduling, Micro, IEEE 28 (3) (2008) 17–25.

[6] M. D. Hill, M. R. Marty, Amdahl's law in the multicore era, IEEE Computer 41 (7) (2008) 33–38.

[7] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, P. Roussel, The Microarchitecture of the Pentium 4 Processor, Intel Technology Journal.

[8] D. Boggs, A. Baktha, J. Hawkins, D. Marr, J. Miller, P. Roussel, R. Singhal, B. Toll, K. Venkatraman, The Microarchitecture of the Intel™Pentium™4 Processor on 90nm Technology, Intel Technology Journal 8 (1) (2004) 1–17.

[9] Copenhagen University College of Engineering, The Microarchitecture of Intel and AMD CPU's: an Optimization Guide for Assembly Programmers and Compiler Makers (2009).

[10] M. T. Yourst, PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator, in: Intl. Symp. on Performance Analysis of Systems and Software, San Jose, California, 2007, pp. 23–34.

[11] M. Yourst, PTLsim Users Guide and Reference: The Anatomy of an x86-64 Out of Order Superscalar Microprocessor, http://www.ptlsim.org/documentation.php (2007).

[12] B. Bloom, Space/Time Trade-offs in Hash Coding with Allowable Errors, Communic. of the ACM 13 (7) (1970) 422–426.

[13] G. Chrysos, J. Emer, Memory Dependence Prediction Using Store Sets, in: Intl. Symp. on Computer Architecture, Barcelona, Spain, 1998, pp. 142 –153.

[14] S. Sethumadhavan, F. Roesner, D. B. J. Emer, S. Keckler, Late-Binding: Enabling Unordered Load-Store Queues, in: International Symposium on Computer Architecture, San Diego, California, 2007, pp. 347–357.

[15] H. Cain, M. Lipasti, Memory Ordering: A Value-based Approach, in: International Symposium on Computer Architecture, Munich, Germany, 2004, pp. 90–101.

[16] A. Roth, Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization, in: International Symposium on Computer Architecture, Madison, Wisconsin, 2005, pp. 458–468.

[17] S. Subramaniam, G. Loh, Fire-and-Forget: Load/Store Scheduling with No Store Queue, in: International Symposium on Microarchitecture, Orlando, Florida, 2006.

[18] F. Castro, L. Pinuel, D. Chaver, M. Prieto, M. Huang, F. Tirado, DMDC: Delayed Memory Dependence Checking through Age-Based Filtering, in: Intl. Symp. on Microarch., Orlando, Florida, 2006.

[19] F. Castro, R. Noor, A. Garg, D. Chaver, M. Huang, L. Pinuel, M. Prieto, F. Tirado, Replacing Associative Load Queues: A Timing-Centric Approach, IEEE Transactions on Computers 58 (4) (2009) 496–511.

[20] P. Shivakumar, N. Jouppi, CACTI 3.0: An Integrated Cache Timing, Power, and Area Model, WRL Tech. Rep. No. 2, Western Research Laboratory (Aug. 2001).

Table 2: SPEC-2006 instrumentation for PTLSim

Benchmark	File modified (in which we insert the invokation to PTLSim)	Line modified
435.gromacs	md.c - Invoked by mdrun.c, that contains the main() function	407 - We jump all variables initializations
445.gobmk	main.c	834, 846, 861, 876, 889, 913, 940, 963, 985, 1014, 1045, 1063, 1087, 1103, 1123, 1142, 1149, 1160 - Instrumented inside each entry of a case, since this way we avoid all initialization and I/O operations
447.dealII	step-14.cc - Contains the main() function, that invokes the run function and the run definition	4064 - It is in the Framework dim::run method, before going into the for loop that performs the main numeric analysis of the program
450.soplex	example.cc - Contains the main () function, that invokes solve(), which resolves the problem	447 - We avoid initializations of variables, readings of files, and printings on screen. NOTE: For simulating 100 million instructions we have to use the ref entry, that generates the files pds-50.mps and ref.mps (../../ptlsim ./soplex -s1 -e -m45000 pds-50.mps // ../../ptlsim ./soplex -m3500 ref.mps)
454.calculix	CalculiX.c - Contains the main() function	182 - After initializing the variables, and before the main loop that performs the main calculations
456.hmmer	hmmcalibrate.c - Contains the main() function, that invokes main_loop_serial(), which performs the basic calculations	502 - After initializing variables in main_loop_serial()
458.sjeng	epd.c - Contains the function run_autotest(), invoked by main(). The function includes the main calculations	315 - Before the main loop and after all initializations. NOTE: Only can execute 30 million instructions
459.GemsFDTD	leapfrog.f90 - Contains the function SUBROUTINE leapfrog(nx,ny,nz), invoked from the main program (in the file GemsFDTD.f9)	197 - After variables initializations
462.libquantum	shor.c - Contains the main function	95
464.h264ref	lencod.c - Contains the main function	307 - After variable initialization and files reading. NOTE: There was a bug in the original distribution: We had to include file ptlcalls.h inside the directory 464.h264ref/run/build_base_amd64-m64-gcc41-nn.0000 and comment the line that defined the byte type (line 27), since was already defined in defines.h, and was giving trouble
465.tonto	mol_main.F90 - In the function read_keywords().	3518
470.lbm	main.c - In the main() function	40 - Right before the main loop
471.omnetpp	libs/cmdenv/cmdenv.cc - In function simulate()	293 - Before the if instruction
482.sphinx3	spec_main_live_pretend.c - In the main() function	167 - After the while loop where the input files are read and before the for loop where the calculations are performed
483.xalancbmk	XalanExe.cpp - In function xsltMain()	801
401.bzip2	spec.c - It contains the main() function	332 - After the initialization phase, and before invoking the compression level provided by the command line.
403.gcc	toplev.c - It contains the compile_file() function, that performs the compilation and writes an assembly file to an entry unit, where we insert the invokation to ptlSim.	2118
429.mcf	mcf.c - It contains the main() function.	157 - After the initialization phase (function primal_start_artificial()), and before beginning the calculation (function global_opt()).
436.cactusADM	flesh.c - It contains the main() function.	72 - After variables initialization (function CCTK_Initialise()) and before invoking function CCTK_Evolve().
437.leslie3d	tml.f - The whole program is implemented in this file	325
444.namd	spec_namd.C - It contains the main() function.	181 - In this line the initialization phase finishes.
400.perlbench	perlmain.c - It contains the main() function.	97 - In this line the function that performs the parser of the input files is finished (perl_parser()), and the perl programs are going to begin their execution (perl_run()).
410.bwaves	shell_lam.f - This function implements the main shell() subrutine.	159
416.gamess	gamess.F - It contains the main() function.	572 - The input/output operations are done, the configuration file is read, and the main operations are just to begin.
433.milc	control.c - It contains the main() function.	35 - The main loop is going to begin execution. All variables are initialized and the input configuration file is read.
434.zeusmp	zeusmp.F - It contains the main() function.	668 - The main loop is going to begin execution. All variables are initialized and the input configuration file is read.