Effective Instruction Prefetching via Fetch Prestaging

Ayose Falcón*
Barcelona Research Office
HP Labs
ayose.falcon@hp.com

Alex Ramirez Mateo Valero
Computer Architecture Department
Universitat Politècnica de Catalunya
{*aramirez, mateo*}*@ac.upc.edu*

Abstract

As technological process shrinks and clock rate increases, instruction caches can no longer be accessed in one cycle. Alternatives are implementing smaller caches (with higher miss rate) or large caches with a pipelined access (with higher branch misprediction penalty). In both cases, the performance obtained is far from the obtained by an ideal large cache with one-cycle access.

In this paper we present Cache Line Guided Prestaging (CLGP)*, a novel mechanism that overcomes the limitations of current instruction cache implementations. CLGP employs prefetching to charge future cache lines into a set of fast prestage buffers. These buffers are managed efficiently by the CLGP algorithm, trying to fetch from them as much as possible. Therefore, the number of fetches served by the main instruction cache is highly reduced, and so the negative impact of its access latency on the overall performance.*

With the best CLGP configuration using a 4 KB I-cache, speedups of 3.5% (at 0.09μm) and 12.5% (at 0.045μm) are obtained over an equivalent Fetch Directed Prefetching configuration, and 39% (at 0.09μm) and 48% (at 0.045μm) over using a pipelined instruction cache without prefetching. Moreover, our results show that CLGP with a 2.5 KB of total cache budget can obtain a similar performance than using a 64 KB pipelined I-cache without prefetching, that is equivalent performance at 6.4X our hardware budget.

1. Introduction

Instruction cache performance is an important limiting factor to current wide-issue processors. The processor backend depends on the instruction supply from the front-end, and delays from the cache hierarchy affect the instruction flow speed along the processor pipeline.

Cache performance is affected by three parameters: *miss rate*, *hit time*, and *miss time*. The common solution to improve cache performance is having a cache hierarchy, using a large first level (L1) cache (which implies low miss rate) with a fast access time (which implies low hit time). Subsequent levels in the cache hierarchy are useful when there is an L1 cache miss, providing a low miss time.

Year	1999	2001	2004	2007	2010
Technology (μm)	0.18	0.13	**0.09**	0.065	**0.045**
Clock Frequency (GHz)	0.5	1.7	4	6.7	11.5
Cycle time (ns)	2	0.59	0.25	0.15	0.087

Table 1. Technological parameters predicted by the Semiconductor Industry Association

In the last years, microprocessors are having a dramatic improvement in their clock rates, while memory technology has not followed the same trend. Table 1 shows the technology trends from the technology roadmap predicted by the Semiconductor Industry Association (SIA) [20]. In the future, it will be common to have processors of tens of gigahertz, with cycle times below 0.1 nanoseconds. This will increase even more the existing gap between microprocessor and memory hierarchy, so common solutions provided to improve cache performance, like increasing cache size or associativity, will no longer be effective.

Based on these predictions, a tradeoff between a large, multi-cycle access L1 cache (low miss rate, but high hit time), or a small one-cycle access L1 cache (low hit time, but high miss rate) is needed. Also, a fast cycle time and a large cache could be combined if cache accesses are pipelined. Cache pipelining does not actually reduce hit time or miss rate, but increases the throughput of cache responses by virtually decreasing the latency of cache hits. However, as more pipeline stages are added, the cost of a branch misprediction increases proportionally.

Figure 1 shows the effect of the L1 I-cache access latency on the overall performance (simulation parameters and cache latencies are described in Section 4). The `ideal` case assumes that all cache sizes can be accessed in one cycle, so performance grows with the cache size. In the `base` case, the performance gain of using a larger L1 I-cache is affected when the cache hit time increases. Including a new

* This work was done while Ayose Falcón was in the Computer Architecture Department at the Universitat Politècnica de Catalunya.

Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)
1530-2075/05 $ 20.00 **IEEE**

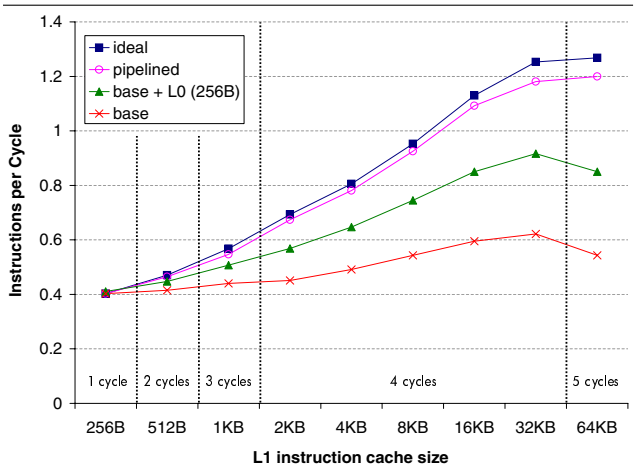


Figure 1. Effect of the L1 I-cache latency on the processor performance (0.045μm)

cache level with one cycle access (`base+L0` [11]) helps to maintain a low hit time, at the expense of a higher L0 miss rate. A `pipelined cache` allows for having a large multi-cycle cache with a virtual low hit time. The loss in performance of the pipelined cache with respect to the ideal case is due to the increase in pipeline stages [1].

The conclusion that can be extracted from Figure 1 is twofold: On the one hand, the use of small caches harms performance, due to their low hit rate. On the other hand, using large caches increases hit time and penalises on each cache access. The a priori appealing solution of using a large pipelined cache has a negative effect on performance due to the increasing branch misprediction latency. Moreover, cache pipelining involves extra energy (extra latches, multiplexers, clock and decoders) and area overhead (extra precharge circuitry, latches, decoders, sense amplifiers, and multiplexer), so implementing a large pipelined cache is not the most efficient solution [1].

In this environment, an effective solution to increase cache performance is *prefetching*. Its goal is to bring cache lines from the high hierarchy levels closer to the processor before the actual cache access, avoiding the possible cache miss, or covering part of the miss latency.

This paper presents *fetch prestaging*, which comprises a step further of the classic instruction prefetching. The goal of prestaging is not only to mask both miss rate and miss time caused by small caches (like prefetching), but also to reduce the negative effect of using large caches (either pipelined or not) on the processor's performance. Our technique, *Cache Line Guided Prestaging* (CLGP), tries to maximise the use of fast prestage buffers (an extension of classic prefetch buffers) while minimising the accesses to multicycle storages. Our results show that CLGP works well even in situations in which previous prefetching mechanisms do not obtain their best results.

1 We assume an ideal pipelining. A real implementation could increase the number of additional stages (higher penalty on mispredicted branches) or affect the cycle time [1].

2. Related work

2.1. Prior work on instruction prefetching

Many studies have been done in the field of instruction prefetching. We will focus on hardware schemes that execute a predetermined hardcoded prefetching algorithm.

Sequential prefetching. They profit from the implicit spatial locality of programs. The simplest method is increasing the cache line size to bring more instructions from the high cache levels to the low ones on each transfer. In the *next N-line* prefetching [21], the next N sequential cache lines are prefetched before they are needed by the CPU.

Non-sequential prefetching. A proposal that combines next N-line prefetching with target line prefetching is presented in [22]. In target line prefetching, a target prefetch table is used to store the successor line of the current fetched line. Thus, it is possible to prefetch along the target of a taken branch or a subroutine call.

Wrong-path prefetching [13] also combines next N-line prefetching with prefetching of instructions along the target path of a branch, despite the direction predicted by the branch predictor. When a branch is detected, instructions from both execution paths are prefetched: the fall-through path using the next line prefetching scheme and the target path using the target line prefetching.

Advanced PC prefetching. These techniques base their performance on the behaviour of the branch predictor, but implementing some kind of run-ahead from the main processor flow to initiate prefetches early enough to arrive on time and hide as much memory latency as possible. A scheme with a small prefetching unit containing a look-ahead PC and a simple own branch predictor is proposed in [4]. This fast unit goes ahead prefetching instructions that will be later consumed by the main flow. Their results show that branch prediction based prefetching outperforms table based prefetching, since the accuracy of the branch predictor is always better than that provided by the target table.

Multi-Level Branch Target Prediction (MBTP) [24] uses a branch direction and target predictor capable of predicting multiple branches in order to lengthen the distance between the current PC and the Look-Ahead PC, and therefore the prefetching distance. *Branch History Guided Prefetching* (BHGP) [23] also uses branch instructions to initiate prefetches of blocks that will appear several branches later in the program execution. Unlike MBTP, BHGP prefetches entire candidate blocks instead of a single first block line plus a sequential prefetching for the rest of block lines, therefore increasing the timeliness of prefetched lines.

In *Fetch Directed Prefetching* (FDP) [16], the branch predictor goes autonomously generating predictions and storing fetch blocks into a fetch target queue, which will be later consumed by the I-cache. These fetch block requests are used to initiate prefetches into prefetch buffers [10, 6] to avoid cache pollution. An improved FDP architecture is presented in [17], which integrates the FDP mechanism with an energy efficient I-cache. The goal is to reduce power consumption, without sacrificing performance.

We also use a decoupled fetch unit [15], with a queue that separates the fetch block generation from the fetch block consumption. Entries from this queue are also used to initiate prefetches. However, our *Cache Line Guided Prestaging* does not use prefetch buffers only as an additional storage to avoid pollution. Indeed, our prefetch buffers have more functionality so we call them "*prestage buffers*". We store instructions several stages ahead of the I-cache access. We exchange the role of I-cache and prestage buffer: prestage buffer is now the main instruction supplier, while the I-cache remains an *emergency cache*.

2.2. Technology and clock scaling

Recent studies have shown that as feature sizes shrink and clock cycle times decrease, the percentage of chip that can be reached in one cycle decreases dramatically [12]. In such situation in which the wire delay causes that signal propagation exceeds the clock cycle, it is unfeasible to have cache structures accessible in one cycle [2, 7].

The effect of the increasing cache miss latency on some prefetching algorithms is also studied in [13], detecting that wrong-path prefetching tolerates better the latency than other previous schemes. The same scenario was studied in [8], but applied to branch predictors. The results presented show that the processor performance drops precipitously as the branch predictor delay increases due to technological constraints.

In this work we also deal with technology and clock scaling, but applied to instruction fetch. We will show that cache latency increase implies that new mechanisms independent on the cache size and latency should appear.

3. A cache-latency insensitive I-fetch

We begin this section with a detailed description of a related technique we will compare to in the results section. Next, we present an in depth description of our proposal.

3.1. Fetch Directed Prefetching (FDP)

The basis of Fetch Directed Prefetching [16] is the use of a decoupled fetch architecture with a fetch target queue (FTQ) that separates fetch block prediction from fetch block consumption. Using a queue that decouples the prediction mechanism from the cache access allows the branch predictor to run ahead at a higher speed, providing multiple fetch requests that will be stored in the FTQ [15].

FDP uses fetch blocks from the FTQ and enqueues new prefetch requests in a prefetch instruction queue, where they remain until they can be served. Prefetched cache lines are stored in a separate fully associative prefetch buffer waiting to be fetched. Indeed, at the fetch stage, both the I-cache and prefetch buffer are searched in parallel looking for the cache line requested.

To avoid unnecessary prefetches, [16] presents some filtering techniques that prevent from prefetching instructions that are already stored in the I-cache. We have obtained the best performance for FDP using *Enqueue Cache Probe Filtering*, which uses an additional tag port (or replicated tags) prior to enqueuing new prefetch requests, to check whether the instructions to prefetch are already in the I-cache or not. This is the policy we will compare to in the results section.

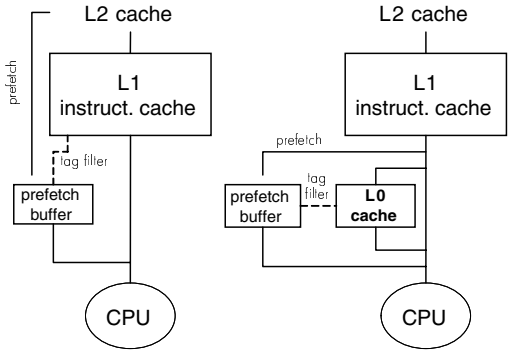
In FDP, when a line from the prefetch buffer is used by the fetch unit, it is transferred to the I-cache and the prefetch buffer entry is marked as available for new prefetches. So, the prefetch buffer replacement mechanism is simple: every entry is marked as replaceable when it is used, and subsequent accesses to the same line will hit in the I-cache. Although this is a good strategy if both the prefetch buffer and the L1 I-cache have the same hit latency, it is not as good if the I-cache has more than one cycle of access latency since the subsequent fetches will be penalised.

3.1.1. Including an L0 cache to reduce L1 I-cache latency impact on FDP. FDP does not prefetch those cache lines that are already in the lower cache level, which can harm performance if the lower level (i. e., L1) cache has a multi-cycle access. To solve this, we implement FDP including an L0 cache placed between the CPU and the L1 I-cache (Figure 2(a)). At the fetch stage, both L0 and L1 I-caches (apart from prefetch buffer) are accessed in parallel. If there is an L0 hit, the L1 latency is avoided [2].

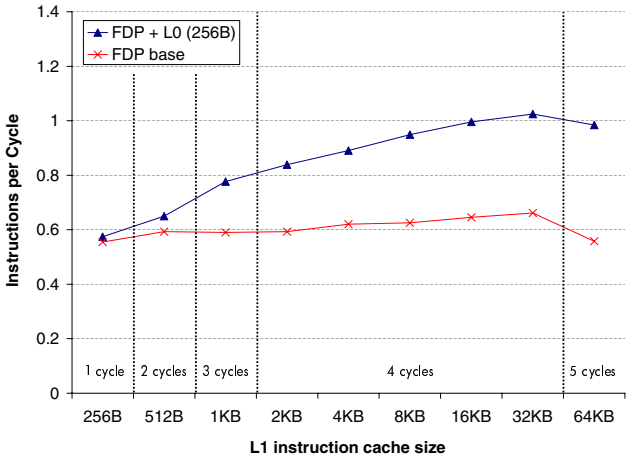
L0 cache has the same size of the prefetch buffer, i. e. the maximum size that can be accessible in a single cycle for each technological process. To benefit from the new cache level, prefetches are now served by the L1 I-cache; if there is an L1 miss, prefetches are served by the L2 cache as in the original FDP algorithm. Besides, on a prefetch buffer hit, the cache line is moved to the L0 cache, not to the L1.

Figure 2(b) compares the performance of FDP with and without an L0 cache. FDP without an L0 cache maintains a similar as the I-cache size increases. Cache lines are provided by the multi-cycle L1 I-cache instead of the prefetch

2 Note that accessing in parallel to L0 and L1 can only improve performance compared to having only an L1 cache. The L0 inclusion does not follow power saving purposes.



(a) Cache hierarchy with and without L0 cache



(b) Performance results for a 0.045μm process

Figure 2. FDP with and without an L0 cache.

buffers, penalising the fetch stage. Including an L0 cache allows to tolerate the increasing L1 I-cache latency. We consider that this is a fair adaptation of FDP to the environment proposed in this paper, and this will be the configuration to compare to in the results section.

3.2. Fetch prestaging

The goal of fetch prestaging is to anticipate the fetch stage, so that the flow of instructions to fetch can be known beforehand. This lets put the instructions that will be needed in a near future in fast prefetch buffers, and so the fetch stage will not be later penalised due to a multi-cycle cache access. Multi-cycle cache access occurs in the following situations: i) the first level cache is small due to technological constraints and has many misses that are served by the second level cache; ii) the first level cache is itself a large, multi-cycle access cache. Accordingly, fetching from prefetch buffers becomes cheaper than fetching from caches with more than one cycle of latency.

Cache lines should be in the prefetch buffers just when they are used for the first time. Therefore, prestaging relies on prefetching to load cache lines into the prefetch buffers before the real fetch occurs. But prestaging adds a new relationship between cache lines and prefetch buffers: cache lines should be kept into prefetch buffers until they are used for the last time. As prefetch buffer space is highly valuable, lines should leave prefetch buffer as soon as it is known that they will not be used again in a near future, leaving room for new lines.

3.2.1. Cache Line Target Queue (CLTQ). Our mechanism uses a decoupled fetch with a branch predictor that generates fetch entities and an instruction fetch that consumes them. Our decoupled fetch is slightly different from the one proposed in [15]. We also implement a queue that is filled with fetch blocks from the branch predictor. However, the contents of this queue is stored in a finer granularity. Before entering the fetch queue, fetch blocks are divided into fetch cache lines, and each fetch cache line is stored in a different fetch queue entry. We call our fetch queue *cache line target queue* (CLTQ), to differentiate it from the original fetch target queue (FTQ) as implemented in [15].

In essence, FTQ and CLTQ hold the same requests to the I-cache. The difference lies in the semantic information they store, i. e. the interpretation of the requests from the branch predictor. While the FTQ stores fetch blocks whose boundaries are taken branches (according to the predictions done by the branch predictor), the CLTQ stores fetch cache lines whose boundaries are the limits of a cache line. FTQ will have less entries occupied, but each entry holds multiple cache lines; CLTQ will have more occupied entries, but each entry holds only one cache line.

Each CLTQ entry contains three fields: the fetch cache line; a '*prefetched bit*' that indicates whether the cache line of this entry has been prefetched or not; and an '*occupied bit*' that indicates whether or not the entry is occupied by a fetch cache line that has not been fetched yet.

3.2.2. Prestage buffers. CLTQ entries are used to initiate new prefetches that will be stored in a fully associative buffer, namely *prestage buffer*. Each prestage buffer entry has four fields: the prefetched I-cache line; a '*consumers counter*' that initially is set to 0; a '*valid bit*' that indicates whether the cache line has already arrived to the prestage buffer from the cache hierarchy or not; and a '*LRU*' (*Least Recently Used*) field used by the replacement mechanism.

Prestage buffers can be seen as an enhancement of prefetch buffers. They are not only an additional storage to keep cache lines but also they become the main instruction supplier. The contribution of prestage buffers is the inclusion of the *consumers counter* field. It indicates how many entries in the CLTQ will produce a fetch of instructions stored in this prestage buffer entry. In other words, the consumers counter field is a metric of how much time should

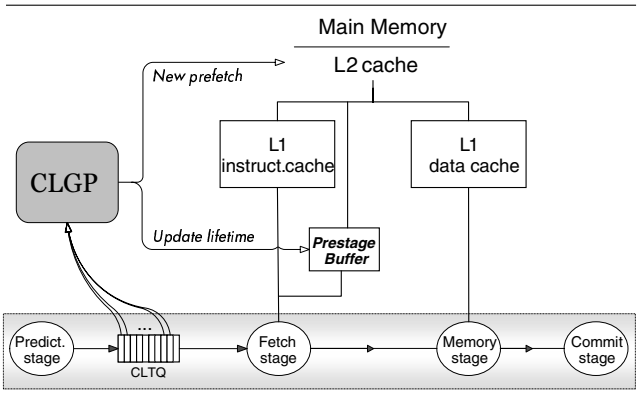


Figure 3. CLGP architecture.

this line be stored in the prestage buffer to serve near future requests [3].

3.2.3. Cache Line Guided Prestaging (CLGP). Figure 3 presents an overview of the CLGP architecture. CLGP algorithm traverses the CLTQ looking for new requests to prefetch. CLGP does not perform any kind of filtering. The idea is to bring useful cache lines always to fast prestage buffers and to fetch from them. Our goal is not only to prefetch instructions to avoid L1 cache misses, but even to avoid the *hit penalty*.

When a fetch cache line from the CLTQ is selected for prefetching, CLGP checks first the prestage buffer contents. Depending on the cache lines (already prefetched or being prefetched) stored in the prestage buffer, CLGP will initiate new prefetches or update prestage buffer entries' lifetime. On the one hand, if the cache line is not in the prestage buffer and there is an available prestage buffer entry, a new prefetch begins. The LRU free entry from the prestage buffer is allocated for the new cache line to prefetch, and the consumers counter field is set to 1. The valid bit is unset and remains so until the cache line arrives, at which moment the bit is set and the line can be used by the fetch unit. On the other hand, if the cache line to prefetch is already in the prestage buffer, no new real prefetch is initiated. CLGP only increases the consumers counter field of the prestage buffer entry that contains the same cache line, thereby extending its lifetime.

In previous prefetching schemes proposed in the literature [6, 9, 10, 16], the prefetch buffer replacement mechanism is quite simple: prefetch buffer entries are set as available once they are used for the first time. Prestaging comprises a more sophisticated replacement strategy. Prestage buffer entries will be set as replaced only when CLGP is sure that this line will not be used in a near future. This is done by maintaining the relationship between a cache line stored in the prestage buffer and its consumers waiting in the CLTQ to be fetched.

Therefore, the consumers counter of each prestage buffer entry determines when the prestage buffer entry can be replaced. If a CLTQ request corresponds to a cache line stored in the prestage buffer, the consumers counter is increased. When a cache line is fetched from the prestage buffer, the consumers counter is decreased. A cache line remains in the prestage buffer as long as there are entries of the CLTQ which reference it. This mechanism allows us to increase the coverage of prefetched lines, since prefetched lines are not replaced from the prestage buffer until they are no longer needed by the fetch engine.

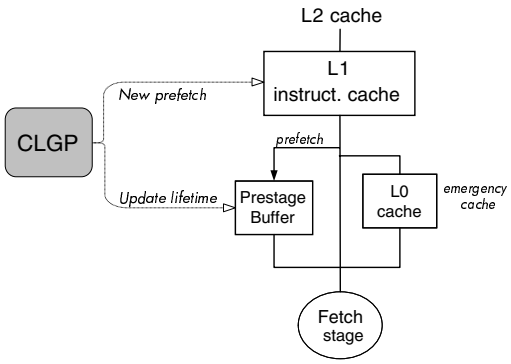
On a branch misprediction, CLTQ contents is flushed. Consumers counters from the prestage buffer are reset, indicating that all entries are available for new prefetches along the new correct path. However, cache lines into the prefetch buffer from the incorrect predicted path remain useful as long as the valid bit is set and prestage buffer are not occupied by new prefetches from the correct path.

As the purpose of our mechanism is to use the contents of the prestage buffer as much as possible, cache lines that will be used in the future remain in the prestage buffer until they are referenced for the last time. When a cache line from the prestage buffer is referenced, it is **not** transferred to the first level I-cache, as other previously proposed prefetching mechanisms do.

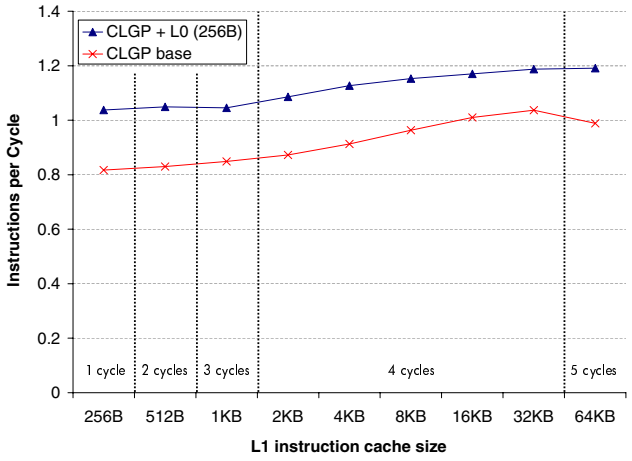
The motivation behind this is the following: in an ideal situation in which we could have a perfect branch predictor, the latency of storage devices would always be masked. Prefetches could be initiated sufficiently early to be effective and all prefetched cache lines would be useful. However, branches are often mispredicted, and prefetches from speculative wrong mispredicted paths have to be discarded. Our idea is that the prestage buffer holds lines prefetched along the path predicted by the branch predictor; if there is a branch misprediction, the cache hierarchy provides finally the cache line requested, which is stored in the lower I-cache level. Accordingly, the prestage buffer stores those cache lines corresponding to highly-executed correct paths, while the L1 I-cache stores those cache lines from few-executed correct paths, after difficult-to-predict branches. In this sense, our L1 cache acts as an *emergency cache*, providing instructions when a branch is mispredicted.

3.2.4. CLGP including an L0 cache. Figure 4 shows CLGP scheme with an L0 cache, just like we did in Section 3.1.1 with FDP. Prefetch requests are now directed to the L1 I-cache. At the fetch stage, the prestage buffer, L0, and L1 caches are accessed in parallel searching for the requested instruction cache line.

3 The consumers counter has some similarities with the CCT used in [17]. Although both keep track of how many cache accesses are to specific cache lines, both are used for different purposes. CCT is necessary to assure consistency in cache accesses, while consumers counter gives CLGP a hint on what will be the last use of a line.



(a) CLGP scheme with an L0 cache



(b) Performance results for a 0.045μm process

Figure 4. CLGP with and without an L0 cache.

Results from Figure 4(b) show that adding an L0 cache to CLGP improves its efficiency. The L0 cache is now the *emergency cache*, and provides instructions from mispredicted correct paths with a one-cycle delay. The main advantage of CLGP compared to FDP is that there is no replication between the contents of prestage buffer and L0 cache, because cache lines replaced from prestage buffer are not transferred to L0 cache. Note that this involves that the amount of cache lines accessible with a one-cycle delay is higher in CLGP than in FDP.

4. Simulation environment

The results presented in this paper were obtained using traces from all the SPECint2000 benchmarks compiled on a DEC Alpha AXP-21264. Due to the large simulation time of SPEC2000, we have collected traces of the most representative 300 million instruction slices following the idea presented in [18].

We have used a custom trace-driven simulator for obtaining our performance results. We permit execution along wrong paths by having a separate basic block dictionary in which we have the information of all static instructions (type, source/target registers). That allows for prefetching even along wrong paths, as well as performing speculative lookups and updates of the branch predictor.

Fetch/Issue/Commit	4 instructions
RUU Size	64 instructions
Branch Predictor	1K+6K -entry stream pred., 1 cycle lat.
RAS	8-entry
Pipeline depth	15 stages
L1 I-Cache	2-way asc., 1 port, 64B/line
L1 D-Cache	32KB, 2-way, 1-cyc lat, 2 ports, 64B/line
L2 Cache	1MB, 2-way asc., 1 port, 128B/line
Mem. lat.	200 cycles
L2 bus BW	64B/cycle
Pre. Buffer / L0 cache	64B/line

Table 2. Simulation parameters.

The baseline processor configuration is shown in Table 2. The pipeline configuration corresponds to a 4-instruction width, 15-stage processor. Although it is well known that the technological evolution will guide to longer pipelines, we have fixed its length in order to be able to compare the performance results and not to introduce a new factor of complexity. Also, we have fixed other parameters (branch predictor, data cache, and memory latency) to obtain results that are independent of changes in their configuration.

The queue that decouples prediction and fetch stages (FTQ in Fetch Directed Prefetching; CLTQ in Cache Line Guided Prestaging) can hold up to 8 fetch blocks. Note that with FDP, each fetch block corresponds to a single entry in the FTQ. However, with CLGP, each fetch block is divided into several fetch cache lines, each one assigned to a different CLTQ entry. Although CLTQ has more entries than FTQ, both queues have the same fetch blocks stored in them, i. e. both techniques have the same opportunities to initiate new prefetches.

Any branch predictor can be used in a decoupled fetch to generate fetch blocks to feed the I-cache [16]. We use the stream predictor [14] in our simulations. It is important to highlight that the same branch predictor will be used in all the simulations for all the configurations.

4.1. Memory configuration

The memory configuration is dependent on the technological factor employed. Along the paper, we use two different configurations depending on the processor generation selected: a current 0.09μm process (found in the recent new Intel Pentium 4 [3] or in the near future Transmeta Efficeon [5]), and a far future 0.045μm process.

To model cache access time, we have used CACTI 3.0 [19], an analytical delay model that evaluates in detail, among other features, the delay of data and

		Cache Size (Bytes)									
		256	**512**	**1K**	**2K**	**4K**	**8K**	**16K**	**32K**	**64K**	**1MB**
Tech.	**0.09 μm**	1 cycle	1 cycle	2 cycles	2 cycles	3 cycles	3 cycles	3 cycles	3 cycles	3 cycles	17 cycles
Process	**0.045 μm**	1 cycle	2 cycles	3 cycles	4 cycles	4 cycles	4 cycles	4 cycles	4 cycles	5 cycles	24 cycles

Table 3. L1 I-cache and L2 cache latencies for different sizes depending on the technology process.

tag paths in cache-like structures. This tool gives us a measurement of the time needed to access a cache for a determined configuration: cache size, line size, associativity, technological process and number of ports. According to the access time provided by CACTI, and the predictions of cycle times done by the SIA for the next microprocessor generations [20], we can calculate I-cache access time for our simulations. Table 3 shows cache latencies for each feature size that will be used along the paper, both for L1 I-cache and L2 cache sizes.

Bus Arbitration Policy. We have modeled a bus to the L2 cache that can only serve one request per cycle, so a bus arbitration policy is needed. The priority policy is the following: the most priority requests are those corresponding to the L1 data cache; then, requests from the L1 I-cache are served; finally, requests from the prefetching mechanism are attended only if no previous request that use the bus is done in the same cycle.

5. Experimental results

In this section we present the performance results of our CLGP compared to a baseline architecture without instruction prefetching and compared to Fetch Directed Prefetching. FDP is known to outperform previously proposed schemes described in the related work section [16], so the fair comparison is against the best known branch predictor guided prefetching scheme.

We first present the IPC performance results obtained. Then, we analyse the fetch and prefetch source distribution that justifies the performance results obtained.

5.1. IPC performance

Figure 5 shows performance results. In each graph, the performance of three different prefetching schemes is plotted: the baseline without prefetching, using Fetch Directed Prefetching, and using Cache Line Guided Prestaging. We will first focus on the results obtained by these configurations with an L0 cache (`base+L0`, `FDP+L0` and `CLGP+L0`), which, as we demonstrated in Section 5, improves the base performance.

Both fully-associative pre-buffers [4] and fully associative L0 caches should have a one-cycle hit time to be effective. Using CACTI 3.0 [19], we have determined pre-buffers and L0 cache sizes that could be accessed in one cycle: 512 bytes at 0.09μm and 256 bytes at 0.045μm.

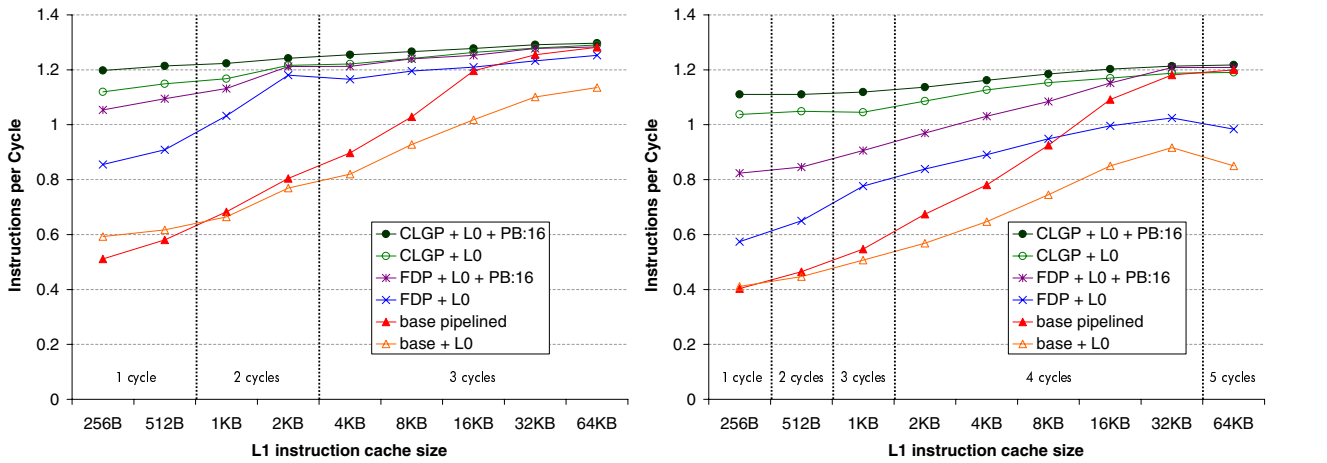
As shows Figure 5, CLGP outperforms both FDP and no prefetching. Although adding an L0 cache solves partly the problems of FDP with long latency caches, CLGP obtains better performance. A better prestage buffer management allows CLGP to give cache response in one cycle most of the time. Using a 4 KB L1 I-cache, CLGP+L0 achieves speedups of 4.8% at 0.09μm and 26% at 0.045μm over FDP+L0. Over the baseline with no prefetching and with an L0 cache (base+L0), CLGP+L0 achieves a 48% speedup at 0.09μm and a 74% at 0.045μm.

Of course, an important part of the speedup comes from the prefetching mechanism itself, and that is why both CLGP and FDP outperform the baseline without prefetching. However, CLGP tries to keep into the fast prestage buffers those instruction lines that will be referenced again in the near future. Meanwhile, CLGP keeps into the lower level I-cache (i. e., *emergency cache*) cache lines from mispredicted paths, which are different from those stored in prestage buffers. This is the main reason why CLGP outperforms FDP: As instructions replaced from the prestage buffers are not moved to the lower level cache, there is no replication between prestage buffer and emergency cache contents, and, therefore, more instructions are accesible at one-cycle distance.

Cache Pipelining. We have also considered cache pipelining as an alternative to using an L0 cache for tolerating cache latency. Pipelining does not reduce hit time, but increases cache delivery throughput. In Figure 5, results are provided for the baseline without prefetching and with a pipelined L1 I-cache (`base pipelined`). Following the same reasoning, we evaluate pipelining pre-buffers. A 16-entry pre-buffer can be used without affecting cycle time, by pipelining its access. We evaluate both FDP and CLGP using a 16-entry pre-buffer, which is pipelined into two stages at 0.09μm and into three stages at 0.045μm.

Without prefetching, using a pipelined cache increases performance as its size is increased, because the branch miss penalty has less impact in the performance than the increasing cache latency. Both FDP and CLGP benefit from having a large pipelined pre-buffer, especially with small L1 I-caches. The speedup of CLGP over FDP is more noticeable at 0.045μm, in which increasing pre-buffer from 4 to 16 entries provides an important speedup.

4 In this section, we will use the term "*pre-buffer*" when both the terms *prefetch buffer* and *prestage buffer* are applicable.



(a) 0.09μm, 8-entry pre-buffer **(b) 0.045μm, 4-entry pre-buffer**

Figure 5. Performance results for the baseline with no prefetching, FDP and CLGP. Latencies for each L1 I-cache size are shown above the X axis.

Using a 4 KB L1 I-cache and pipelined pre-buffers, CLGP obtains speedups of 3.5% at 0.09μm and 12.5% at 0.045μm over FDP. Over the baseline without prefetching but with a 4 KB pipelined I-cache, CLGP obtains a 39% speedup at 0.09μm and a 48% at 0.045μm.

An important observation from these results is that CLGP almost saturates its performance at very small L1 cache sizes. As CLGP tries to fetch mostly from prestage buffers, it is almost insensible to L1 cache size and latency. This is a very important data compared to using a large pipelined cache. Implementing a pipelined cache involves an important area and energy overhead, due to the extra latches, multiplexers, decoders, and sense amplifiers. A conventional and small instruction cache can be used with CLGP without increasing area and energy, and with a better performance.

As an example, CLGP with an L0 cache, a 16-entry pipelined prefetch buffer and a 1 KB L1 I-cache at 0.09μm (in total, 2.5 KB of hardware budget) obtains better performance than using a 16 KB pipelined L1 I-cache without prefetching, that is, 6.4X our hardware budget.

Figure 6 presents IPC results for each SPECint2000 benchmark using a 8 KB L1 I-cache at 0.045 μm. The baseline has an L1 I-cache pipelined into four stages. Both FDP and CLGP use an L0 cache (256 bytes) and 16-entry pre-buffers pipelined into three stages. The results show that CLGP is the best technique for all benchmarks except *gzip*, with speedups over FDP that reach 9% for *gap*, 11% for *vortex*, and 20% for *eon*.

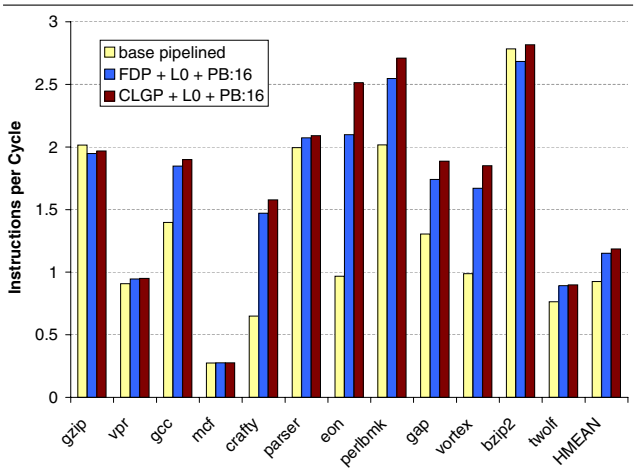
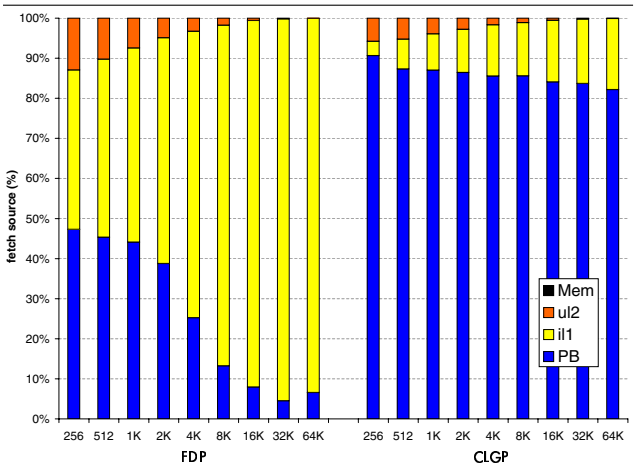


Figure 6. Performance results per benchmark for the best configuration of the baseline, FDP, and CLGP (8KB L1 I-cache; 0.045μm).

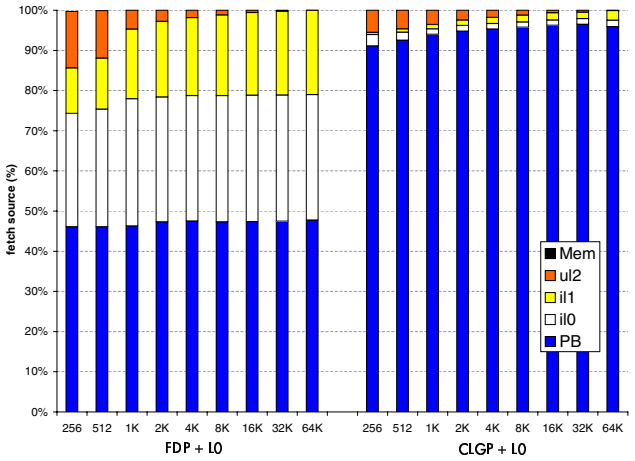
5.2. Fetch source distribution

A key data in our evaluation process is how often prestage buffers are used. It becomes critical to maximise their use, because they are the fastest way to get instructions. Figure 7(a) shows the percentage of instruction fetches provided by each storage element for FDP and CLGP using various L1 cache sizes with and without using an L0 cache.

CLGP is the technique that better exploits the use of pre-buffers, hence the good performance results shown before. FDP decreases the use of pre-buffers as the I-cache size grows, because the percentage of fetches provided by the slow I-cache increases (no prefetch is done if the line is already in L1 or L0), thereby increasing the overall fetch latency. With a 32 KB I-cache, more than a 94% of the FDP fetches comes from L1. This implies that at small technological processes of 0.09μm or 0.045μm the high hit penalty



(a) 4-entry pre-buffer



(b) 4-entry pre-buffer + L0 cache

Figure 7. Distribution of fetch source using different L1 I-cache sizes (bytes), at 0.045μm. `PB` $\equiv$ Pre-Buffer; `il0/il1` $\equiv$ L0/L1 I-cache; `ul2` $\equiv$ Unified L2 cache; `Mem` $\equiv$ Main memory.

of such a large cache limits the performance of FDP.

On the contrary, CLGP minimises L1 I-cache accesses, even at large sizes. The percentage of fetches that are served by the 4-entry pre-buffer is always over 86%, which allows for avoiding the I-cache access penalty most of the time.

Figure 7(b) shows the distribution of fetch sources when an L0 cache is included in both FDP and CLGP. For FDP, around 79% of fetches comes from sources with a one-cycle latency (prefetch buffer or L0 cache), which explains the speedup of FDP+L0 versus FDP. Even without an L0 cache, CLGP has less accesses to L1 I-cache than FDP+L0, which explains the speedup of CLGP versus FDP+L0. Moreover, CLGP also takes benefit from the inclusion of an L0 cache. As prefetches come now from the L1 cache instead of from the L2 cache, more requested prefetches will arrive on time

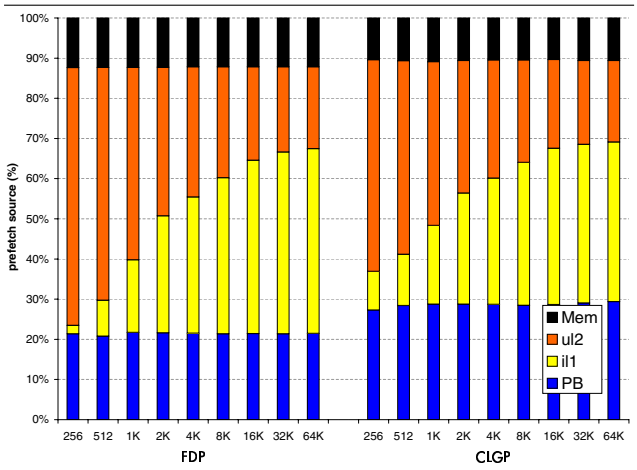


Figure 8. Distribution of prefetch source for different L1 I-cache sizes (bytes), at 0.045μm.

and be stored in the prestage buffer when they are needed. With this new configuration, around 95% of all fetches in CLGP come from either prestage buffer or L0 cache.

The conclusion is that using CLGP, the L1 size is not a critical factor in order to obtain high performance. Even with a tiny L1 cache, combined with a tiny prefetch buffer and—not necessarily required—a tiny L0 cache, CLGP is able to outperform the rest of techniques. This implies that the total cache budget required to obtain a required performance is low, which implies less area, power, and energy.

5.3. Prefetch source distribution

To measure the impact of cache latencies on the prefetching algorithm, Figure 8 shows the prefetch source distribution for FDP and CLGP. With the term prefetch source, we mean the original location of cache lines when a prefetching request is initiated.

The benefits of CLGP are demonstrated by the number of prefetching requests that are already in the prestage buffer. If a cache line is in the prestage buffer, then no prefetch is done, and no cache latency is suffered due to a transfer from L1/L2 cache. On average, FDP has a pre-buffer hit in 21.5% of the prefetch request, while CLGP increases prefetching saves to up to 28%. Furthermore, CLGP performs less prefetches to L2 than FDP. This is especially true with small L1 caches, in which CLGP benefits from a better management of the prestage buffer contents to keep useful lines at one-cycle distance. On average, FDP gets cache lines from L2 cache 37% of the prefetching request, and 12.5% from main memory; CLGP performs 32% of prefetches from L2, and 10.5% from main memory.

As CLGP has a better pre-buffer management and does not replicate its contents in other caches, it allows for keeping more instructions at cache levels closer to the CPU, and

so accesses to higher memory levels decrease.

6. Conclusions

As CPU clock and wire delays increase, the existing gap CPU–memory increases more and more. This is leading designers to implement smaller caches that can be accessed in one cycle, or to maintain common cache sizes with a multi-cycle access (commonly pipelined to increase instruction throughput). In this scenario, we have detected that current I-cache designs are far from their top performance.

In this paper we propose fetch prestaging, a step further of the classic instruction prefetching. Our prestaging technique, *Cache Line Guided Prestaging* (CLGP), tries to maximise the use of prestage buffers in order to alleviate the high cost of accessing the I-cache: high miss rate on small caches and high hit time on large multi-cycle caches. We have observed that using CLGP, around a 88% (95% with an L0 cache) of fetches are provided by the prestage buffer. Meanwhile, the I-cache behaves like an *emergency cache* that holds cache lines useful when there is a prefetch buffer miss, mainly in the event of branch mispredictions.

We have also evaluated additional techniques to overcome cache latencies problems, such as including an L0 cache or using a pipelined cache. Following the same reasoning, we implement pipelined prestage buffers along with an L0 cache, which increases even more CLGP performance. CLGP with an L0 cache is able to provide instructions from one-cycle sources 95% of the time. The best CLGP configuration achieves speedups of 39% at 0.09μm (3-cycle latency L1 cache) and 48% at 0.045μm (4-cycle latency L1 cache) over the baseline using a pipelined cache. It also achieves speedups of 3.5% at 0.09μm and 12.5% at 0.045μm over the best equivalent FDP configuration.

In the longer term, further technological advances will lead to even higher memory latencies. Future processor designs will benefit from the inclusion of mechanisms—like prestaging—that not only adapts its working to the particular memory configuration, but that also apply smarter algorithms to avoid unnecessary memory accesses.

Acknowledgments

We thank the anonymous reviewers for their many suggestions for improving this paper. We especially thank Prof. Brad Calder for his help, ideas, and for suggesting the term *prestaging* for our mechanism.

This work was supported by the Ministry of Science and Technology of Spain under contracts TIC–2001–0995–C02–01 and TIN–2004–07739–C02–01, and by the HiPEAC European Network of Excellence.

References

[1] A. Agarwal, K. Roy, and T. Vijaykumar. Exploring high bandwidth pipelined cache architecture for scaled technology. In *Design, Automation and Test in Europe*, Mar. 2003.

[2] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Procs. of ISCA-27*, June 2000.

[3] D. Boggs, A. Baktha, J. Hawkins, D. Marr, J. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman. The microarchitecture of the Intel Pentium 4 processor on 90nm technology. *Intel Technology Journal*, 08(01), Feb. 2004.

[4] I. K. Chen, C.-C. Lee, and T. Mudge. Instruction prefetching using branch prediction information. In *Intl. Conf. on Computer Design*, Oct. 1997.

[5] D. Ditzel. Transmeta's low power Efficeon microprocessor. Invited Presentation at COOL Chips VII, Apr. 2004.

[6] K. Farkas, N. Jouppi, and P. Chow. How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors. In *Procs. of HPCA-1*, Jan. 1995.

[7] R. Ho, K. Mai, and M. Horowitz. The future of wires. In *Procs. of the IEEE*, Apr. 2001.

[8] D. Jiménez, S. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *MICRO-33*, Dec. 2000.

[9] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Procs. of ISCA-24*, June 1997.

[10] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Procs. of ISCA-17*, June 1990.

[11] J. Kin, M. Gupta, and W. Mangione-Smith. The filter cache: An energy efficient memory structure. In *MICRO-30*, 1997.

[12] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, Sept. 1997.

[13] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *Procs. of MICRO-29*, Dec. 1996.

[14] A. Ramirez, O. Santana, J. Larriba-Pey, and M. Valero. Fetching instruction streams. In *Procs. of MICRO-36*, 2002.

[15] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. In *ISCA-26*, 1999.

[16] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *Procs. of MICRO-32*, Nov. 1999.

[17] G. Reinman, B. Calder, and T. Austin. High performance and energy efficient serial prefetch architecture. In *Procs. of ISHPC-IV*, May 2002.

[18] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Procs. of PACT*, Sept. 2001.

[19] P. Shivakumar and N. Jouppi. CACTI 3.0, an integrated cache timing, power and area model. TR 2001/2, Compaq WRL, Aug. 2001.

[20] Semiconductor Industry Association. The international technology roadmap for semiconductors, 2001.

[21] A. Smith. Cache memories. *ACM Computer Surveys*, 14(3):473–530, Sept. 1982.

[22] J. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Procs. of Supercomputing*, Nov. 1992.

[23] V. Srinivasan, E. Davidson, G. Tyson, M. Charney, and T. Puzak. Branch history guided instruction prefetching. In *Procs. of HPCA-7*, Jan. 2001.

[24] A. Veidenbaum, Q. Zhao, and A. Shameer. Non-sequential instruction cache prefetching for multiple-issue processors. *Intl. Journal of High Speed Computing*, 10(1), Mar. 1999.