Whitepaper:

Hardware Prefetch Controls for Intel® Atom® Cores

Document No: 357930-001US
December 2023

Notices & Disclaimers

Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Intel technologies may require enabled hardware, software, or service activation.

Revision History

Revision	Description	Date
357930-001US	Initial Release.	December 2023

CONTENTS

1 INTRODUCTION

Hardware prefetchers are an excellent way of improving performance by fetching information ahead of time. The most basic prefetchers that only fetched the next cache line were introduced in the early days of processors. Today, prefetchers can track complex patterns to get the most relevant data into the caches ahead of usage. However, prefetching in multicore systems is a complex task; it can improve performance significantly, but driving more data into the caches can also evict more relevant data. Furthermore, prefetchers drive up the DDR bandwidth, which can become a bottleneck and lower overall system performance. By tuning the prefetchers, one can ensure optimal performance at any given time; for instance, Figure 1 shows SpecInt GCC performance on a Raptor Lake microarchitecture with default settings and a relatively basic dynamic prefetch tuning, see [1].

This document describes how the hardware prefetchers are used in the Intel® Atom® Efficient-cores (or E-cores) and how they can be tuned to improve the core and memory-related performance. It is relevant for the Intel Atom E-cores, starting with the Alder and Raptor Lake microarchitectures, and server CPUs based on those Intel Atom E-cores, such as those with Grand Ridge and Sierra Forest microarchitectures.

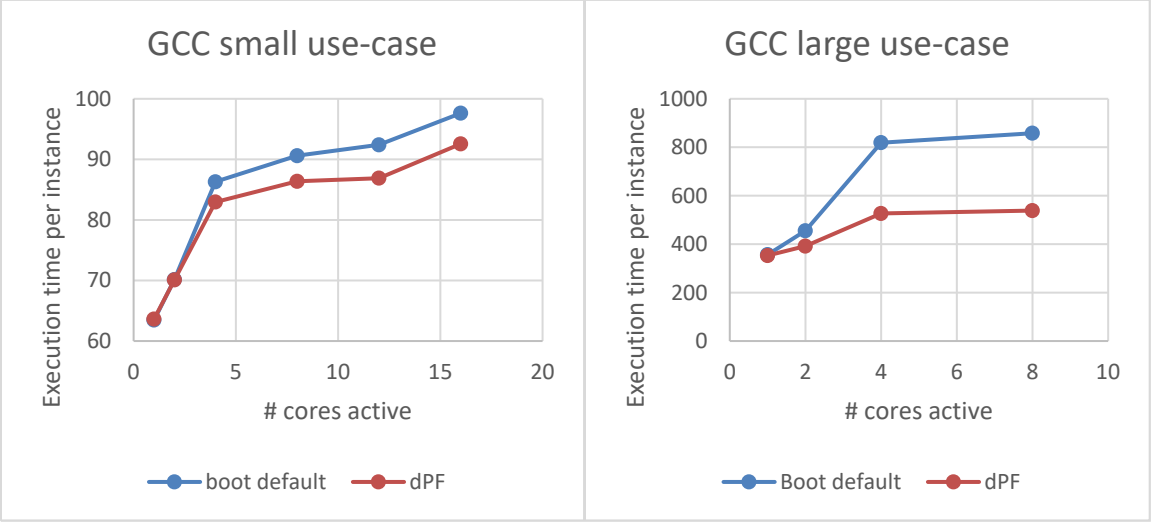

Figure 1: Performance gain with rudimentary dynamic prefetch tuning (dPF) vs. default static boot settings.

1.1 HARDWARE PREFETCH OVERVIEW

The Intel Atom cores are placed in a group of four per module with private L1 caches for each core. The banked L2 mid-level cache (MLC) is shared (see Figure 2). Each module is then attached to the CPU interconnect, where the centrally shared L3 cache is located. The interface between the cores and the L2 cache is called the L2 Queue (L2Q). Any L2/MLC cache misses continue through the External Queue (XQ) into the Bus Interface Unit (BIU) and onto the Uncore interconnect towards the L3 Last-level caches (LLC). Transactions that miss in the L3 will continue to the DDR.

Each core has a set of L1 hardware prefetchers that can be individually enabled/disabled. These feed prefetches into the L2Q and the normal load/store requests.

The L2 prefetch block is shared for all cores in the module and contains several trackers. Each tracker searches for and then locks on to a flow of either instruction or data. These trackers are shared between the cores but only serve one core at a time. Multiple prefetch functions within the block work together to form a unity. All prefetch requests are based on one or multiple cache lines, with one cache line being 64 bytes of data. The generated requests are injected into the L2Q, similarly to the core requests. Data is then fetched to either the L2/MLC or the L3/LLC cache. The L2 prefetcher monitors data requests coming from a load instruction, referred to as demand reads, through the queues in the module's L2 pipeline. This is to identify access patterns as a basis for onward predictions. These predictions are combined with data such as queue depth, throttling indications, etc., to generate prefetches and ensure they work optimally.

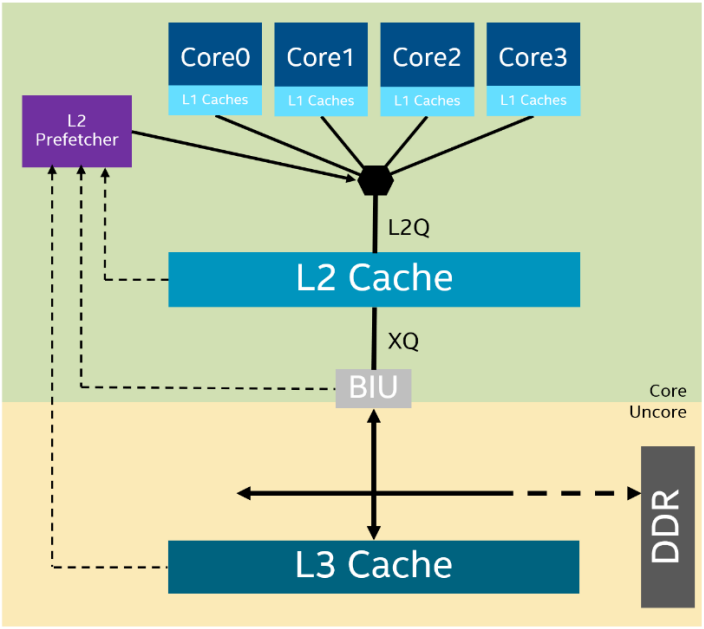

Figure 2: Intel® Atom® Core Module and Interface to Uncore

The overall target is to prefetch data that will be used relatively soon, decreasing the latency to access the data. Fetching data as close to the core as possible is preferred to minimize latency, i.e., L1, then L2, and L3. In the worst case, one must wait for the DDR, which has the longest latency and a more limited bandwidth. However, prefetching data too soon risks evicting data from the cache still in use, which is counterproductive. The prefetcher will lock on access patterns that are only repeated for some time, typically a loop in the software or a block of loaded new data. Hence, the access pattern will stop sooner or later, and the prefetchers will have requested data that won't be used. Some wasteful data fetches are generally an acceptable tradeoff to reduce latency. However, the degree of wasteful fetches is highly application-dependent; for example, many small loops with random data accesses vs. a few lengthy loops of highly repetitive fetches. The tolerance for such waste is also very system-dependent. A system with bottlenecks in DDR bandwidth has nothing to gain from prefetches that add further load on DDR. However, the latency to DDR is a function of the DDR bandwidth used with a knee towards the end. Systems that operate with a low or medium level of DDR load can benefit significantly from more aggressive prefetching as long as the DDR loads stay below this knee.

The balance of when and where to prefetch can be tricky, especially since the application and system behavior often change over time. This is where the L1 and L2 Prefetch tuning capability comes in, allowing for per-core and per-module configuration, both static and dynamic, over time. These can be combined with Performance Monitor Unit (PMU) events and other statistics counters to identify load on the DDR interfaces, load/stores executed by each core, cache hit/miss ratio, and rate of prefetches that are subsequently used.

1.2 PREFETCHERS

The Atom core generation up to Gracemont, i.e., Alder Lake and Raptor Lake platforms, includes the L1 prefetchers listed below. These L1 prefetchers can enable/disable through MSRs but not further tunable through software. They are integrated into each core and only consider per-core aspects. The generated prefetch requests are sent from the respective core into the L2Q, similar to normal load/stores.

1.2.1 Instruction Point Prefetcher (L1 IPP)

The IPP is commonly listed as the DCU_IP_PREFETCH[1] in Basic Input/Output System (BIOS).

The IPP Keeps track of individual load instructions. When a load instruction is detected to have a regular stride, a prefetch is sent to the next address, which is the sum of the current address and the stride.

[1] DCU – Data Caching Unit is the block that holds the L1 data cache.

1.2.2 Next Line Prefetcher (L1 NLP)

The NLP is commonly also listed as the DCU_STREAMER_PREFETCH in BIOS.

Prefetches next line of any given access, applicable to instructions and data. The NLP is one of the earliest types of prefetchers introduced at a time well before multi-core was the norm. It was a blunt approach: simple to implement, worked well for single-core systems, but drove much bandwidth. This impacted both DDR bandwidth and L1/L2 caches and queues. Use of the NLP should be done with care, and a general recommendation is to run with the NLP disabled unless you can observe a benefit to your system.

1.2.3 Next Page Prefetcher (L1 NPP)

The NPP is commonly also listed as the DCU_NEXT_PAGE_PREFETCH in BIOS.

Tracks access during streaming and prefetches the next TLB entry, both applicable to instructions and data. The NPP is lightweight and reduces page-walk costs, which is especially important for applications that touch a lot of new data.

1.3 L2 PREFETCHERS

Three types of prefetchers within the L2 prefetch block are jointly structured into a set of trackers. These are named since they make decisions by tracking requests from the core, as normal demand reads and L1 prefetches. The number of trackers varies between implementations, but there are typically multiple dozens of trackers per module. Each tracker searches for access patterns, locks on to them, and starts proactively fetching ahead. If a pattern breaks for some time, the tracker stops prefetching and returns to search for a new pattern. The different prefetcher types work together and share the same queue of requests that feeds into the L2Q.

The type of requests used to train and trigger the L2 prefetcher is one of the configurable parameters; see Section 4.5. The L2 prefetchers are highly configurable. However, additional mechanisms not covered in this paper ensure fairness, overload protection, and timely relevance in the requests.

A central feedback parameter is the depth of the XQ. The XQ threshold level for when to issue prefetches is also a valuable configuration parameter present in multiple MSRs. A higher number of entries in that queue states that more loads go beyond the L2 cache; hence, the L3 cache will see a higher load. If these miss in the L3 cache, they will likely generate a significant DDR load, for which it is generally a good practice to reduce the number of prefetches issued. Requests towards MMIO address space (such as PCIe) will also be routed through the XQ but are seldom a significant load.

1.3.1 MLC Streamer

MLC Streamer is commonly listed as the MLC_STREAMER_PREFETCH in BIOS and sometimes referred to as simply the Stream Prefetcher or L2 Streamer.

The MLC Streamer works on both instructions and data; it tracks access patterns such as A+0, A+1, A+2, … where A is the linear address of a cache line. Patterns can be either incremental or decremental. Once a pattern is locked, it will fetch subsequent accesses into the L2 cache. The number of requests ahead of demand reads issued is configurable; see Section 4.1.

1.3.2 LLC Streamer

The LLC Streamer is sometimes called the LLC Prefetcher or L3 Prefetcher.

The LCC Streamer behaves like the MLC Streamer but fetches into the LLC. The intention is that the LLC Streamer should issue requests further ahead, i.e., further out in time. The larger LLC size can be leveraged to be more speculative and balance the risk of miss-fetches vs. potential benefits. The MLC prefetcher carries the data closer to the core once we have more certainty that the data will be used, i.e., when the distance to current demand reads is closer.

1.3.3 Adaptive Multi-Path (AMP)

The AMP prefetcher builds on top of the MLC Streamer but can fetch more complex patterns such as A+0, A+8, A+0+N, A+8+N, A+0+2N, … where N is a distance ahead/behind. The AMP prefetcher generally has more precision and, therefore, higher priority than the Streamers. In a scenario where prefetching should be limited, it can be valuable to disable or limit the streamers and let the AMP stay on.

The confidence level of AMP can be set for different throttling levels, where the latter is a feedback parameter taken from the uncore, see Section 0.

1.3.4 L2 Next Line Prefetch (L2 NLP)

The L2 Next Line Prefetcher (L2 NLP) is part of the MLC Streamer block and behaves similarly to the L1 NLP except that it fetches to the L2 cache. This commonly generates a significant increase in prefetches, which can be of value to certain workloads and systems where single-core performance is important. However, this increased request load is generally negative for multicore systems, and the L2 NLP is, by default, turned off.

2 Tunable parameters

The tunable parameters vary between prefetch types and apply primarily to the MLC or LLC Streamer and AMP. These three blocks share some common structures; hence, the tuning parameters are slightly intertwined.

2.1 ENABLE / DISABLE

Each prefetcher's "big hammer" is the enable/disable bit; they are spread out over MSR 0x1A4 and MSR 0x1320/21. Note that 0x1A4 is also used by the P-cores but with slightly different definitions. For instance, the L2 Spatial Prefetcher is unavailable on Intel® Atom® cores.

Prefetcher	MSR	bit
MLC/LLC Streamer disable	0x1A4	0
DCU Streamer (L1 NLP)	0x1A4	2
DCU IP Prefetcher (L1 IPP)	0x1A4	3
DCU Next Page Prefetcher (L1 NLP)	0x1A4	4
AMP	0x1A4	5
LLC Streamer	0x1320	43
L2 NLP	0x1321	40

2.2 AMP AND MLC/LLC STREAMER SOURCE SELECTION

The source selection for AMP and the MLC/LLC Streamer can be found in MSR 0x1323; see Section 4.5. There are two types of switches: those that enable training of the prefetcher from the respective source and those that enable tracking the prefetcher from the respective source. Both should be enabled for the respective source that is of interest.

The types of requests that can be selected include the various hardware L1 prefetches, different software prefetches generated by the code, and different types of reads. Reads, also known as demand reads, are normal load operations and can co-exist in multiple caches. Read for Ownership (RFO) are reads where the core intends to write a value and requests exclusive ownership of the cache line.

2.3 STREAMER CONFIGURATION

There are three key parameters for the MLC and LLC Streamer, respectively.

1. The maximum distance ahead of current generated requests, i.e., where the core is currently executing; see Section 4.2. A higher value will drive more prefetches and hence increase the chance of hitting in the respective cache layer, but this can also evict relevant data that would be better to keep in the caches. A longer distance will also drive more DDR bandwidth.
2. The XQ threshold; see Section 4.2 for MLC Streamer.
   a. Note that this setting is also shared with AMP. For more information about LLC Streamer, see Section 0. The XQ queue handles all requests that are misses in the L2 cache.
   b. The value set specifies how many empty entries the XQ must have to accept new prefetch requests. A high threshold value will generally lower the prefetch aggressiveness.
3. The third is the *demand density* parameters; see Section 4.3 for the MLC Streamer and Section 0 for the LLC Streamer. The name refers to demand reads (and the number of prefetches upgraded to demand reads) from the core over an internally pre-defined period of time.
   a. A high-demand density indicates that the core is using the prefetches.
   b. Demand density is measured as an average for all trackers assigned to a given core in the module. A value lower than the configuration will hold prefetch requests from all the MLC respective LLC Streamer trackers. This is because the overall accuracy in prefetches is deemed too low.

c. The trackers will continue to follow the data flows and update the demand density. However, individual trackers that reach a higher level of demand density, specified by the demand density override, are deemed accurate enough and will, therefore, generate prefetch requests.

Finally, two configurations connect the demand density and XQ Threshold. If the demand density is lower than the L2_LLC_STREAM_DEMAND_DENSITY_XQ setting, then the XQ Thresholds are set per the L2_LLC_STREAM_AMP_XQ_THRESHOLD. See Sections 4.3 and 0.

2.4 AMP CONFIGURATION

The AMP prefetcher shares the XQ threshold with the MLC Prefetcher but has no maximum distance setting. However, it has two additional settings:

1. AMP has a tunable confidence level required to trigger a request; see Section 0.
   a. This parameter is set for different throttling levels. The throttling level is an unconfigurable uncore feedback parameter for the AMP block. Setting a higher confidence level will make AMP more selective in making requests, lowering the overall number of requests generated.
   b. Note that this can drive more MLC/LLC Streamer requests as AMP has higher priority. Only one prefetcher issues requests per timeslot. Therefore, increasing confidence does not necessarily lower the total number of prefetch requests unless Streamer parameters are also tuned down.
2. An enable/disable recursive lookup pattern specifies how AMP should trigger on patterns; see Section 4.2. Disabling recursive lookup will lower the number of prefetch requests. It is strongly recommended to leave this disabled.

3 PERFORMANCE MONITORS

Performance monitoring events in the Intel® Atom® core (the Gracemont microarchitecture) and Intel® Core™ (Alder/Raptor Lake microarchitectures) are relevant to hardware prefetch tuning. See the respective device manual for details. Performance events are also available at perfmon-events.intel.com. Some generic suggestions for how these can be used follow.

3.1 IMC RD CAS

The Integrated Memory Controller (IMC) on client platforms typically has two static counters tracking the number of read and write requests. On server platforms, there are generally configurable performance monitoring counters instead. These counters measure requests at the cache line level of granularity, so the value should be multiplied by 64 to get the number of bytes transferred. The static counters start counting on power-on, and the configurable counters start counting when the start signal is provided. In either case, the amount of DDR requests generated is derived by taking the delta values between two points in time.

The read counter is of primary interest for hardware prefetch tuning as hardware prefetchers drive additional reads, although dirty cache lines can also be pushed out as writes. These counters are shared resources at the platform level, not tied to individual cores or modules. Note that there are generally multiple IMCs per device. The usage of DDR interleaving, which is the default configuration, gives a near-equal load on the interfaces.

Setting prefetch aggressiveness based on DDR load is a simple and efficient method to improve performance.

3.2 MEM_UOPS_RETIRED_ALL_LOADS

This per-core event tracks the number of load operations completed, i.e., not including speculative loads or prefetches not upgraded to demand reads. This value can be used with respective cache layers hit/miss events.

The total number of loads relative to respective core loads can also be used to identify the core driving most loads. However, a core that executes many loads could hit well in caches and does not necessarily contribute to the DDR load. Another parameter to size the DDR pressure per core is LONGEST_LAT_CACHE.MISS or MEM_LOAD_UOPS_RETIRED_DRAM_HIT.

3.3 MEM_LOAD_UOPS_RETIRED_L2_HIT

This event measures the number of load operations concluded where the load hits in the MLC / L2 cache.

3.4 MEM_LOAD_UOPS_RETIRED_L3_HIT

This event measures the number of load operations concluded where the load hits in the LLC / L3 cache.

The L2 hit vs. L3 hit and DDR hit (i.e., L2 miss) can be used to identify the L2 hit ratio. This is similar to the L3 hit ratio to the L3 hit vs. DDR hit. These can be leveraged to estimate how well prefetches to the MLC and LLC work.

3.5 LONGEST_LAT_CACHE.MISS

This event counts the number of cacheable memory requests that miss in the LLC and hence targets DDR or other coherent memory.

3.6 MEM_LOAD_UOPS_RETIRED_DRAM_HIT

This event measures the number of load operations concluded where the load hit in DDR. This can be leveraged to identify how much of the overall DDR bandwidth originates from the respective core. A more fine-grained selection of what cores should be throttled can thus be made.

See also event MEM_BOUND_STALLS.LOAD_DRAM_HIT, which counts the cycle's cost for the penalty of going to DDR.

3.7 XQ_PROMOTION_ALL

This even counts the number of XQ entries initiated as hardware prefetches, but where a normal request was made that upgraded the entry. This indicates that the prefetch was correct and reduced access latency. However, prefetches that completed and brought the data into the respective cache and then were used are not counted.

It is currently impossible in the Intel Atom cores covered in this document to measure the number of prefetches issued. But the promotions relative L2 misses and respective such ratio per core can be an estimate of how successful the prefetching is. This must also account for respective core prefetch-setting.

4 L2 PREFETCH MSR

MSRs are shared within a module, and changes from one core will be seen by the other in that module. Only cores within the module can access the module's configuration. For example, it is enough if one core changes the MSR; use core affinity to ensure you are executing on the relevant module.

The MSRs described below are only applicable to the Intel® Atom® cores. Use the CPUID feature to identify core types, and if a hybrid platform is in use, see [2] for more details. Bits outside of the ones described may be reserved or have other functions, including having other functions on other cores; see the respective device documentation and ensure that values outside these listed are preserved.

4.1 MSR 0X1A4

Width	Bits	Descriptor
1	0:0	MLC Stream disabled; when set to 1, the MLC Streamer is disabled.
1	2:2	L1 data Stream disabled; when set to 1, the L1 data Streamer is disabled.
1	3:3	L1 instruction Stream disabled; when set to 1, the L1 instruction Streamer is disabled.
1	5:5	L2 AMP disabled; when set to 1, the L2 AMP is disabled.

4.2 INTEL® ATOM® MSR 0X1320

Width	Bits	Descriptor
5	4:0	L2 Stream and AMP XQ threshold. The value specifies the number of empty XQ entries to emit prefetches. I.e., a higher value drives fewer prefetches.
5	24:20	L2 Stream max distance from core execution. A higher number will prefetch more data.
1	30:30	AMP recursive lookup disabled; setting to 1 decreases the number of prefetches.
6	42:37	LLC Stream max distance from core execution. A higher number will prefetch more data.
1	43:43	LLC Stream disable; when set to 1, the LLC Streamer is disabled.
5	62:58	LLC Stream XQ threshold. The value specifies the number of empty XQ entries to emit prefetches. For example, a higher value drives fewer prefetches.

```
struct msr1320_s{
        uint64_t L2_STREAM_AMP_XQ_THRESHOLD : 5;
        uint64_t pad0 : 15;
        uint64_t L2_STREAM_MAX_DISTANCE : 5;
        uint64_t pad1 : 5;
        uint64_t L2_AMP_DISABLE_RECURSION : 1;
        uint64_t pad2 : 6;
        uint64_t LLC_STREAM_MAX_DISTANCE : 6;
        uint64_t LLC_STREAM_DISABLE : 1;
        uint64_t pad3 : 14;
        uint64_t LLC_STREAM_XQ_THRESHOLD : 5;
};
```

4.3 INTEL® ATOM® MSR 0X1321

Width	Bits	Descriptor
1	0:0	MLC Streamer and AMP enabled tracking of instruction flows.
8	28:21	MLC Streamer demand density.
4	32:29	MLC Streamer demand density override.
1	40:40	L2 Next Line Prefetcher disabled; when set to 1, the L2 NLP is disabled.
6	46:41	XQ threshold when demand density is low. Typically set higher than normal XQ thresholds. Applicable to MLC/LLC Streamer and AMP.

```
struct msr1321_s{
        uint64_t L2_STREAM_AMP_CREATE_IL1 : 1;
        uint64_t pad0 : 20;
        uint64_t L2_STREAM_DEMAND_DENSITY : 8;
        uint64_t L2_STREAM_DEMAND_DENSITY_OVR : 4;
        uint64_t pad1 : 7;
        uint64_t L2_DISABLE_NEXT_LINE_PREFETCH : 1;
        uint64_t L2_LLC_STREAM_AMP_XQ_THRESHOLD : 6;
};
```

4.4 INTEL® ATOM® MSR 0X1322

Width	Bits	Descriptor
9	22:14	LLC Steamer demand density.
4	26:23	LLC Steamer demand density override.
6	32:27	AMP Confidence threshold at throttle level 0 (no throttling): • Emits prefetches when confidence is above this level. • A higher level gives less aggressive prefetching.
6	38:33	AMP Confidence threshold at throttle level 1 (light throttling): • Emits prefetches when confidence is above this level. • A higher level gives less aggressive prefetching.
6	44:39	AMP Confidence threshold at throttle level 2 (moderate throttling): • Emits prefetches when confidence is above this level. • A higher level gives less aggressive prefetching.
6	50:45	AMP Confidence threshold at throttle level 3 (aggressive throttling): • Emits prefetches when confidence is above this level. • A higher level gives less aggressive prefetching.
3	61:59	Demand density to switch to alternative XQ threshold. (L2_LLC_STREAM_AMP_XQ_THRESHOLD)

```
struct msr1322_s{
        uint64_t pad0 : 14;
        uint64_t LLC_STREAM_DEMAND_DENSITY : 9;
        uint64_t LLC_STREAM_DEMAND_DENSITY_OVR : 4;
        uint64_t L2_AMP_CONFIDENCE_DPT0 : 6;
        uint64_t L2_AMP_CONFIDENCE_DPT1 : 6;
        uint64_t L2_AMP_CONFIDENCE_DPT2 : 6;
        uint64_t L2_AMP_CONFIDENCE_DPT3 : 6;
        uint64_t pad1 : 8;
        uint64_t L2_LLC_STREAM_DEMAND_DENSITY_XQ : 3;
};
```

4.5 INTEL ® ATOM® MSR 0X1323

Width	Bits	Descriptor
1	34:34	Begin tracking software data prefetches with the intent to write. Affects both the MLC Streamer and AMP.
1	35:35	Begin tracking software data prefetches. Affects both the MLC Stream and AMP.
1	37:37	Begin tracking requests coming from the L1 NLP prefetcher. Affects both the MLC Stream and AMP.
1	38:38	Begin tracking demand requests with intent to write. Affects both the MLC Stream and AMP.
1	39:39	Allow SW prefetch read for ownership to train a prefetch streamer entry.
1	40:40	Allow SW prefetch reads to train a prefetch streamer entry.
1	41:41	Allow IL1 to train a prefetch streamer entry.
1	43:43	Allow hardware prefetch read to train a prefetch streamer entry.
1	44:44	Allow read-for-ownership (RFO) requests to train a prefetch streamer entry.
1	45:45	Begin tracking requests coming from the L1 NPP prefetcher. Affects both the MLC Stream and AMP.
1	46:46	Begin tracking requests coming from the L1 IPP prefetcher. Affects both the MLC Stream and AMP.
1	47:47	Allow NPP prefetcher to train L2 prefetchers.
1	48:48	Allow IPP prefetcher to train L2 prefetchers.

```
struct msr1323_s{
        uint64_t pad0 : 34;
        uint64_t L2_STREAM_AMP_CREATE_SWPFRFO : 1;
        uint64_t L2_STREAM_AMP_CREATE_SWPFRD : 1;
        uint64_t pad1 : 1;
        uint64_t L2_STREAM_AMP_CREATE_HWPFD : 1;
        uint64_t L2_STREAM_AMP_CREATE_DRFO : 1;
        uint64_t STABILIZE_PREF_ON_SWPFRFO : 1;
        uint64_t STABILIZE_PREF_ON_SWPFRD : 1;
        uint64_t STABILIZE_PREF_ON_IL1 : 1;
        uint64_t pad2 : 1;
        uint64_t STABILIZE_PREF_ON_HWPFD : 1;
        uint64_t STABILIZE_PREF_ON_DRFO : 1;
        uint64_t L2_STREAM_AMP_CREATE_PFNPP : 1;
        uint64_t L2_STREAM_AMP_CREATE_PFIPP : 1;
        uint64_t STABILIZE_PREF_ON_PFNPP : 1;
        uint64_t STABILIZE_PREF_ON_PFIPP : 1;
};
```

5 REFERENCES

[1] **GitHub**:hardwarePrefetching. 2023. [Online]. Available: https://github.com/uart/hardwarePrefetching

[2] **Intel Architecture Instruction Set Extensions Programming Reference**. [Online]. Available: https://www.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html.