Chapter 4. "Zihintntl" Non-Temporal Locality Hints, Version 1.0

The NTL instructions are HINTs that indicate that the explicit memory accesses of the immediately subsequent instruction (henceforth "target instruction") exhibit poor temporal locality of reference. The NTL instructions do not change architectural state, nor do they alter the architecturally visible effects of the target instruction. Four variants are provided:

The NTL.P1 instruction indicates that the target instruction does not exhibit temporal locality within the capacity of the innermost level of private cache in the memory hierarchy. NTL.P1 is encoded as ADD *x0, x0, x2*.

The NTL.PALL instruction indicates that the target instruction does not exhibit temporal locality within the capacity of any level of private cache in the memory hierarchy. NTL.PALL is encoded as ADD *x0, x0, x3*.

The NTL.S1 instruction indicates that the target instruction does not exhibit temporal locality within the capacity of the innermost level of shared cache in the memory hierarchy. NTL.S1 is encoded as ADD *x0, x0, x4*.

The NTL.ALL instruction indicates that the target instruction does not exhibit temporal locality within the capacity of any level of cache in the memory hierarchy. NTL.ALL is encoded as ADD *x0, x0, x5*.

The NTL instructions can be used to avoid cache pollution when streaming data or traversing large data structures, or to reduce latency in producer-consumer interactions.

A microarchitecture might use the NTL instructions to inform the cache replacement policy, or to decide which cache to allocate into, or to avoid cache allocation altogether. For example, NTL.P1 might indicate that an implementation should not allocate a line in a private L1 cache, but should allocate in L2 (whether private or shared). In another implementation, NTL.P1 might allocate the line in L1, but in the least-recently used state.

NTL.ALL will typically inform implementations not to allocate anywhere in the cache hierarchy. Programmers should use NTL.ALL for accesses that have no exploitable temporal locality.

Like any HINTs, these instructions may be freely ignored. Hence, although they are described in terms of cache-based memory hierarchies, they do not mandate the provision of caches.

Some implementations might respect these HINTs for some memory accesses but not others: e.g., implementations that implement LR/SC by acquiring a cache line in the exclusive state in L1 might ignore NTL instructions on LR and SC, but might respect NTL instructions for AMOs and regular loads and stores.

Table 6 lists several software use cases and the recommended NTL variant that *portable* software—i.e., software not tuned for any specific implementation's memory hierarchy—should use in each case.

Scenario	Recommended NTL variant
Access to a working set between and in size	NTL.P1
Access to a working set between and in size	NTL.PALL
Access to a working set greater than in size	NTL.S1
Access with no exploitable temporal locality (e.g., streaming)	NTL.ALL

Scenario	Recommended NTL variant
Access to a contended synchronization variable	NTL.PALL

Table 6. Recommended NTL variant for portable software to employ in various scenarios.

The working-set sizes listed in Table 6 are not meant to constrain implementers' cache-sizing decisions. Cache sizes will obviously vary between implementations, and so software writers should only take these working-set sizes as rough guidelines.

Table 7 lists several sample memory hierarchies and recommends how each NTL variant maps onto each cache level. The table also recommends which NTL variant that implementation-tuned software should use to avoid allocating in a particular cache level. For example, for a system with a private L1 and a shared L2, it is recommended that NTL.P1 and NTL.PALL indicate that temporal locality cannot be exploited by the L1, and that NTL.S1 and NTL.ALL indicate that temporal locality cannot be exploited by the L2. Furthermore, software tuned for such a system should use NTL.P1 to indicate a lack of temporal locality exploitable by the L1, or should use NTL.ALL indicate a lack of temporal locality exploitable by the L2.

If the C extension is provided, compressed variants of these HINTs are also provided: C.NTL.P1 is encoded as C.ADDi *x0, x2*; C.NTL.PALL is encoded as C.ADD *x0, x3*; C.NTL.S1 is encoded as C.ADD *x0, x4*; and C.NTL.ALL is encoded as C.ADD *x0, x5*.

The NTL instructions affect all memory-access instructions except the cache-management instructions in the Zicbom extension.

As of this writing, there are no other exceptions to this rule, and so the NTL instructions affect all memory-access instructions defined in the base ISAs and the A, F, D, Q, C, and V standard extensions, as well as those defined within the hypervisor extension in Volume II.

The NTL instructions can affect cache-management operations other than those in the Zicbom extension. For example, NTL.PALL followed by CBO.ZERO might indicate that the line should be allocated in L3 and zeroed, but not allocated in L1 or L2.

Memory hierarchy	Recommended mapping of NTL variant to actual cache level				Recommended NTL variant for explicit cache management			
	P1	PALL	S1	ALL	L1	L2	L3	L4/L5
Common Scenarios								
No caches	---				none			
Private L1 only	L1	L1	L1	L1	ALL	---	---	---
Private L1; shared L2	L1	L1	L2	L2	P1	ALL	---	---
Private L1; shared L2/L3	L1	L1	L2	L3	P1	S1	ALL	---
Private L1/L2	L1	L2	L2	L2	P1	ALL	---	---
Private L1/L2; shared L3	L1	L2	L3	L3	P1	PALL	ALL	---
Private L1/L2; shared L3/L4	L1	L2	L3	L4	P1	PALL	S1	ALL
Uncommon Scenarios								
Private L1/L2/L3; shared L4	L1	L3	L4	L4	P1	P1	PALL	ALL
Private L1; shared L2/L3/L4	L1	L1	L2	L4	P1	S1	ALL	ALL
Private L1/L2; shared L3/L4/L5	L1	L2	L3	L5	P1	PALL	S1	ALL
Private L1/L2/L3; shared L4/L5	L1	L3	L4	L5	P1	P1	PALL	ALL

Table 7. Mapping of NTL variants to various memory hierarchies.

When an NTL instruction is applied to a prefetch hint in the Zicbop extension, it indicates that a cache line should be prefetched into a cache that is *outer* from the level specified by the NTL.

For example, in a system with a private L1 and shared L2, NTL.P1 followed by PREFETCH.R might prefetch into L2 with read intent.

To prefetch into the innermost level of cache, do not prefix the prefetch instruction with an NTL instruction.

In some systems, NTL.ALL followed by a prefetch instruction might prefetch into a cache or prefetch buffer internal to a memory controller.

Software is discouraged from following an NTL instruction with an instruction that does not explicitly access memory. Nonadherence to this recommendation might reduce performance but otherwise has no architecturally visible effect.

In the event that a trap is taken on the target instruction, implementations are discouraged from applying the NTL to the first instruction in the trap handler. Instead, implementations are recommended to ignore the HINT in this case.

If an interrupt occurs between the execution of an NTL instruction and its target instruction, execution will normally resume at the target instruction. That the NTL instruction is not reexecuted does not change the semantics of the program.

Some implementations might prefer not to process the NTL instruction until the target instruction is seen (e.g., so that the NTL can be fused with the memory access it modifies). Such implementations might preferentially take the interrupt before the NTL, rather than between the NTL and the memory access.

Since the NTL instructions are encoded as ADDs, they can be used within LR/SC loops without voiding the forward-progress guarantee. But, since using other loads and stores within an LR/SC loop does void the forward-progress guarantee, the only reason to use an

NTL within such a loop is to modify the LR or the SC.