DISCLAIMER:

This document does not meet the
current format guidelines of
the Graduate School at
The University of Texas at Austin.

It has been published for
informational use only.

The Report committee for Douglas Raye Reed
Certifies that this is the approved version of the following report:

Performance and Complexity Tradeoffs in Partially-Inclusive Caches

APPROVED BY

SUPERVISING COMMITTEE:

Supervisor:

Vijay Janapa Reddi

Co-Supervisor:

John Greer

Performance and Complexity Tradeoffs in Partially-Inclusive Caches

by

Douglas Raye Reed, B.S.E.E.

REPORT

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2015

Performance and Complexity Tradeoffs in Partially-Inclusive Caches

Douglas Raye Reed, M.S.E.
The University of Texas at Austin, 2015

Supervisors: Vijay Janapa Reddi
John Greer

Multi-level inclusive cache hierarchies have historically provided a convenient tradeoff between performance and design complexity. However, as the desire for more intermediate levels of caches rises, the shrinking size disparity between adjacent levels of cache exacerbates the wasteful redundancy inherent in inclusive cache designs. Where it is still beneficial to have larger, slower caches act as inclusive caches and snoop filters for smaller, faster caches nearer to the core, those benefits can be undermined by excessive data duplication and frequent back-invalidations when the larger cache is only a factor of two- to four-times the size of the smaller cache.

One technique to address the issues that arise with inclusive caches is *partial inclusivity*. Partially inclusive caches can help address the problem of data duplication in a cache hierarchy, while still providing performance and robust snoop filtering akin to that of a traditional inclusive cache. Moreover, such cache designs can decrease the frequency of back-invalidates caused by strictly inclusive caches. We describe two approaches to implementing a partially inclusive mid-level cache, while exploring the implications of our design

decisions on performance, array size, and implementation complexity. We show that the first approach, ThinL2, allows for simpler coherence record-keeping but dramatically increases snooping of the first-level caches. We also show that the second approach, WideL2, allows for relatively efficient snooping of the first-level caches but incurs much more record-keeping complexity. We then provide ideas for addressing some of the complexity problems associated with WideL2.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the disparity between processor speed and memory speed threatening to cripple system performance, multi-level inclusive cache designs have proliferated as a way for computer architects to address memory performance issues while keeping complexity under control. The landscape of inclusive cache designs has traditionally offered attractive tradeoffs between design complexity and performance, but new programming paradigms, increasing hardware requirements, and relatively slow DRAM innovation continues to apply intractable pressure to conventional design techniques. As performance requirements outpace advances in process technology, the tired wisdom of simply making existing cache structures larger–and therefore slower–is no longer a feasible approach for high-performance computing.

Rather than simply increasing the size of existing caches, a popular approach to providing more physical storage locality is to increase the *number of levels* of cache in a cache hierarchy, instead of (or in addition to) changing the size of existing cache structures. As the desire for more intermediate levels of caches arises, the shrinking size disparity between adjacent levels of cache exacerbates the data redundancy inherent in inclusive cache designs. Where it is still beneficial to have larger, slower inclusive caches to act as snoop filters for smaller, faster caches nearer to the core, those benefits can be undermined by excessive data duplication and frequent back-invalidations when the larger

cache is only a factor of two- to four-times the size of the smaller cache[6].

One technique to address the aforementioned issues is *partial inclusivity*. Partially inclusive caches can help lessen the effects of data duplication in a cache hierarchy by removing the restriction that higher-level caches have all their cache lines duplicated in lower-level caches. In most cases, though, designs incorporating a partially inclusive cache can still have robust snoop filtering akin to designs incorporating similarly-sized inclusive caches.

This paper describes two partially inclusive mid-level cache designs that differ in how much state they retain in their directory arrays, and examines the performance and complexity tradeoffs that result; specifically, *ThinL2*'s directory retains only the state corresponding to the L2, while *WideL2*'s directory array retains both L2 state and additional L1D and L1I state. We begin with a discussion of how the caches manage coherency, followed by a discussion of our expected first-order differences. Our two approaches are then modeled and benchmarked for an empirical comparison of our two approaches. We conclude with thoughts on how one might best approach designing and implementing a partially inclusive mid-level cache.

Chapter 2

Background

Most modern high-performance processors incorporate a three-level cache hierarchy[5][1][2]. A typical three-level cache hierarchy incorporating a large last-level cache (LLC/L3), a smaller mid-level cache (L2), and still smaller instruction (L1I) and data (L1D) caches is shown in figure 2.1. The system shown has instruction and data caches (L1I and L1D), as well as mid-level caches (L2), coupled tightly into each core. Each core's L2 is connected to a shared last-level cache (LLC).

All caches we discuss are write-back caches, meaning they are capable of containing lines with local modifications[7]. We also assume that all caches are write-allocate, meaning that transactions originating from both load instructions ("reads") and store instructions ("read-invalidates", or "ReadInv" transactions) can allocate into the cache. This matches the behavior of Intel's high-performance microarchitectures[4], so we believe this assumption to be valid. However, it may be interesting to repeat the study with different types of insertion and modification policies.

2.1 Inclusive Cache Management

The cache models discussed in the paper generally adhere to the Modified-Exclusive-Shared-Invalid (MESI) protocol defined by Papamarcos, et al.[8]

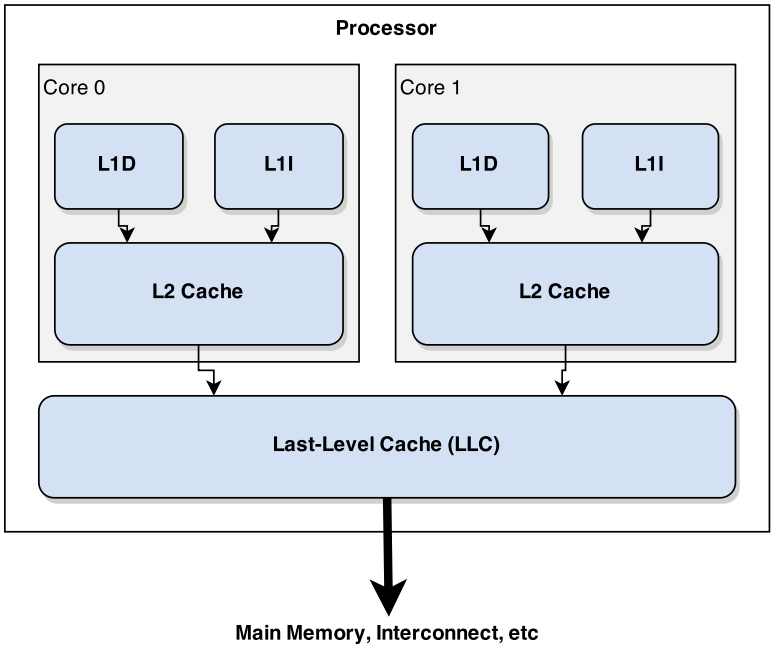

Figure 2.1: A typical three-level cache hierarchy in a chip multiprocessing (CMP) package.

Table 2.1: Inclusive Cache Structures

Structure	Purpose	Comments
Tag Array	stores each cache line's tag	
MESI Array	stores one or more MESI states per cache line	
Tag/MESI Pipeline	pipeline to access Tag and MESI Arrays	
Transaction Queues	tracks transaction state	
Data Array	stores each cache line's data	does not affect coherence
Data Pipeline	pipeline to access Data Array	does not affect coherence

unless specified otherwise. Many modern high-performance microprocessors implement MESI[4] or a variant thereof[11]. The rest of this section describes the hierarchy in figure 2.1 in more detail, assuming all caches are inclusive. We will also introduce two issues inherent in inclusive caches: data duplication and back-invalidation[6]. Later sections will describe how ThinL2 and WideL2 implementations differ from the strictly inclusive L2 both in implementation and in addressing those two issues.

This section assumes the caches in 2.1 are inclusive and describes their behavior. Specifically, the inclusion property means that any cache line in a particular cache must also be in all lower-level caches. For example, a line in the L2 will always be in the LLC as well. That line may or may not exist in the L1I or L1D. To support inclusion, each cache contains the structures shown in table 2.1. The table references the data array and pipeline for completeness; discussions of these structures is outside the scope of this document, since they do not materially affect the coherence schemes.

When the MESI was defined in 1983, coherence was managed across only one level of private caches[8]. As a result, MESI as it was defined by Papamarcos does not directly scale to a multi-level inclusive cache hierarchy. Each inclusive cache's MESI state is defined with respect to peer- and lower-level caches only, meaning that a Shared cache line in the L2 is only Shared with respect to the LLC and other L2s. That same cache line might be Mod-

Table 2.2: Legal MESI States in an Inclusive Cache Hierarchy

LLC	L2	L1D	Comments
I	I	I	A line must be in the LLC for it to exist anywhere.
S	I	I	
	S	I/S	
E/M	I	I	
	S	I/S	L2 data matches LLC data
	E	I/S/E/M	L2 data matches LLC data, L1D might be Modified
	M	I/S/E/M	LLC != L2 != L1D possible if all are Modified

ified in the LLC, meaning that the LLC has modifications with respect to system memory. The MESI states in this scenario would also imply that the L2 has the same data as the LLC, and has no local modifications beyond that. Table 2.2 shows all valid combinations of L1D, L2, and LLC MESI states for a fully inclusive cache hierarchy. L1I MESI states are not included, as they are a strict subset of L1D MESI states with Modified and Exclusive states removed.

The tag and MESI arrays in each of the caches need to be accessed from a pipeline as shown in figure 2.2. Since a cache line's respective tag and MESI are inextricably linked–one cannot be interpreted in the absence of the other–they are combined into a singe Tag/MESI array for most of this paper. The array must be accessed near the beginning of the pipeline since the array outputs are used to determine the appropriate action (e.g., determining whether a load request should hit or miss). This implies that any pipeline request that *conditionally* writes the array based on the contents of the array requires two separate passes down the pipeline.

Throughout the rest of this paper, a cache line's MESI state will be referred to with the notation **cache/MESI**. For example, if the L1D is Exclusive, we write L1D/E. Since levels of cache other than the L1s include state corresponding to higher-level caches, those states are referred to as **cache.subdirectory/MESI**. For example, if the L2 believes the L1D has a line Shared, we refer to that state's storage element as L2.L1D and the state itself as

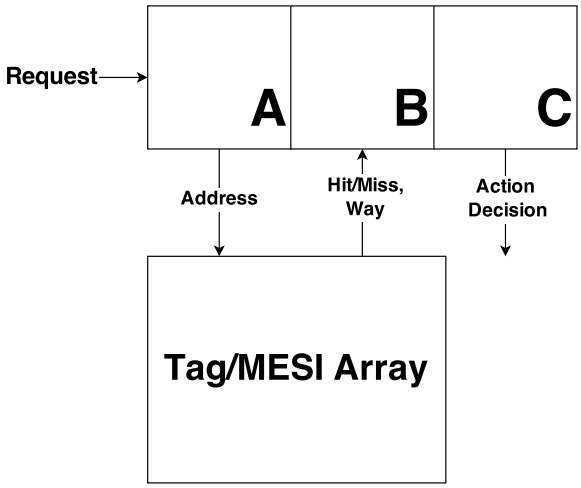

Figure 2.2: A simple three-stage pipeline to access the Tag/MESI array.

L2.L1D/S.

Note that each level of cache sets the maximum MESI state for all levels of cache above it. If a cache line is Invalid or Shared, then the next-higher level of cache may be at most Invalid or Shared, respectively. If a cache line is Exclusive, each of the next-higher levels of cache may be any MESI state due to the fact that caches can silently upgrade a cache line's MESI state from Exclusive to Modified. If a cache line is already Modified, then the restrictions on the next-higher level of cache are the same as if the line were Exclusive. An important implication of this MESI scheme is that any time a cache line is evicted from any level of cache, that line must also be snooped out of all higher level caches. To allow those cache lines to remain would break the inclusion principal. These snoops are referred to as *back-invalidations* and are a tremendous source of inefficiency inherent in inclusive cache hierarchies[6].

As stated previously, each level of cache beyond the L1s maintains

Table 2.3: Legal L2 Directory States in an Inclusive Cache Hierarchy

L2	L2.L1D	L1D	Comments
I	I	I	A line must be in the L2 to also be in the L1D
S	I	I	The L1D cannot own a line without the L2 knowing
	S	I/S	
E/M	I	I	
	S	I/S	L1D data matches L2 data
	E	I/S/E/M	L2.L1D/E implies that the L1D *may* be Modified

directory state corresponding to its next-higher levels of cache. Because of silent Exclusive-to-Modified state transitions, this MESI state is said to be *approximate*. Table 2.3 shows all combinations of legal L2 directory contents and legal L1D contents. As before, the L1I is not shown, and is a strict subset of valid L1D states. Note that the L2.L1D state is never Modified, since the L2 does not learn of L1D modifications until the line is snooped out or evicted from the L1D. Identical relationships exist between the LLC and L2, and are omitted for brevity.

As a result of both the hierarchical MESI states shown in table 2.2 and the next-level directory bits in table 2.3, any level of cache other than the L1s may choose to forego snooping its next-level caches in cases where the snoop is known to be optional. For example, if the L2 receives an invalidating snoop with L2/S, L2.L1D/I, and L2.L1I/I, the L2 may decide to issue a snoop response immediately rather than snooping the L1 caches. This behavior, known as snoop filtering, can dramatically decrease the amount of coherence traffice in a cache hierarchy[9].

Rather than detailing the state transitions required to implement the scheme described above, an example transaction is shown that illustrates how the cache hierarchy processes cacheable loads. Figure 2.3 shows a L1D load that misses in the L2 and hits in the LLC. In this example, the cache lines to be replaced (the *victims*) in the L1d and L2 are clean (i.e. S or E) and dirty

(i.e. M) respectively.

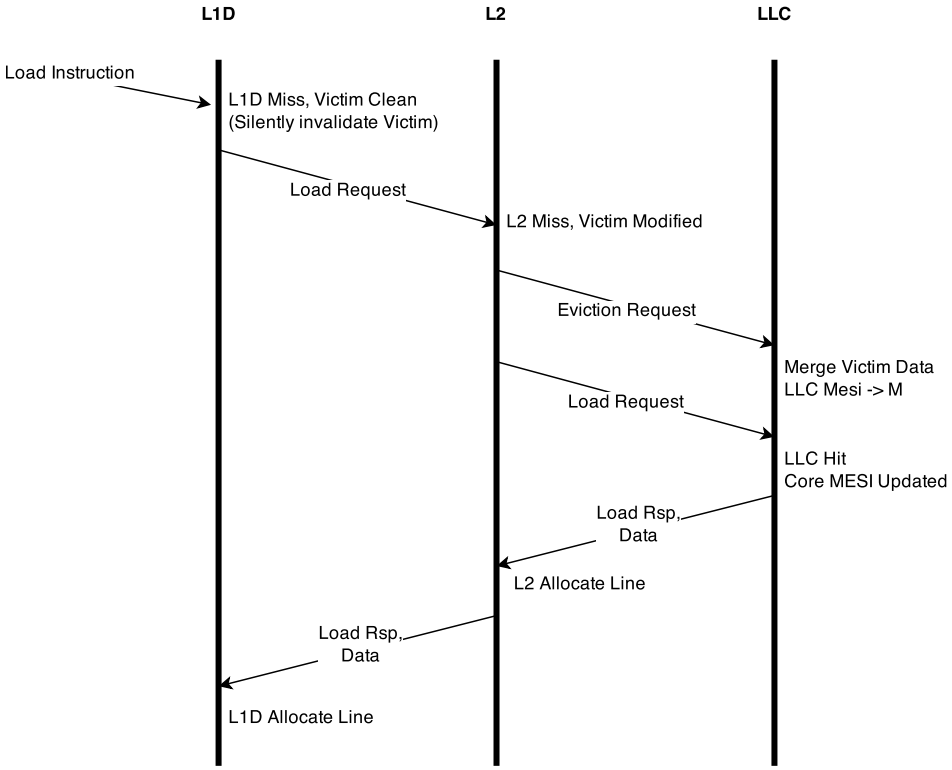

Figure 2.3: Example L1D Load Request that hits in the LLC.

2.2 Partially-Inclusive Caches

As stated in section 2.1, Inclusive caches suffer from two inherent problems: data duplication and back-invalidation[6]. One strategy for addressing those two problems is to make caches partially-inclusive. The problem of data duplication is exacerbated when the larger inclusive cache is similar in size to the smaller included cache[6]; the closer in size the caches, the more storage area is effectively wasted. Partially inclusive caches address this problem by

removing the *requirement* that caches be inclusive, but also not prohibiting inclusion. In a three-level hierarchy where a 256KB partially-inclusive L2 is paired with two 32KB L1s, the L1 and L2 caches collectively contain a theoretical 320KB of unique cache lines. A similar hierarchy with an inclusive L2 would be limited to 256KB of storage in the same three caches, since all L1 cache lines are duplicated in the L2.

Partial-inclusivity also addresses the problem of an inclusive L2 back-invalidating lines in the L1s. As stated in the previous section, an L1D back-invalidation occurs when an inclusive L2 decides to evict a cache line that is resident in the L1D. Whereas an inclusive cache would back-invalidate the line in the L1D upon eviction to enforce inclusion, a partially-inclusive cache would let the line remain in the L1D.

2.2.1 Related Work

There have been notable examples of partially-inclusive cache designs in industry. Intel[10] has adopted this approach since the Nehalem microarchitecture, by implementing a partially-inclusive mid-level (L2) cache between the L1 caches and the fully inclusive LLC. Unfortunately, the Intel Optimization Manual[4] gives no details regarding what sort of information is stored in the L1D, L1I, and L2 caches apart from confirming that they are non-inclusive with respect to one another. Therefore, though they have clearly implemented *some* scheme, it is unclear how it relates to the exploration conducted in this paper.

While there are notable examples of partially-inclusive caches in industry, little work has been done in academia to explore tradeoffs in partially inclusive cache designs. Zahran, et al.[12] suggested in 2007 that a partially-

inclusive scheme (as defined in this paper) "needs a lot of bookkeeping, does not make use of the whole cache area, and does not simplify coherence"; his paper, therefore, focuses on caches that are mutually exclusive with respect to one another. That 2007 was only a year before the debut of Nehalem (along with its new partially-inclusive L2) is telling of the dearth of academic research in the area at the time.

Chapter 3

Defining our Models

We begin by exploring the characteristics of two hypothetical cache systems, each of which incorporates an L1I, L1D, L2, and LLC organized as shown in figure 2.1. In each of these systems the LLC is inclusive of all higher-level caches. Each system's L2 cache is partially inclusive of its L1D and L1I caches, operating with partially inclusive cache behavior described in table 3.1. Table 3.1 summarizes the legal MESI states for three types of cache:

1. An L2 *inclusive* of its L1s
2. An L2 *exclusive* of its L1s
3. An L2 *partially inclusive* of its L1s

It bears mention that our partially inclusive models are contrived, and we cannot verify that our coherence scheme matches that of any processor's

Table 3.1: Valid States for each Inclusion Scheme

	L2	L1D	L1I
Incl.	I	I	I
	S	I/S	I/S
	E/M	I/S E/M	I/S I
Excl.	I	I/S E/M	I/S I
	S/E/M	I	I
Part. Incl.	I	I/S E/M	I/S I
	S	I/S	I/S
	E/M	I/S E/M	I/S I

partially-inclusive L2. In the table, a cache line's MESI state is defined with respect to adjacent and lower levels of cache as described in section 2.1; for example given an inclusive L2, a cache line might be Shared in a core's L1D and L1I, even though the line is Modified in the core's L2. The Shared lines will then have a copy that is clean with respect to the L2, even though the core's L2 will still correctly respond Modified to snoops from the LLC.

As is evident from the table, when a cache line is valid in the L2, the L2 behaves like an inclusive cache. However, if a line is not in the cache–meaning, it is in the Invalid state–the L1D and L1I are allowed to have the line in any of the MESI combinations valid in the L2/E case. The partially-inclusive L2 is thus allowing the L1D and L1I to retain the cache lines without performing back-invalidation.

In the first system (hereafter referred to as *ThinL2*) our L2's directory contains only enough state to remember the MESI state of the the L2; it does not record the MESI state of its core's local L1s. In the second system (hereafter referred to as *WideL2*) our L2's directory contains all state in ThinL2 in addition to an approximate MESI state (as defined in section 2.1) of the L1D and L1I for each cache line stored in the L2. We say that this state is approximate to allow for the L1D to upgrade a line from Exclusive to Modified without notifying the L2. Except where specified otherwise, the LLC, L1I, and L1D caches are identical between the two systems. To be clear, the WideL2 does not necessarily contain all lines in its L1D and L1I. However, if cache line is in the L2 as well as the L1D or L1I, the L2's MESI array will reflect that line's presence in the L1D or L1I.

The tradeoffs between ThinL2 and WideL2 potentially impact memory performance in several ways. We will illustrate some of these impacts in sec-

tion 4 by examining how each of our L2 models might respond to hypothetical load, eviction, and snoop traffic.

Chapter 4

Analysis of ThinL2 and WideL2 Models

As stated previously, caches exist to service loads. Therefore, our analysis of ThinL2 and WideL2 must begin by examining how each of the caches handles incoming load requests from the L1I and L1D. Compared to inclusive caches, loads to partially inclusive caches have several added complications. Because the caches are not strictly inclusive, L1D loads that miss the L2 may be valid in the L1I and vice versa; therefore, the loads must be handled carefully to ensure that proper MESI coherence is maintained. If the same ambiguity exists in other cases–e.g., L2 hits in ThinL2–care must be taken to prevent coherence violations that arise from situations legal with partial-inclusion, but illegal with strict inclusion.

Jaleel, et al.[6] also showed another significant performance advantage of partially-inclusive caches over inclusive caches: dramatically reduced back-invalidations. Because it is legal for a partially-inclusive cache to silently invalidate a clean cache line at any time, they do not perform back-invalidations on clean lines at all; it is therefore not useful to compare back-invalidations between ThinL2 and WideL2. However, both caches do potentially generate wasteful snoop traffic to the L1s due to their inability to accurately filter snoops to higher level caches. These wasteful snoops could be avoided entirely if the L2 were inclusive. We therefore model both an inclusive L2, ThinL2, and WideL2, and measure *total* L1D snoops for all three L2 cache models

over traces of SPEC2006. This allows us to visualize how inclusive cache back-invalidations compare to wasteful snoop traffic in each partially-inclusive cache.

4.1 Load Transactions

Load requests can either be reads (e.g. `mov ebx, dword ptr [eax]`), writes (e.g. `mov dword ptr [eax], ebx`), or read-modify-writes (e.g. `inc dword ptr [eax]`). Because we already assumed that our caches are write-back and write-allocate, we do not need to distinguish between writes and read-modify-writes (since both transactions result in the line being allocated in the E state in the L2). Writes and read-modify-writes are collectively referred to as read-invalidates (ReadInv). We do have to distinguish these from read transactions, because reads are allowed to allocate a line into either the Shared or Exclusive state. This section contains a look at how both ThinL2 and WideL2 respond to both read and read-invalidate load transactions. It also reveals a potential pitfall to be wary of regarding how a microprocessor supports L1D writes to lines in the L1I (i.e., self-modifying code).

4.1.1 Load Transactions in the ThinL2

Table 4.1 describes how the ThinL2 handles incoming L1D and L1I load transactions. Because the L2 does no bookkeeping on behalf of the L1s, any time the L1D issues a write allocation to the L2 the L2 must conservatively snoop the L1I. Another similar potential issue with this model that arises in the table is that a L1D read that hits Exclusive or Modified in the L2 can only result in the line being taken Exclusive in the L1D if the L1I is snooped. Foregoing the L1I snoop would allow the L1D to take the line Exclusive while

the L1I retained the line Shared, leading to a clear coherence violation.

On the L1I side, another ThinL2 issue is that all L1I accesses that do not hit Shared in the L2 require a snoop into the L1D. This is because a line that is Exclusive or Modified in the L2 could potentially also be Exclusive or Modified in the L1D. Further, a line that is Invalid in the L2 could also be Exclusive or Modified in the L1D due to the L2's partial inclusivity. The L2 could wait for the LLC response in the latter case, and only snoop the L1D if the line is Exclusive or Modified. It is not obvious that skipping the superfluous snoop in the S cases is advantageous given that all accesses that hit Exclusive/Modified in the LLC would then have their snoop delayed until the LLC response. Neither are particularly palatable options.

Table 4.1: ThinL2 Response to L1I and L1D Loads

Type	L2 MESI	Array Update	Actions
L1D Read	I	allocation	Forward to LLC
	S	-	L2 Hit
	E,M	-	L2 Hit
L1D ReadInv	I,S	upgrade	Forward to LLC, Snoop L1I
	E,M	-	L2 Hit, Snoop L1I
L1I Read	I	allocation	Forward to LLC, Snoop L1D
	S	-	L2 Hit
	E	maybe mesi wr	L2 Hit, Snoop L1D
	M	-	L2 Hit, Snoop L1D

Table 4.1 also has an important implication on how the L1D accesses lines that are potentially stored in the L1I. This is a pattern that is central to self-modifying code (SMC), which very is challenging to support in high-performance, out-of-order processors. If SMC support depends at all on the L2 to know when there is potential SMC in the machine–for example, by serializing the core when a possible SMC condition is detected–that mechanism is almost certainly based on the L2's ability to detect when a L1D load request hits in the L2 and must snoop the L1I. In that scenario, the processor core's ability to execute out-of-order might be hampered by the fact that the L2

now has to snoop the L1I every time it processes a ReadInv from the L1D. This is an especially high cost to pay given the relative infrequency of true SMC. That shortcoming could potentially be addressed in the L2-miss case by delaying the L1I Snoop until after the LLC responds, allowing the L2 to only issue the Snoop if the LLC responds that the cache line already may have existed somewhere in the core cache hierarchy. However, this solution can potentially delay the L1I snoop by serializing it behind a relatively slow LLC access, and does nothing to help the cases where the line is Shared, Exclusive, or Modified in the L2.

Most of the transactions in 4.1 do not require MESI array updates. The "maybe mesi wr" array update only occurs if the line is Modified in the L1D when the L1I's load hits Exclusive in the L2.

4.1.2 Load Transactions in the WideL2

Table 4.2 describes how WideL2 handles incoming L1D and L1I load transactions. Because the WideL2 does record approximate L1 MESI information, it can avoid many of the excessive L1 snoops that are evident in the ThinL2 cache. In the cases where L1 snoops are problematic, the L2 can defer the L1 snooping until after it receives a response from the LLC that affirms that the line might exist in the other L1 cache.

Even though the snooping behavior of the WideL2 cache is much more efficient than the ThinL2 cache, every type of load transaction requires an update to the MESI array to record that the corresponding L1 cache is obtaining a copy of the cache line. The relative array costs will be explored in more detail later in section 4.3.2. The WideL2 still cannot distinguish between an L1D being Modified or Exclusive, since to do so would require the

Exclusive→Modified transition in the L1D to forward the update information to the L2.

Table 4.2: WideL2 Response to L1I and L1D Loads

Type	L2 MESI	L2.L1D/L1I	Array Update	Actions
L1D Read	I	-/-	allocation	Forward to LLC
	S,E,M	-/-	mesi wr	L2 Hit
L1D ReadInv	I	-/-	allocation	Forward to LLC, Snoop L1I
	S	-/I	upgrade	Forward to LLC
	S	-/S	upgrade	Forward to LLC, Snoop L1I
	E,M	-/I	mesi wr	L2 Hit
	E,M	-/S	mesi wr	L2 Hit, Snoop L1I
L1I Read	I	-/-	allocation	Forward to LLC, Snoop L1D
	S	-/-	mesi wr	L2 Hit
	E,M	I/-	mesi wr	L2 Hit
	E,M	S/-	mesi wr	L2 Hit
	E,M	E/-	mesi wr	L2 Hit, Snoop L1D

4.2 LLC Snoop Transactions

The L2 must also handle incoming snoop transactions from the LLC. In this section we examine how each of the L2 models responds to snoops, and how the presence or absence of L1 MESI bits affects the L2's ability to filter snoops. In a fully inclusive cache hierarchy, the L2 need not worry about the presence of valid lines in the L1 caches of which the L2 is unaware. In both ThinL2 and WideL2, however, that condition is supported and must be handled with care. Further, even in the case of a snoop hitting in the L2, ThinL2's lack of L1 directory state necessitates even further conservatism to prevent coherence problems.

It occurs to us that having an inclusive LLC in our hierarchy may still provide robust snoop filtering for the L2, so the L2's inability to aggressively filter LLC snoops may or may not be problematic compared to an inclusive L2. However, if a partially inclusive L2 is being designed for a given L3, or vice versa, the interactions between the L3 and L2 with regard to snoop filtering are important to consider in choosing an appropriate design.

4.2.1 LLC Snoop Transactions in the ThinL2

Table 4.3 describes how ThinL2 handles incoming snoops from the LLC. Because the ThinL2 does not record L1 MESI information in its MESI array, the L1s must always be snooped in response to incoming ReadInv snoops. Since the L2 does not know the L1s in which a line might reside, the L2 must be conservative and snoop all L1s. In the case of snoops from Read transactions, only the L1D must be snooped. This is due to the L1I's inability to have local modifications. If an application tends to have lines in the Shared state ThinL2 performance might be acceptable, but the inability to snoop filter on ReadInv snoops will likely have a substantial performance impact. Such an impact will also depend on how effectively the LLC can filter snoops for the L2.

In cases where a snoop is forwarded to one or both L1 caches, the L2 still responds to the LLC, but crafts a response that represents the resultant MESI state of the entire L1D/L1I/L2 complex. Only transactions that affect L2 MESI need to perform array updates.

Table 4.3: ThinL2 Response to LLC Snoops

Type	L2 MESI	Array Update	Actions
Read	I	-	Forward to L1D
	S	-	Respond Hit
	E,M	L2→S	Forward to L1D
ReadInv	I,S,E,M	L2→I	Forward to L1D, L1I

4.2.2 LLC Snoop Transactions in the WideL2

Table 4.4 describes how WideL2 handles incoming snoops from the LLC. Because the WideL2 does record L1 MESI information in its MESI array, the L1s are snooped less frequently than in the ThinL2 model. In cases where the L2 is Exclusive or Modified, but the cache line is in both the L1D

and L1I, the L2 can safely respond with data to a Read transaction's snoop without further snooping the L1s.

The biggest source of wasteful snoops introduced by WideL2 are those inherent to partially inclusive caches. If a snoop misses in the L2, it must conservatively forward the snoop to either the L1D (in the case of a Read transaction) or both L1D and L1I (in the case of a ReadInv transaction). Eliminating those wasteful snoops would require us to adopt a different inclusive scheme. Luckily, the LLC will still be filtering snoops for the L2 and should thus prevent most snoops that would otherwise miss in the L1I, L1D, and L2 from entering the L2 entirely.

Table 4.4: WideL2 Response to LLC Snoops

Type	L2 MESI	L2.L1D/L1I	Array Update	Actions
Read	I	-/-	-	Forward to L1D
	S	-/-	-	Respond Hit
	E/M	I/I	L2→S	Respond Hit/HitModifed
	E/M	S/S	L2→S	Forward to L1D/L1I
	E/M	E/I	L2→S	Forward to L1D
ReadInv	I	-/-	-	Forward to L1D/L1I
	S	I/I	L2→I	Respond Hit
	S	S/S	L2→I	Forward to L1D/L1I
	E/M	I/I	L2→S	Respond Hit/HitModifed
	E/M	S/S	L2→S	Forward to L1D/L1I
	E/M	E/I	L2→S	Forward to L1D

4.2.3 Experimental Setup for Performance Evaluation

While in previous sections we defined the conditions under which both ThinL2 and WideL2 are required to snoop the L1D, the discussion so far has contained no quantitative measurements of any of the properties we have mentioned. Since the frequency of L1D snooping is difficult to reason about, we used a simple cache model, CacheSim, written by the author to measure the frequency of wasteful L1D snooping in a simulated cache hierarchy.

The author began by augmenting CacheSim, a simple C++ multi-level cache simulator used internally at Centaur Technology for reasoning about

cache requirements in complex inclusive cache topologies. CacheSim did not support partially-inclusive caches, so the bulk of the work on the simulator was done to add support for both WideL2 and ThinL2 caches. CacheSim connects to Centaur Technology's golden processor model, CNSIM, and can be executed in two modes:

- verification mode, which executes a number of built-in self checks to verify connectivity and coherence; and,
- fast mode, which disables all self-checking routines and simply executes x86 instructions from an input test vector.

CacheSim's verification mode was used extensively to validate that each of the inclusive L2, ThinL2, and WideL2 models were functioning correctly before running the experiments. We chose SPEC2006 as our input test suite to evaluate performance, since it is a common performance metric and we already had access to time-tested CNSIM test vectors that are representative of critical memory traffic. We simulated each SPEC2006 benchmark with three different four-core configurations:

- Sliced 2MB-per-core L3, 256KB Inclusive L2, 32KB L1D and L1I
- Sliced 2MB-per-core L3, 256KB WideL2, 32KB L1D and L1I
- Sliced 2MB-per-core L3, 256KB ThinL2, 32KB L1D and L1I

Our experimental setup suffers from a few limitations. One is that since we are using internal software, we unfortunately cannot make available our testing apparatus. Though the topologies and cache sizes are realistic and

comparable to similar products on the market[4], we are running only single-threaded versions of the benchmark without any inter-core interference from other applications, other applications, and so forth which would be present on a real machine. Still, as an exploratory report, we see no reason why these limitations would be at all unacceptable.

4.2.4 Snoop Efficiency for SPEC2006

We now consider in more detail the frequency of wasteful snoops that are sent into the L1D. Figure 4.1 shows the L1D Snoop frequency on our input data set sampled from SPEC2006. Since wasteful, unneeded snoops are the most perilous in the midst of heavy load traffic–after all, bothering a busy cache is much worse that bothering an idle cache–the Y axis plots snoops per thousand loads (SPKL). As expected, the ThinL2 model performs relatively poorly, as the small directory structure and limited accounting responsibility meant that the L2 had to conservatively generate lots of extra snoop traffic to the L1s. Results are generally in line with previous sections.

There are, however, some interesting caveats. The WideL2 model *far* surpassed our expectations in terms of addressing wasteful L1D snoop traffic. Whereas ThinL2 does create a significant amount of L1D snoop traffic in several benchmarks, the WideL2 does not dramatically increase L1D snoop traffic in any benchmark. This holds true even when compared to the L1D snoop traffic in the Inclusive model. Since the Y axis in figure 4.1 plots all L1D snoops, not just those that were unneeded, we see the WideL2 cache snooping the L1D even less than the inclusive L2's back invalidations. This is due to the WideL2 cache's ability to locally invalidate lines *without* bothering the L1D. This benefit is also present in the ThinL2 model, but is overshadowed by the

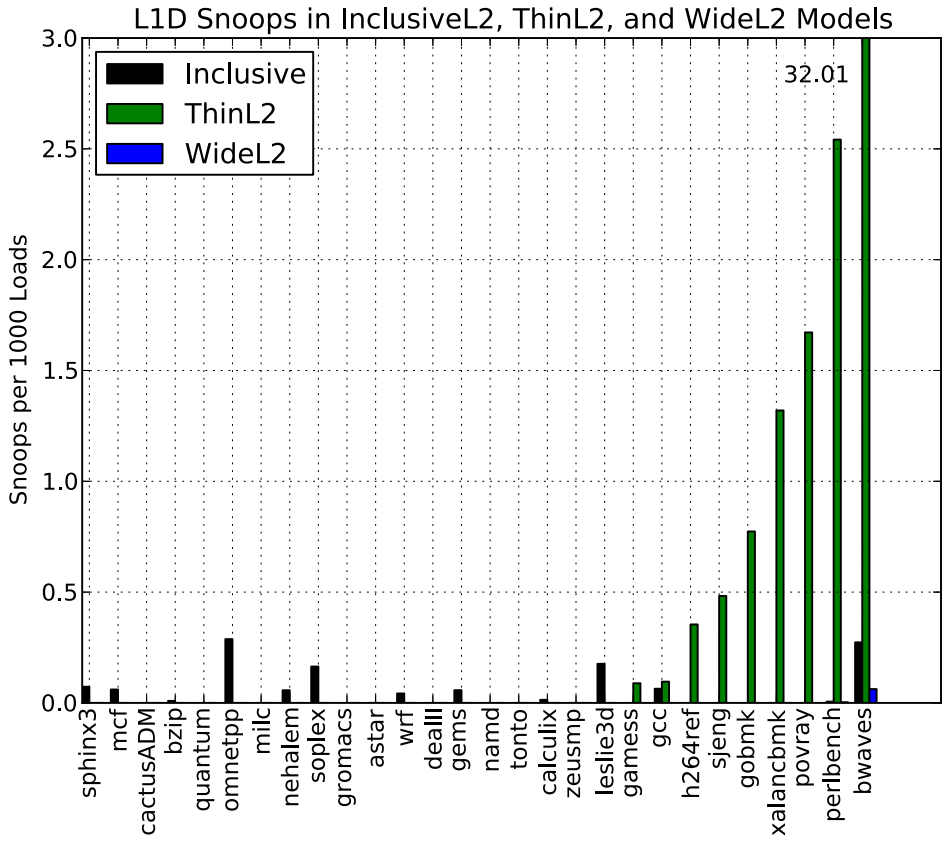

Figure 4.1: Snoops per 1000 Loads on SPEC2006 Workloads

many L1 snoops required to maintain coherence.

4.3 Implications on the Design of the MESI Array

Our two L2 models differ in how much state they keep in their respective MESI arrays, but the manner in which that state is tracked has many architectural implications beyond simple storage capacity. Because the ThinL2 model tracks only L2 MESI state, most of the transactions that enter ThinL2 only

Table 4.5: ThinL2 Array Updates by Transaction

Src/Dst	Type	MESI Access	Comments
L1D	Load	R	Read to determine Hit/Miss
	Evict	R/W	Read to determine Hit/Miss
			Write to update L2 MESI
	Snoop Rsp	-	No array accesses needed
L1I	Load	R	Read to determine Hit/Miss
	Evict	-	No array accesses needed
	Snoop Rsp	-	No array accesses needed
LLC	Load	W	Write for new allocation/upgrade
	Evict	R/W	Read to determine eviction is required
			Write to mark line Invalid
	Snoop	R/W	Read to determine Hit/Miss
			Write to update L2 MESI

need to read the Tag/MESI array. On the other hand, the WideL2 model tracks L1 MESI state in addition to L2 MESI state, so many transactions required reads *and* writes to the Tag/MESI array. In this section, we discuss the implications of each Tag/MESI array format on array accesses for Load, Snoop, and Eviction transactions.

4.3.1 ThinL2 MESI Array Requirements

The most obvious benefit of ThinL2 compared to WideL2 is array size. The ThinL2 MESI array only needs to store two state bits per cache line to fully encode L2 MESI. Further, the simpler array format is not just smaller; it also needs to be updated by fewer transaction types, meaning it could potentially be designed with fewer access ports than a similarly-pipelined WideL2. Table 4.5 shows the array access requirements for each of the common transactions that the L2 services. In this table, we assume that whatever transaction generated the snoop into the L1D or L1I is also responsible for updating the L2 MESI appropriately as needed.

In addition to utilizing a relatively small Tag/MESI array, ThinL2 offers another convenience: L1D and L1I loads–the most performance-critical transactions in the L2–only require an array read, not an array write. The ar-

ray access should occur early in the pipeline (e.g., in the A stage) since it must happen before the proper action is determined. If the action determined in the C-stage involves a write to the array, as is sometimes the case with snoops and evictions, the transaction would have to replay, request the pipeline again, and do a totally separate write request.

4.3.2 WideL2 MESI Array Requirements

In the same way that the ThinL2's modest storage requirements work to its benefit, the WideL2's more intricate storage requirements introduce substantial complexity to the design. The WideL2 MESI array needs to store the same two bits of L2 MESI state per cache line that the ThinL2 stores, but must additionally store three more bits per cacheline: two bits for L1D mesi, and one bit per L1I state. The L1I can only be Shared or Invalid, so it doesn't require two bits of storage to express all four MESI states. Table 4.6 shows the array accesses requirements for each of the common transactions that the L2 services.

Table 4.6: WideL2 Array Updates by Transaction

Src/Dst	Type	MESI Access	Comments
L1D	Load	R/W	Read to determine Hit/Miss
			Write to update L2.L1D MESI
	Evict	R/W	Read to determine Hit/Miss
			Write to update L2.L1D MESI
	Snoop Rsp	W	Write to update L2, L2.L1D MESI
L1I	Load	R/W	Read to determine Hit/Miss
			Write to update L2.L1I MESI
	Evict	R/W	Read to determine Hit/Miss
			Write to update L2.L1I MESI
	Snoop Rsp	-	Write to update L2.L1I MESI
LLC	Load	W	Write for new allocation/upgrade
	Evict	R/W	Read to determine eviction is required
			Write to mark line Invalid
	Snoop	R/W	Read to determine Hit/Miss
			Write to update L2 MESI

The added directory bits clearly cost more than just the additional storage size requirements. Keeping them in sync requires turning almost *every*

transaction type into an array write. Worse still, many transaction types–including the most performance critical ones, L1D and L1I loads–require both reads *and* writes to the MESI array. This is a clear implementation challenge. Requiring each load request to traverse the pipeline twice (first to read, and again to write) would significantly hurt cache bandwidth. Again looking back to figure 2.2, the fact that the array read must occur before the action decision makes reading and writing in a single pass very challenging and potentially intractable.

Chapter 5

Evaluating the Tradeoffs

In the previous chapter we performed a qualitative analysis of how both ThinL2 and WideL2 respond to L1 loads, and a quantitative analysis of how much wasteful L1D snoop overhead is introduced by each cache. Given the results of the previous chapter, the WideL2 model would be clearly superior to a ThinL2 model if one could avoid the problems mentioned in section 4.3.2. This chapter provides two possible ways to address the Tag/MESI array issues that we previously identified, potentially allowing for a WideL2 design that is unencumbered by the problems previously mentioned.

5.1 Fixing WideL2

There are a few possible ways to address the read-modify-write issue inherent in WideL2 as defined in section4.3.2. The most obvious way to address the problem is to employ a dual-ported MESI array. This architecture is shown in figure 5.1. Although it does address the problems with read-modify-writes, letting e.g. L1D loads hit from the read port and set their bit in the write port, multi-ported arrays can be expensive. Even if the cost is not prohibitive as shown, if the number of pipelines increases to two or more, it's likely that the resulting array–a quad-ported Tag/MESI array–will be highly undesirable.

A better approach is to observe that L1D loads base their action deci-

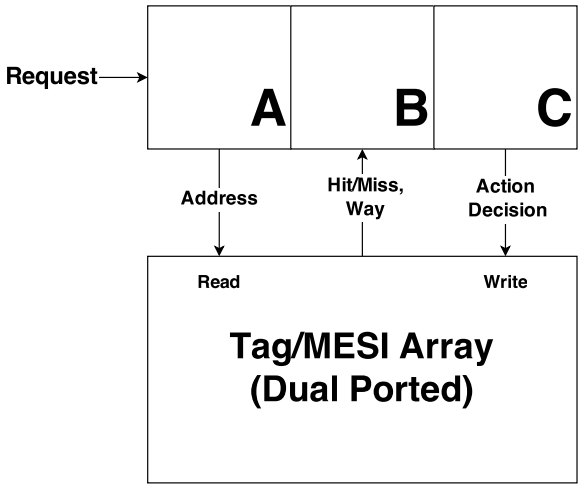

Figure 5.1: A three-stage pipeline to access the MESI array, with dedicated read and write ports.

sion (Hit/Miss, and whether or not to snoop the L1I) on only L2 and L2.L1I. L2.L1D is written as a result of a Hit, but that decision can be predicated on only L2 MESI and does not require knowledge of L2.L1I. Similar observations can be made about the reading of L2 and L2.L1D and the writing of L2.L1I on L1I loads. All load transactions require reading the L2 MESI, but none require writing it. Table 5.1 shows the results of separating the L2, L1D, and L1I accesses for each common transaction type from table 4.6.

Table 5.1: WideL2 Array Updates, Separated MESI

Src/Dst	Type	L2	L2.L1D	L2.L1I
L1D	Load	R	W	R
	Evict	R	W	
	Snoop Rsp	W		
L1I	Load	R	R	W
	Evict	R		W
	Snoop Rsp			
LLC	Load	W	W	W
	Evict	R/W	R/W	R/W
	Snoop	R/W	R/W	R/W

Based on 5.1, another possible approach to doing single-request load

hits is to split the L1D and L1I MESI arrays out of the L2 Tag/MESI array. The new L1D- and L1I-specific structures could be accessed in the C stage–well after the A-stage MESI array access–and can either have their outputs factor into the action decision, or can be updated as a result of actions determined on either the A-stage MESI outputs. For transactions that cannot be done in a single cycle, secondary or tertiary requests could retain state from a previous pipeline access. Figure 5.2 shows how such a pipeline might be connected to the arrays.

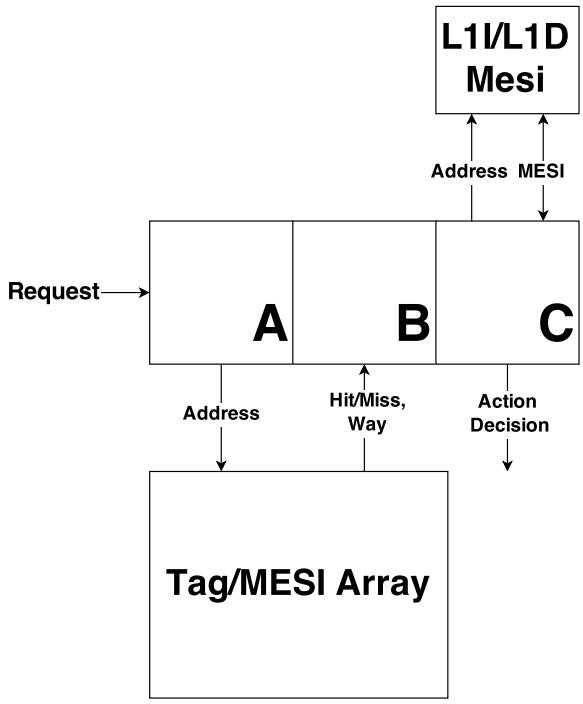

Figure 5.2: A three-stage pipeline to access the MESI array, with L1D and L1I arrays split from the L2 MESI array.

Chapter 6

Conclusion

In this paper we discussed two possible implementations of a partially inclusive L2 cache: ThinL2, which only stores L2 MESI in its Tag/MESI array, and WideL2, which also stores L1D and L1I MESI in its Tag/MESI array. We set out to answer the question of which design is better, and our results confirm the age-old engineering maxim: *it depends*.

We defined expected behavior of each cache design with respect to how each cache handles incoming load transactions, incoming evictions, and external snoops. We also performed an empirical analysis of L1I→L1D cross-snooping behavior by modeling each of the caches, including a typical inclusive L2 cache, and simulating SPEC2006 traces. Table 6.1 summaries our observations, with comments on how some of the observed issues can be addressed.

Table 6.1: A Summarized Comparison of ThinL2 and WideL2

Feature	**ThinL2**	**WideL2**	**Comments**
Tag/MESI Array Size	smaller	larger	WideL2may be prohibitively large if multi-ported; see 5.1
L1D/L1I Load Array Accesses	read only	read + write	WideL2can be made to support the accesses in a single pass; see 5.1
Wasteful snooping of L1s	wasteful	efficient	L1→L1 cross-snooping is bad in ThinL2, even if LLC does snoop filtering
Back-invalidates	efficient		Partially-inclusive caches avoid back-invalidates by design
Data duplication	better than inclusive		Partially-inclusive caches still duplicate data, but are more efficient than inclusive caches.

For best performance, our results suggest that the ThinL2 array and

complexity savings savings are more than outweighed by the performance benefits of a properly implemented WideL2. That is not to say that WideL2 is without problems; we demonstrated two serious issues that are barriers to an efficient implementation, and proposed solutions that address each of them. The solutions may not be practical in all WideL2 implementations, but it suffices to say that potential solutions to the issues we observed do exist.

Index

Bibliography

[1] AMD. Amd fx processors. http://www.amd.com/en-us/products/processors/desktop/fx, 2015. [Online; accessed 04-May-2015].

[2] AnandTech. The ipad air review. http://anandtech.com/show/7460/apple-ipad-air-review/3, 2013. [Online; accessed 04-May-2015].

[3] Standard Performance Evaluation Corporation. Spec cpu 2006. https://www.spec.org/cpu2006/, 2014. [Online; accessed 06-May-2015].

[4] Intel. Intel 64 and ia-32 architectures optimization reference manual. https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2014. [Online; accessed 01-May-2015].

[5] Intel. Products (formerly haswell). http://ark.intel.com/products/codename/42174/Haswell, 2015. [Online; accessed 04-May-2015].

[6] Aamer et al. Jaleel. Achieving non-inclusive cache performance with inclusive caches. www.jaleels.org/ajaleel/talks/TLA_MICRO2010.ppt, 2010. [Online; accessed 06-May-2015].

[7] Norman Jouppi. Cache write policies and performance. http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-91-12.pdf, 1991. [Online; accessed 06-May-2015].

[8] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *25 Years of*

the International Symposia on Computer Architecture (Selected Papers), ISCA '98, pages 284–290, New York, NY, USA, 1998. ACM.

[9] Aanjhan Ranganathan. Experimental analysis of snoop filters for mpsoc embedded systems. `http://www.tuxmaniac.com/work/abstracts/master.pdf`, 2010. [Online; accessed 07-May-2015].

[10] real world technologies. Inside nehalem: Intel's future processor and system. `http://www.realworldtech.com/nehalem/7/`, 2008. [Online; accessed 01-May-2015].

[11] Taeweon Suh, Douglas M. Blough, and Hsien-Hsin S. Lee. Supporting cache coherence in heterogeneous multiprocessor systems. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '04, pages 21150–, Washington, DC, USA, 2004. IEEE Computer Society.

[12] M. Zahran. Non-inclusion property in multi-level caches revisited. `http://www.mzahran.com/ijca07.pdf`, 2007. [Online; accesses 04-May-2015].