UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ENERGY EFFICIENT MEMORY SPECULATION WITH MEMORY LATENCY TOLERANCE SUPPORTING SEQUENTIAL CONSISTENCY WITHOUT A COHERENCE PROTOCOL

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

David Alexander Munday

March 2013

The Dissertation of
David Alexander Munday is approved:

Table of Contents

List of Figures

List of Tables

Abstract

Energy Efficient Memory Speculation with Memory Latency Tolerance Supporting Sequential Consistency Without A Coherence Protocol

by

David Alexander Munday

Modern out-of-order processor architectures focus significantly on the high performance execution of memory operations. Because memory instructions pose ordering requirements, their execution becomes a significant bottleneck for out-of-order execution, particularly slow executing loads.

Many high-overhead structures such as StoreSets, Load Queues and Store Queues are included in these processors to support memory speculation in an attempt to relax these ordering hazards whereever possible. However, each of these structures presents a new and significant source of energy consumption and design complexity to processor architects.

The execution of memory instructions becomes further complicated by the introduction of multi-core processors. Memory coherence is needed which often requries a coherence protocol and interconnection network. Additionally, the timing and ordering of memory instructions' execution between cores can have critical impacts on program functionality and output which programmers must concern themselves with in the form of a memory consistency model.

This work proposes a decoupled memory execution verification mechanism that supports memory speculation without costly, complex, and scaling limited structures. This in-order verification can reduce the average energy dissipation by over 16% with a simpler design that removes the Load and Store queues, StoreSets, and even invalidation-based cache coherence protocols. These benefits are realized by a system providing the straight-forward and intuitive Sequential Consistency memory consistency model.

Acknowledgments

Graduate school is centrally focused around the working relationship of a graduate student and their advisor. I have been very fortunate to be advised by Jose Renau. Jose's mentoring has been unlike any I have received in the past. He has known when to more directly influence my work and when to be more flexible to allow me to pursue my own ideas. He has also been the most available mentor I could have ever hoped for, always willing to guide my work. I am grateful to have had the benefit of his deep microarchitectural insight and could not have been more fortunate to have been his student.

For the last seven years I have had the pleasure of teaching undergraduates in cooperation with Patrick Mantey, Stephen Petersen, and John Vesecky. Each of these professors has had a distinct impact in my education and growth as an individual. Pat has been a supervisor, mentor, coach, and counselor to me since I was an undergraduate. His consistent impeccably timed guidance has not only contributed to my success in graduate school, but has also kept me pointed in a positive direction. Steve taught me not only many of the technical skills I have today, but what it is to be an engineeer in our society, with the skills to create and make an impact. John has taught me what it means to listen first and and react second, and has shown me how much more effective I can be when I do the same.

All four of these men are perhaps the most learned men that I have ever met, and I hope that over the course of my career I am able to develop as deep a knowledge in my field as they have in theirs.

I have learned a great deal in graduate school. My approach to solving problems and understanding complex ideas is fundamentally different than when I began. But the monumental changes in my thinking that I have developed over the past six years are dwarfed by the major changes my wife Sarah Munday has made in me. Sarah continually reminds me of what is truly important in life. Whether advising me how to mentor an undergraduate student, or proof reading my research papers for errors, Sarah is truly my other half and brings balance to my thinking. I owe much of my success and the completion of my Ph.D. to her. Without her constant support and confidence in my ability to be successful in the presence of significant challenge and difficulty, there were times that I would have given serious consideration to giving up. I am blessed in my life by having a partner who believes in me. For that I am always grateful. The patience she has afforded me and the personal sacrifice she has undertaken to allow me to complete graduate school cannot be repaid. So I instead hope that the work that comprises this dissertation is a demonstration of how hard I worked.

I also owe a tremendous amount of gratitude to my parents Mike and Christine Munday who have always taught me that I can do anything if I am willing to do the work. The values of hard work and honesty that they taught me served me well throughout my Ph.D. experience, whether researching late into the night or writing results for publication. And when I have felt unsure or less confident, I have only had to pick up the phone and call my brother Andrew Munday who consistently reminds me that the key to success is not based solely in luck, or circumstance, or even having great ideas, but in consistent, unyielding, relentless focus on a goal. My brother has taught me that there is no such thing as giving up.

Finally, I must acknowledgte my grandfather Robert Romo, who immigrated to this country from Mexico and studied to become a tool and die engineer. He started a family and a business and taught me that technical competence is only truly successful when complemented by a practical view of the impact our work has in our world. This dissertation is dedicated to him and his legacy to my family.

Chapter 1

Introduction

This dissertation is focussing on re-designing the memory subsystem of a SuperScalar Out-of-Order processor with energy consumption as the primary design consideration. The contributions of this work are as follows:

- An out-of-order core memory execution path that is optimized for energy efficiency.
- A high performance, energy-efficient architecture that supports Sequential Consistency.
- A simple mechanism to avoid excessive memory replay and costly decoupled memory execution checks.
- Several simplifications removing the coherence network from the memory hierarchy, ignoring memory ordering instructions, eliminating StoreSets, and removing the Load and Store queues.

1.1 Memory Speculation is Costly

Out-of-order processors extract instruction level parallelism by speculatively executing instructions out of program order while respecting their data dependencies (read after write etc.). Often, the execution of memory instructions can bottleneck out-of-order processors due to those dependencies. Memory speculation allows increased instruction-level parallelism (ILP) extractaction by attempting to alleviate some of that potential bottleneck. Processors with memory speculation execute load and store memory operations without resolving their data dependencies which must then be verified before commit time. This increased ILP extraction can lead to increased execution performance.

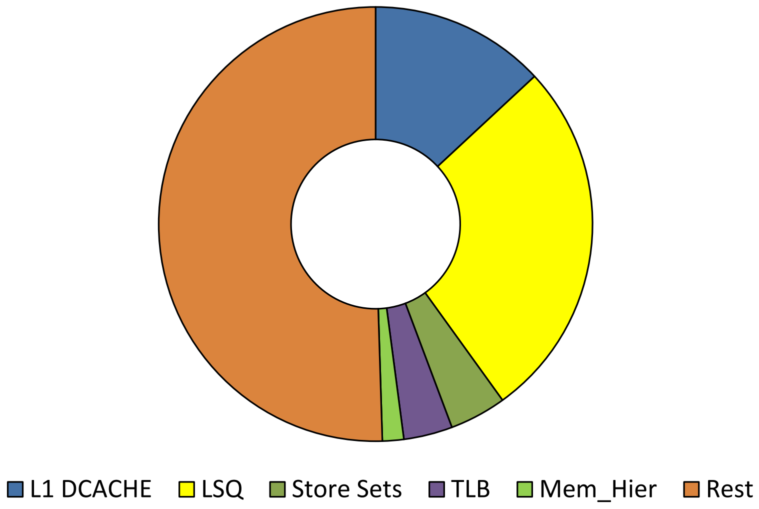


Figure 1.1: L1 data cache and LSQ, represent 40% of the energy per instruction for a typical 4-way core.

Perhaps the most central challenge for current processor designers is to provide increased processor performance without increasing energy consumption. One of the major contributors to processor energy consumption is the memory subsystem. Particularly for Super Scalar out-of-order processors, memory instruction execution is very costly in terms of energy consumption. Processors that issue multiple instructions to the execution pipeline in parallel are commonly referred to as “Super Scalar” processors. As Super Scalar processors increase the number of parallel instructions they attempt to execute, the probability of more in-flight memory instructions increases.

Modern Super Scalar processors extract instruction level parallelism by speculatively executing instructions out-of-order while respecting their data dependencies. Memory speculation allows extraction of additional ILP; processors with memory speculation execute load and store memory operations without resolving their data dependencies. Traditionally, out-of-order processors track and solve memory speculation using fully associative load and store queues commonly referred to jointly as the Load Store Queue(LSQ). The Load Queue(LQ) is responsible for detecting data dependency violations between already executed store instructions and currently executing load instructions. The Store Queue(SQ) orders stores in program order and commits stores to the data cache in program order. Additionally, the SQ is also associatively searched for every executed load for forwarding opportunities by stores that have not yet committed to the data cache, but have resolved their address and data. Because of their fully associative structures, the LQ and SQ present significant scaling challenges to Super Scalar out-of-order processors. As processors seek to support more inflight instructions for greater ILP, the energy consumption of the LQ and SQ increase as a result of adding more entries requiring more fully associative look-ups and more priority logic to carry out additional comparisons which also take longer.

In traditional out of order processors, when the LQ detects that a store instruction has executed too early (by scanning the SQ), it triggers a pipeline flush (replay) and restarts execution from the triggering load. Because the SQ only commits store instructions to the memory hierarchy at commit time in program order, there is no need to repair the state of the cache. Address translation can be another challenge for Load Store Queues. To reduce Load Store Queue access latency as much as possible, Load Store Queues are often accessed using the lower bits of instructions' virtual addresses thus avoiding the need to access the Translation Lookaside Buffer (TLB) to resolve the physical address for each access. However, using the virtual address to detect memory ordering violations can lead to false positives due to virtual aliasing in the lower virtual address bits. Without a full physical address translation, the Load Store Queue cannot differentiate completely between independent addresses.

Another memory operation phenomena that can impact Load Store Queues are silent stores. A silent store is a memory write operation that updates the memory hierarchy with the same value already stored in the target memory location. In [46], Lepak and Lipasti found that 20%-60% of dynamic stores are silent. If the Load Queue searches the Store Queue and finds that a store that was younger than the current load executed out of order, the Load Queue will trigger a replay regardless of whether the store was a silent store or not. If the store was silent, the load would have consumed the correct data and the replay would have been unneeded.

Previous work has proposed replacing LSQ Content Addressable Memories (CAMs) with indexed SRAMs, FIFOs or other structures that consume less energy [18, 61, 67]. These proposed structures perform similar functions to the SQ or LQ, but must generate an index to allow for accessing a single entry or small group of entries in the non-associative replacement structures. The challenge of generating such an index is overcome by using a variety of prediction tables, heuristics, and other dependency prediction mechanisms requiring additional implementation complexity and power.

Memory speculation for the purposes of this dissertation is defined as attempting to relax the requirements for memory disambiguation and forwarding opportunities. Many memory speculation techniques have been introduced to increase processor execution speeds. Previous work in this area has focused on allowing loads to issue past stores speculatively. Their proposed architectures then rely on load re-execution techniques to verify if a load consumed the correct data. If a load consumes incorrect data, a pipeline flush is triggered. Forward progress is guaranteed by restarting execution from the mis-speculated load instruction.

One of the main challenges for load re-execution based memory speculation, is that every load instruction must be executed twice: once in the speculative execution path, and once in the verification execution path. There are several implications to executing each load twice. First, data cache pressure can increase which can in turn increase total power dissipation as well as constrain memory bandwidth. Also, the verification path requires some type of correctness-enforcement mechanism capable of determining when a load has consumed incorrect data. This disseration classifies correctness verification into two types: always correct verification and estimated correctness verification. Always correct verification refers to a correctness verification mechanism that can unambiguously determine when a load has consumed incorrect data. Estimated correctness verification refers to a correctness mechanism that cannot precisely determine if a load has consumed incorrect data, but can detect when a load may have possibly consumed incorrect data.

Whatever structures are utilized to facilitate the verification path for loads will experience additional back pressure as loads wait to receive their correct data from the proposed correctness enforcement mechanism. For example, some proposals have used the re-order buffer or register file to support load verification. In those proposals, additional back pressure on those structures was found and additional ports or other support structures were added to alleviate those stresses.

A different approach allows stores to speculatively update the data cache hierarchy out-of-order [29]. This eliminates the need for a SQ but necessitates rolling back the architectural and memory hierarchy state in the event of a mis-speculated store. Architectural and memory hierarchy rollback is supported by keeping tables storing the architectural state of the processor for every retired instruction and memory hierarchy checkpoints for each retired memory instruction. In the case of a mis-speculation, the processor and cache memory hierarchy are rolled back to the point before the mis-speculation occurred and the load is allowed to execute again.

To avoid the need to forward data from in-flight stores to executing loads, in-window register file communication has been explored [62]. Using modifications to the rename logic in an out-of-order processor, store-load forwarding was accomplished by renaming the load's destination register to the store's input register. This technique eliminates the need for explicit load-store forwarding and thus an associative Store Queue search for stores, but still requires a structure for in-order commit of stores to the memory hierarchy and must still perform correctness verification. These requirements have significant power ramifications due to increased register file pressure, which necessitates extra register file read ports and TLB read ports.

For any memory speculation technique, energy dissipation is a central concern. The traditional Load Store Queue works well by detecting memory ordering violations, but as discussed above, presents significant latency and power scaling issues. Any solution to replace the Load Store Queue must not only detect memory ordering violations as well as the Load Store Queue, but must also be more capable of scaling up the number of supported in-flight memory instructions (size) with more efficiency (energy) than the traditional Load Store Queue. Replacing the Load Store Queue with more scalable structures has been shown possible, but nearly every proposed solution adds additional structures in addition to the newly proposed Store and Load Queues enabling indexing, prediction, correctness verification, or re-execution filtering. Energy is a chief concern when targeting the Load Store Queue.

Energy efficiency has become as critical as performance. It is a key parameter from data centers to mobile devices. Figure 1.1 shows the breakdown of dynamic energy consumption per instruction (EPI) for a typical 4-way out-of-order core, similar to Intel's Sandy Bridge [32], running several SPEC 2006 applications. Both the LSQ and L1 data cache are main sources of energy consumption with over 27% and 13% respectively, but the TLB and the StoreSets are also not negligible with 4% of the total energy consumption each.

Execution of memory operations is costly, and most of the resources in modern processors are specifically built to support it. Processors implement multiple cache levels, Translation Lookaside Buffers (TLB), Load Store Queues (LSQ), and StoreSet predictors to increase memory operation execution efficiency. Most of these resources are on the critical path for load operations. Slow execution of load operations significantly affects overall system performance. As a result, architects invest area, energy, and additional complexity to support low latency load operations. If load operations were not as critical, it would be possible to design much simpler and more energy efficient systems.

With tighter power budgets due to shrinking technology size and increased integration in the same die area, the LSQ has become a significant source of energy consumption as well.

This work proposes a decoupled memory execution verification like the L0 Cache [27] and other works [13, 30]. In a decoupled memory operation, the loads and stores execute speculatively, independent of correctness. Then they are re-executed in-order at retirement to verify correctness. This solution has been shown to be efficient for performance, but there has been no work on designing an energy efficient decoupled memory execution. Intuitively, re-executing all the loads and stores should double the data cache energy consumption for loads and stores. Additionally, previous works propose additional structures to reduce the memory replays that need to be accessed for every memory operation.

The micro-architecture presented in this work is known as the e-PDEMI architecture: the Efficient-Power Decoupled Execution of Memory Instructions architecture.

The e-PDEMI verification of the memory operations is not in the critical path. Instructions waiting for in-order verification hold a reorder buffer (ROB) entry which can potentially slowdown the core. Nevertheless, increasing the retirement phase is less performance sensitive than adding delays to the execution phase.

This work's contribution is to develop an energy efficient memory speculation mechanism that removes some critical structures from the critical path, providing an opportunity to optimize these structures for energy efficiency. The proposed architecture performs decoupled memory execution, and it does not implement an LSQ or StoreSets. e-PDEMI removes the TLB from the speculative execution and it is accessed only at retirement. As a result, e-PDEMI uses a virtually indexed and virtually checked cache-like structure called Virtual Predictive Cache (VPC) instead of the L1 data cache.

Since verification is not in the critical path, e-PDEMI proposes a small but efficient Filter cache [41] that dissipates almost 50% less energy per access than a costly L1 cache.

e-PDEMI also makes use of a small 64 entry direct-mapped VPC Buffer to avoid frequent replays and filter speculative data pollution from the VPC. These two buffers improve memory speculation efficiency.

The fact that the VPC is speculative by nature also allows several energy consumption optimizations with little performance impact. For example, e-PDEMI does not perform write allocation and drops VPC write accesses that miss in the VPC. e-PDEMI also reduces VPC accesses by 10% on average by first accessesing the smaller (and more efficient) VPC Buffer.

A naive memory decoupled execution has several replays which significantly affects processor performance and as a result, energy efficiency. Previous work [56] has proposed structures like Bloom Filters or other mechanisms to avoid frequent replays. A StoreSets-like structure could enforce memory dependences but it would be energy intensive. e-PDEMI includes a novel mechanism that serializes memory operations when two memory replays are detected in close temporal proximity. This effectively avoids memory speculation for a subset of the program execution. Sections 4.1 and 5.1 present results from experiments carried out where e-PDEMI serializes 60 memory operations when two replays have less than 200 retired instructions' temporal distance. This simple mechanism provides an efficient and simple implementation.

This work differs from the existing literature due to its focus on energy efficiency. e-PDEMI focusses not only on the LSQ but the StoreSets, the TLB, and its energy consumption. For a 4-way out-of-order core, e-PDEMI achieves 16.4% average energy per instruction savings while improving the performance by up to 6.4%. These results are achieved with a novel decoupled memory execution architecture that implements a trivial mechanism to avoid memory replays and costly decoupled memory execution checks. e-PDEMI introduces a simple virtually indexed, virtually checked predictor cache (VPC), an L2 filter cache structure, and move the TLB out of the critical path. This work then evaluates several optimizations to improve the energy efficiency of the VPC.

1.2 Multiple Cores Introduce Shared Memory Complexity

As frequency scaling has proven to be infeasible, architects have turned to designing processors with increased capacity for thread-level parallelism to further increase processor performance. One common model for multi-core processors is the Shared Memory Model. In this model, all threads belonging to a particular process share the same memory space. This allows applications to be written as a group of threads that can perform parallel computation on common data. In order for threads to be able to read and write common memory locations in parallel and maintain program correctness, the shared memory must be kept coherent. Memory Coherence refers to the arbitration and tracking of memory location updates by multiple writers to guarantee that all readers receive the most up-to-date copy of common data. Code Listing 1.1 outlines a particular multithreaded algorithm that requires memory coherence. Threads 1 and 2 read the value of Address A at the beginning of execution. Next, Thread 1 writes to Address A, which means that Thread 2's copy of Address A is now stale and not the most recent version of the data stored at Address A. Later, when Thread 2 reads Address A again, it must read the new value stored at address A instead of the value it read previously. Thread 2 cannot know that the data at Address A has changed until it reads it from memory.

One solution to this problem would be to have each of the cores running each thread share a memory element such as a Level 1 data cache. In this design, the coherence challenge above is solved because as soon as Thread 1 (Core 1) writes new data to Address A, it is immediately visible to any read coming from Thread 2 (Core2). However, in practice this design can be infeasible for high performance processors because such a shared memory structure would have to be larger than a typical Level 1 cache to adequately provide storage space for both threads. Modern Chip Multi Processors (CMP) often have many more cores than two, and as the number of cores grows, the size of such a shared structure could become untenable due to energy consumption and latency scaling challenges. Further, the access latency associated with such a large and heavily contested Level 1 Data Cache could significantly diminish processor performance, thus negating the parallel advantage originally sought.

```
Thread1:
  Read  Address A;
  Write Address A;
  Read  Address B;
Thread2:
  Read  Address A;
  Write Address B;
  Read  Address A;
```

Listing 1.1: Example of Shared Memory Data Contention Between Threads

More commonly, each core of a CMP is given a private memory hierarchy that typically is backed by a larger shared hierarchy. One example might be a CMP where each core has a private L1 and L2 cache, with the L2 cache being backed by a common L3 cache shared amongst all of the cores. In such a system, the coherence challenge presented in Listing 1.1 is complicated. When Thread 1 writes Address A, the new data will be stored in Core 1's L1 data private cache. When Thread 2 attempts to read from Address A, it will execute a load instruction on Core 2 which does not have access to Core 1's private L1 data cache. A coherence mechanism is requried to support maintaining coherence across these private caches. Many coherence protocols and mechanisms have been proposed in the literature [14, 23, 53, 54, 63], each of which relies fundamentally on passing the information that Core 2's copy of Address A becomes invalid after Core 1's modification of Address A (an invalidation-based protocol). Commonly, these protocols are implemented using an interconnection mechanism between the private caches on a CMP die. As core counts increase in CMPs, the design and verification of such coherence mechanisms becomes particularly complex. Specifically, the number of edge cases that must be verified and the complexity of scaling interconnect topologies as core densities continue to increase becomes difficult. It has been shown that scaling of these technologies is possible, but it is also commonly accepted that the design and verification of such mechanisms at large scales is complex. Additionally, the energy consumed by the various structures and logic required to maintain coherence states and arbitration adds to overall CMP energy consumption as core counts increase.

The e-PDEMI architecture evaluated in this dissertation is able to tolerate longer latencies in the memory hierarchy. This latency tolerance capability allows the removal of invalidation-based cache coherence with the replacement of a shared address mapped cache hierarchy. The multi-core e-PDEMI system's memory hierarchy is implemented as a single, large shared, address-mapped, banked memory hierarchy system. In such a system, there is only ever one copy of a given cache line. Thus, a memory coherence protocol is not needed by the e-PDEMI system and is removed. The e-PDEMI system could work with traditional cache coherence, but evaluation shows a negligible performance impact which does not seem to justify the additional associated complexity.

1.3 Memory Consistency Impacts System Programmability

A Memory Consistency Model is a contract between programmers and a multi-processor system specifying rules that if followed will guarantee that memory operations will be executed in a predictable fashion. There are many memory consistency models. Derek Hower [35] proposed the following descriptive comparison for discussing various memory consistency models: A memory consistency model is considered weaker than another if it allows a memory instruction ordering that another does not. Conversely, a stronger memory consistency model is one that does not allow a particular memory instruction ordering that another does allow. To support multiple platforms, programmers must write applications that properly execute on the weakest memory consistency model of the targetted platforms.

The most intuitive of such models is called Sequential Consistency. The Sequential Consistency memory model specifies that the memory operations of a given processor in a multi-processor system appear in program order to that processor. This model is the most intuitive because it means that a programmer writing a thread in a multi-threaded application can assume that each memory operation in the thread is executed in the sequence in which it was written.

Sequential consistency is usually avoided in most modern commercial systems [20, 21, 37, 70] because although it is very intuitive, it can introduce severe performance degradation [7, 20, 24, 28, 35, 37, 40, 65, 70]. To support Sequential Consistency, systems composed of out-of-order cores might need to delay memory instructions to ensure program order execution, or support a speculation mechanism that can detect if Sequential Consistency is violated, then correct and reset the architectural state, and re-execute the offending instructions. Instead, many processors use a Release Consistency model.

Release consistency is the memory consistency model supported by the well-known pthreads library [3]. Release consistency allows reordering of memory instructions' execution. To guarantee that a particular set of memory instructions on a given CPU executes before a different set of memory instructions, special synchronization instructions must be inserted into the program. These synchronization instructions are often referred to as a Memory Barrier or Memory Fence. When a CPU encounters these instructions, it cannot execute any additional memory instructions until all inflight memory instructions are retired. Although this memory consistency model can slightly slow performance during memory instruction synchronization related instructions, the flexibility to reorder memory instructions inside of barrier or fence instructions allows out-of-order processors to typically run faster. One drawback to the Release Consistncey model is that programmers must be more accutely aware of the underlying system they are programming and deeply consider the implications of possible thread interactions and other system environmental factors. Because without barrier or fence instructions, memory operations could be reordered which may or may not execute in the way the programmer intended. The extreme alternative would be to guard every load and store with a fence or barrier instruction which would then make the execution behavior very predictable for the programmer, but would significantly restrict the out-of-order pipeline's ability to reorder instructions for maximum ILP.

The e-PDEMI architecture verifies all memory operations off of the critical path, in-order, at retirement. This, in addition to an address mapped shared cache hierarchy effectively means that each e-PDEMI core provides a Sequential Consistency memory model for a multi-core e-PDEMI system. Notice that the in-order verification also allows further simplifications like the elimination of all memory ordering instructions (*e.g.*, Memfence, MemBarrier, etc.) that are required for correct execution on a release consistency system.

As discussed in Chapter 5, e-PDEMI provides this much more intuitively programmable architecture without any significant performance impact. In multithreaded applications with heavy synchronization usage, e-PDEMI can provide up to a 14% performance improvement, and on average reduces energy dissipation by 10%.

Chapter 2

Related Work

2.1 Memory Speculation

The main distinguishing feature of e-PDEMI compared to related works is that it redesigns the micro-architecture with energy efficiency as the primary design parameter.

2.1.1 Store Queue Index Prediction

SQIP [61] addresses the scaling challenges associated with maintaining the fully associative Store Queue(implemented with CAMs) in traditional out-of-order processors. They propose a mechanism for speculative indexing of the Store Queue to avoid full structure lookups. For each load, a single Store Queue entry is predicted for data forwarding. The indexed store in the Store Queue is then checked for an address match and its absolute age in program order is interrogated to determine if it is younger than the requesting load. If these conditions are met, data is forwarded. To support indexed access, two predictors were introduced. First, a StoreSet-like [19] PC indexed table identifies likely forwarding of Store Queue entries. Second a predictor based on the Exclusive Collision Predictor [71] predicts whether loads should be delayed until their producer stores commit, due to forward table mispredictions. e-PDEMI in contrast removes the LSQ, and relies on value prediction and replay.

2.1.2 NoSQ

In NoSQ, Sha *et al.* [62] extend their work to remove the store queue. Their proposal uses Speculative Memory Bypassing [51] to speculatively predict loads to either "bypass" the traditional out-of-order pipeline, or to execute as normal through the pipeline. Bypassed loads receive data from their producer stores through an additional read port on the register file rather than a fully associative Store Queue. Issued dynamic stores' target addreses are tracked in a Store Sequence Bloom filter, which is accessed by every load at rename. If a load's producer store is predicted to be in-flight, it's output register mapping is set to the physical register corresponding to the predicted bypassing store's data input.

This re-mapping is achieved using extensions to the proposed pipeline's rename and register allocation logic to properly perform the rename and track all in-flight register references (to avoid recycling a register that has been re-mapped due to load bypassing). The register for re-mapping is retrieved from a Store Register Queue that is added to track and properly index all in-flight stores' data input registers. To capture path-dependent bypassing patterns, NoSQ uses two parallel tables to predict load bypassing. The first table simply replies upon load PC addresses while the second table uses load PC addresses and path history bits. To reduce possible bypass misprediction due to narrow-store/wide-load communication, as well as pathological paths that the above predictors are unable to capture, NoSQ adds a confidence counter to each predictor entry. Confidence counters are updated as correct and incorrect predictions occur.

The removal of the Store Queue requires the re-order buffer to perform store address generation and to retrieve store data from the register file. NoSQ adds read ports to the register file and TLB for this purpose. For the re-execution verification of bypassing loads, the re-order buffer also uses an additional register file read port for this purpose. The proposed architecture in this work does not introduce any additional ports on critical structures such as the register file.

2.1.3 Fire-and-Forget

Fire and Forget [61] builds on Store Queue Index Prediction by removing the Store Queue completely [67]. Instead of a Store Queue, stores are kept in program order as normal in an out-of-order processor using the Re-order Buffer. Store results are kept in a result register added to the ROB, which facilitates the complete removal of the Store Queue. Loads never access the store queue in any way and stores use three added prediction tables to speculatively forward their data to one and only one load in the load queue. Mis-forwards and non-forwards are resolved by re-executing loads at commit time to verify their consumed data in-order against the data cache which is also updated in-order. Once a store has forwarded its value, it is allowed to “forget” the value and the load queue is obligated to store the forwarded value. It is pointed out that memory cloaking [51] could further enhance this work. While the architecture proposed in this work relies on ROB to hold the speculatively run memory instructions before they are verified, it does not add extra logic to the ROB.

2.1.4 SMDE: L0 cache

A different approach to providing a scalable memory disambiguation scheme was to remove the Load Store Queue altogether and replace it with a speculatively updated and accessed cache. Garg *et al.* proposed this approach with an “L0” cache [27]. This technique allows memory instructions to issue speculatively to the L0 cache out-of-order. Program correctness depends on in-order back-end execution whereby the speculative data consumed by load instructions is checked against data obtained in the re-execution path that is guaranteed to be correct. A fully associative fuzzy disambiguation queue is also proposed to reject loads from executing to the L0 cache if there are any older in-flight stores matching its address that have not yet executed. In this way, potential load-triggered replays are avoided. Additionally, an age ordered Memory Operation Sequencing Queue is proposed to keep the address and data for speculatively executed loads, and the address and data for stores waiting to be executed in-order in the back-end execution. These additional structures support the L0 cache in allowing memory speculation without the need for complex heuristics or various support tables to track and detect memory order violations. Because the verification path in the L0 cache is a data-based verification, spurious replays are avoided. While this is the most closely related work to e-PDEMI, SMDE does not consider energy efficiency in its proposal. For example, e-PDEMI removes the fully associative fuzzy disambiguation queue, and implements a different mechanism to avoid excessive replays. Section 4.3.2 in Chapter 5 further compares and evaluates the distinct differences between SMDE and e-PDEMI.

2.1.5 Other Related Work

Sethumadhavan *et al.* approached the area and power efficiency of CAM-based Load Store Queues by proposing late-binding [60]. Under late binding, entries in the Load Store Queue are allocated when memory instructions issue rather than when they are dispatched in the out-of-order pipeline. Akkary *et al.* proposed two-level store buffers where stores issue into the first level buffer and then are displaced into the second level buffer when the first level overflows [26]. Stores always commit from the second level buffer, but both buffers are fully associative and support forwarding. Stone *et al.* investigated partitioning the Load Store queue into three structures, distributing and address interleaving across all three [66]. These structures were an address indexed time stamp table for memory verification, a set associative cache for forwarding, and a non-associative FIFO for commit. Delayed memory dependence checking was investigated by Castro *et al.* in [15], using filters to delay memory verification until commit, and avoid many associative searches by store instruction age information in auxiliary registers. Sethumadhavan *et al.* further investigated filtering techniques using Bloom filters to avoid associative searches in the Load Store Queue [59].

2.2 Simplifying Memory Coherence

As modern multi-core architectures scale to tens or hundreds of cores, hardware based cache coherence can become prohibitively expensive and difficult to design and verify [5, 44]. Hence, the simplification or removal of coherence hardware has been studied in several works. There are many papers dealing with cache coherence and software cache coherence. These are the most related works.

2.2.1 Thread Migration Prediction

Shim *et al.* [64] approached simplification of memory coherence mechanisms by considering thread migration at instruction-level granularity to keep shared data local from thread perspective. The authors evaluate their proposal in the context of a Non-Uniform Cache Architecture (NUCA). In a NUCA based system, the address space is divided amongst the cores in the system such that every valid address is assigned a *home node* which is responsible for caching the related data. In order to read or write data in a remote core, the accessing core must send a request over an interconnection network. The authors of the thread migration work instead propose to send the *context* of a thread which includes that thread's architectural state when a thread requests a remote memory access to the home node to begin executing that thread as a *guest* of that home node. They allow for every core to have a native thread and a guest thread. Native threads must be evicted to allow execution of a guest thread, and if guests are evicted they must return to their native core. This mechanism prevents threads from having to access remote memory structures, however the authors note that moving cores' architectural states across the interconnect can introduce significant latency overheads. They thus propose a decision making algorithm relying on a direct-mapped structure that tracks the PC address of potentially migratory memory access instructions which can determine when a thread migration is feasible from a performance standpoint. They then investigate potential thrashing cases in their migratory predictor and suggest additionally keeping track of how many times a core has contiguously accessed a set of remote memory locations. They present their best case average results as a 24% speedup when waiting for a core to make 3 contiguous accesses to a remote memory structure before migrating its thread. The energy consumption of the proposed system was not evaluated.

2.2.2 Synchroniztion Based Cache Coherence

Lin *et al.* [48] described a system that uses specialty cache policies with memory lock and barrier instructions to support cache coherence and Release Consistency without the traditional cache coherence protocol. The system uses blocking caches supporting a single outstanding cache miss with a write-through policy for cache hierarchy synchronization to avoid coherence messages. The system relies on memory barrier instructions to guarantee value propogation to a shared L2 data cache coupled with the Scope Consistency memory model ScC [36], which requires programmers to specify shared or private scopes for data segments. Energy consumption was not considered in their presented results.

2.2.3 Software Managed Cache Coherence

Zhou *et al.* [72] advocated that a software-based coherence scheme is a more practical approach for design, verification, and implementation flexibility. They point out that hardware-based cache coherence can be prohibitively expensive and very difficult to verify when core counts increase to several tens of hundreds of cores [5,44]. They note many emerging workloads have low data sharing and hardware coherence may be overkill. They also argue that future architectures will be designed for execution of multiple operating systems at once. As such, these architectures would benefit from having independent coherence domains. Since most hardware mechanisms do not support independent coherence domains, the authors suggest that a software based coherence mechanism would better be able to support the dynamic allocation needs of multi-workload executions, as various processes could dynamically need more or less memory, effectively increasing or decreasing the size of their respective coherence domains. They propose a hypervisor-like layer that intercepts system calls to provide independent coherence domains. A given coherence domain provides coherence at the virtual page level. One thread is responsible to maintain a "golden copy" of a given page while sharer threads update local copies. At synchronization points, those copies are merged back into the golden page. Traditional virtual memory protection mechanisms and access modes (*Invalid, Read-only, Read-Write*) are relied upon for coherence. The authors add an additional type qualifier keyword *shared* to the standard C/C++ languages to inform the proposed system of data that the proposed coherence mechanism should manage. By default, all data is considered private unless specified otherwise by the programmer. The authors compare their software-based coherence proposal with the hardware coherence in a 32-core system for the well-known Blackscholes and Art benchmarks. They find relative slowdowns for software-based cache coherence to be only 12% and 7% for those benchmarks respectively. Energy consumption of their proposed system was not evaluated.

2.2.4 OS-Based Cache Coherence

Fensch and Cintra [25] proposed integrating coherence into the operating system. A hybrid hardware/software approach using a NUCA cache organization was suggested where a cache line can only exist in one memory structure. Cores could be required to make remote accesses, but coherence was not needed. The authors then added ports to the L1 data cache and TLB to support remote accesses. The proposed system evenly partitioned the virtual memory space across the system and sent requests to the L1 data cache and TLB as well as to a "MAP" structure that functioned like a directory. The OS and system page tables were modified to support tracking of where a given virtual page was located on the system. Each memory request on a given processor would be simultaneously sent to the local TLB and L1 data cache as well as the MAP. If the MAP indicated a remote access, the local TLB and L1 Data cache accesses were cancelled. Global visibility of stores was guaranteed using memory barrier instructions and an L1 write back policy. The OS intercepted store instructions, marking the first write to given virtual pages. Subsequent reads of a page from other processors were allowed, but writes would be stalled by the OS and create a new MAP entry. MAP entries were required to be invalidated on lock acquires, and caches had to be invalidated on memory barriers. The authors note that lock or barrier intensive benchmarks such as *raytrace* and *ocean* respectively, experienced significant performance overheads. On average, the authors found their proposed coherence scheme to have a performance degradation of 16% with respect to an aggressive directory-based coherence mechanism which the authors found impressive given the aggressive directory-based baseline and their proposal's simple hardware support. Energy consumption was not evaluated for this system.

2.3 Supporting Sequential Consistency

Sequential Consistency is the most intuitive memory consistency model. Most programmers implicitly assume sequential consistency when they design their applications by considering that each successive line of code is executed after the line before it. Since out-of-order processors strive to extract instruction level parallelism (ILP) from the program by executing instructions out-of-order, this assumption is not always correct. Data races, livelocks, and deadlocks are some of the potential problems that can result from assuming Sequential Consistency on a system with a more relaxed memory consistency model. Programmers must then add additional complexity to their multi-threaded programs such as locks, memory fences and memory barriers to guarantee their programs' proper execution. The following works attempt to abstract that complexity away from the programmer so that systems appear Sequentially Consistent.

2.3.1 How to Make a Multiprocessor Computer That Correctly Executes Multiprocessor Programs

Leslie Lamport is credited with first defining Sequential Consistency [45], using the code segment shown in Listing 2.1 to illustrate the concept. Lamport noted that as long as only

```
Process 1:
  a := 1;
  if b = 0 then critical section:
    a := 0;
  else ...
Process 2:
  b := 1;
  if a = 0 then critical section:
    b := 0;
  else ...
```

Listing 2.1: Mutual Exclusion Example Algorithm

one of the processes' critical sections is executed at a given time, then the multiprocessor system running both processes is considered sequentially consistent. He first pointed out that if process 1 were allowed to execute the "a:=1" and "fetch b" memory operations in *swapped* fashion with respect to program order, although execution on processor 1 would still be correct due to no read after right dependency between those operations, both processes could enter into their critical sections simultaneously and Sequential Consistency would not be maintained. Lamport thus first required that memory operations from a particular processor be issued to the memory hierarchy in program order. A multiprocessor system executing the code in listing 2.1 could also violate Sequential Consistency if memoy elements did not process memory requests in the order in which they were received. Lamport shows the following sequence of steps:

1. Processor 1 sends the "a:=1" request to its port in memory module 1. The module is currently busy executing an operation for some other processor.

2. Processor 1 sends the "fetch b" request to its port in memory module 2. The module is free, and execution is begun.

3. Processor 2 sends its "b:=1" request to its port in memory module 2. This request will be executed after processor 1's "fetch b" request is completed.

4. Processor 2 sends its "fetch a" request to its port in memory module 1. The module is still busy.

At the end of the steps above, there were two operations pending execution. If processor 2's "fetch a" operation was performed first, then both processes could have entered their critical sections at the same time which would break Sequential Consistency. This error could only occur if the two requests pending in module 1 were executed out of the order in which they were received. Lamport thus proposed the second requirement for Sequential Consistency: memory elements must only process requests to the same datum in the order in which they were received. Thus it was shown that a multiprocessor system following the two requirements defined by Lamport would be Sequentially Consistent.

2.3.2 BulkSC

BulkSC was proposed by Ceze *et al.* [16] to utilize the Bulk [17] set of hardware mechanisms that were originally proposed to support Transactional Memory(TM) and Thread-Level Speculation(TLS) by adding checkpointing hardware to cores in a multi-core processor. By checkpointing architectural state and memory accesses, Bulk is able to commit groups of dynamic instructions known as "Chunks". Chunks can then speculatively write cache lines in a seperate memory structure that holds checkpoint state until a Chunk is determined safe for commit.

A Bloom Filter-based Bulk Disambiguation Module (BDM) checks unique Chunk signatures that were composed of accumulated addresses for memory collisions between Chunks. If a collision was detected, a Chunk could be squashed and re-executed. The authors extended the Bulk mechanism to support Sequential Consistency by enforcing two rules: 1. Updates from Chunks were not visible to other Chunks until Chunk committment. 2. Loads from a Chunk must return the same value as if the Chunk was executed at its commit point. These rules ensure that cores on the system all observe the global memory state update in the same order, and Chunks from individual cores maintain program order. The author's proposal allowed individual cores to reorder instructions in any way they are able just as in a uniprocessor system. Sequential Consistency in the proposed system relied upon Bulk's supporting arbitration structures to determine any possible collisions that if committed could violate Sequential Consistency.

In addition to the BDM, the authors also added an Arbiter module to determine the commit ordering of chunks from various cores. For small core counts, a single arbiter was found acceptable, but for larger systems a distributed arbiter was proposed. The authors also noted, that their proposal required extensions to a traditional cache coherence directory mechanism to support BulkSC's Chunk signatures for collision detection. These extensions enabled cache directories to expand signatures into their constituent addresses, forward signatures to relevant caches for address disambiguation, and to conservatively disable directory entries of all lines written by a committing Chunk until the new Chunk values were visible to all cores on the system to preserve Sequential Consistency. The authors proposed optimizing their signature-based collision detection by excluding statically private data in a given core from generated signatures by adding an attribute to virtual memory pages in software. They further optimized for private data by considering dynamically private data updated between Chunks on the same core. Dynamically private data was managed using an additional dedicated signature for private data, and a *Private Buffer* in the BDM that could keep track of older copies of cache lines. In this way, dynamically private data could be updated in the data cache while maintaining Sequential Consistency, and core local data coherence amongst dynamic inflight instructions was maintained by keeping the older copies of cache lines available in the Private Buffer when needed. The authors found that their most optimized proposed system performend equivalently to a baseline Release Consistency system. The authors did not evaluate the energy consumption of the proposed system.

2.3.3 Sequential Consistency in Distributed Systems

Mizuno *et al.* Developed an analytical framework to test systems for sequential consistency and then proposed and verified a new protocol of their own supporting Sequential Consistency. [49] The authors first pointed out that previous works [9, 10, 12] all relied on a broadcast-based invalidation protocol to support sequential consistency. Mizuno *et al.* proposed a new protocol that did not rely on broadcast invalidations and instead relied on a combination of shared global memory and private local memory. Additionally, the proposed system added a bit vector to each cache line indicating whether the values stored in that cache line should be read locally or globally by a particular processor. In this way, global visibility of write operations was controlled such that Sequential Consistency was preserved. The ePDEMI architecture does not add any additional information to cache lines or cache state data, nor does it need to track the global visibility of writes in order to provide Sequential Consistency.

2.3.4 Implementing Sequential Consistency in Cache Based Systems

Sarita Adve and Mark Hill [8] disproved previous assertions that Strong Ordering [24] guarantees Sequential Consistency because propagation of memory write operation updates across a multiprocessor system could experience non-uniform latency. They further pointed out that the additional assertion that Sequential Consistency could only be achieved with *one at a time* memory access designs proposed in [57, 58] was too conservative. Those previous works proposed issuing accesses in program order, and no memory access was issued until the previous access was globally visible to all processors in the system. Adve and Hill relaxed that requirement to allow overlapping of some memory requests by introducing six memory operation conditions:

1. Accesses are issued in program order.
2. All processors observe writes to a given location in the same order.
3. An access cannot be issued by a processor if a previous acccess from the same processor has not been globally performed.
4. Write operations must be globally performed in sequence so that if two different processor writes(1 and 2) go to the same cache line, and write 2 is waiting on previous writes from its processor, write 1 must wait until write 2 completes. Similaly, if write 1 were instead a read, it would still wait for write 2 to complete before returning its data.
5. If one processor issues a read while still having several outstanding writes, and another processor issues a write to the same cache line as the read, the write must wait until all of the first processor's pending writes issued before the read are globally visible.
6. A read issued by a processor while some of its previous writes are not globally performed should return the last value written on any copy of the accessed line, where last is defined by the order ensured by condition 2.

The authors proposal was then further described based on a directory-based, writeback invalidation cache coherence protocol. For write operations, the proposed system relied on parallel sending of a cache line to the requesting processor along with invalidation messages to the other processors in the system. Each processor was assigned a counter to record how many processors had received an invalidation message (indicating global visibility of the write), and each cache line was augmented with a *reserved* bit indicating if the local cache should allow remote processors to access a modified cache line. This additional tracking information supported the prevention of remote processors consuming newly updated data too early with respect to Sequential Consistency. During the course of normal execution, if a processor issued a memory access that resulted in the need to evict a cache line that was marked reserved, the processor would be stalled until pending memory requests to that cache line had been globally performed and the line was no longer marked reserved. Because of this potential need to stall processors, the authors also noted that workoads with heavy data sharing could experience a slowdown due to excessive outstanding remote writes. The authors concluded by offering a qualitative analysis showing algorithmically where the *one at a time* memory access design would stall on remote writes waiting for cache lines to return and thus where their proposal could overlap reads and writes to other cache lines. ePDEMI provides Sequential Consistency without needing to stall cores to guarantee global visibility of writes.

2.3.5 Is SC + ILP = RC

Gniady *et al.* [29] show a way for Sequential Consistency(SC) Memory Models to achieve equivalent performance to Relaxed Consistency (RC) Memory Models. They argue that SC Memory Models require: 1. "Full Fledged speculation" whereby loads and stores are allowed to execute completely out of order with no constraints. 2. maintaining a large speculative state allowing a processor to entirely replay both architectural and cache hierarchy state back in the event of a mis-speculation. 3. A fast common case in which SC Memory Models do not introduce considerable overhead to execution time on average. 4. Infrequent replays so that SC Memory Models do not heavily rely on restoring processor state due to mis-speculation thus eroding any performance advantage. They allow store instructions to speculatively retire out of order by using a read-modify-write operation to the L1 data cache. To guarantee correct processor and data cache state after a rollback, Store History Queue(SHiQ) and Block Lookup Table structures are proposed to keep all relevant processor and memory hierarchy state including physical register number, old register rename map, old value of destination register, etc. for each retired instruction as well as the tags and previous state of any speculatively modified cache blocks. The authors also find that when the Re-order buffer is increased from 64 entries to 1,024 entries, the number of rollbacks increases significantly causing the RC Memory Model system performance to increase over the SC Memory Model system performance. In e-PDEMI, all speculative data isstored in a value prediction cache (VPC) and a simple mechanism to limit the number of replays is implemented.

Chapter 3

Sequential Memory Consistency Alternatives

This chapter presents an overview of several memory consistency models as alternatives to Sequential Memory Consistency. Some of the consistency models presented can support Sequential Consistency through the use of special synchronization instructions that enforce serialization between different classes of memory instructions. Others violate Sequential Consistency in an effort to provide compilers more flexibility to re-order instructions accessing different addresses for performance optimization.

Relaxed memory consistency models that allow the re-ordering of memory instructions accessing different addresses can allow compilers not only to re-order those instructions as needed, but also to overlap them to hide latency. One common design, that is usually accompanied by Release Consistency, allows an out-of-order processor in combination with non-blocking caches [42] to issue multiple memory instructions even in the presence of outstanding misses. This behavior is more detailed in Section 3.2, but the key point is the support for hiding the latency of outstanding memory operations that may have missed in a given level of the memory hierarchy is a major enabler of high performance in out-of-order processors. Finally, it is important to note that the multi-core e-PDEMI implementation allows compilers total flexibility to re-order memory instructions (assuming programatic correctness is preserved), and relies upon non-blocking caches to sustain the high performance of the baseline architecture as discussed in Chapter 5 Section 5.1.5.

3.1 Weak Ordering

The Weak Ordering memory consistency model [24] was proposed by Dubois *et al.* This memory consistency model classifies memory instructions into two classes: *data* operations, and *synchronization* operations. It is left to the programmer to identify which memory locations (variables) belong to which class. Weak Ordering only enforces data memory operation ordering across synchronization memory operations. Any data memory operations occurring between synchronization operations can be re-ordered in any way. Thus, a synchronization memory operation may not execute to the memory hierarchy unless all previously issued load and store instructions complete their execution in the memory hierarchy. Store memory instructions appear atomic to the programmer under Weak Ordering and as such, load instructions are not allowed to consume data from loads that are pending memory hierarchy execution in any local buffer (store forwarding disallowed).

3.2 Release Consistency

Release Consistency refers to a memory consistency model that relies upon a paradigm of classifying memory operations into three categories. The first category is "normal" operations, and refers to most load and store operations. The other two categories of memory operations are referred to as "acquire" operations and "release" operations. An acquire memory operation prevents future normal memory operations from executing until the acqurie operation is retired and thus the issuing processor has acquird ownership of the related memory space. A release operation cannot retire until all previous normal memory operations have retired. Once those previous normal memory operations have retired the release operation may retire, thus releasing ownership of the related memory space from the issuing processor. When a processor issues a load instruction to the memory hierarchy, the load instruction is allowed to bypass pending writes and releases but cannot bypass an aquire. When a processor issues a store instruction to the memory hierarchy, the store intruction will only be stalled if the target memory element's write buffer is full, or another processor has aquired the related memory space. When a processor issues an acquire memory operation to the memory hierarchy, that processor cannot issue additional memory instructions until that acquire operation retires. When a processor issues a release memory operation to the memory hierarchy that release operation cannot be retired until all previous stores and releases are retired. In this way, a programmer writing a multi-threaded application can enforce the ordering of memory operations between processors on a multi-processor system by using the acquire and release operations as memory sychronization operations.

Within the Release Consistency memory consistency model, there are two sub-classes called RCsc and RCpc. These sub-classes of Release Consistency are differentiated by how they execute memory instructions related to synchronization. RCsc enforces Sequential Consistency among sychronization memory instructions, while RCpc provides Processor Consistency between sychronization memory instructions.

3.3 Processor Consistency

Processor Consistency [28] considers memory consistency from the perspective of a single processor. Load instructions from a given processor are allowed to consume the data from any local or remote store instruction that occurs, either before or after the load instruction, in the program order, even before the store becomes globally visible. To enforce ordering between memory instructions, Processor Consistency requires the use a of read-modify-write instruction. However, this technique only applies to load instructions because Processor Consistency does not guarantee atomicity between the read and write in a read-modify-write instruction. The main advantage of Processor Consistency is to hide the memory latency associated with store instructions by allowing load instructions to execute out-of-program order with respect to previous store instructions.

3.4 Total Store Ordering SPARC V8

The Sun Microsystems SPARC V8 processor implements a memory consistency model known as Total Store Ordering(TSO) [2]. TSO applies to uniprocessor and multiprocessor variants of the SPARCV8. TSO guarantees that store, FLUSH, and atomic load-store instructions are executed by memory serially in an order that conforms to the order that those instructions were issued by all procesors. On a given processor, local load instructions are allowed to consume local store instructions that have not yet become globally visible to other processors. However, on the same given processor, local load instructions cannot consume data from remote store instructions before they become globally visibile. Also, load instructions block their issuing processor until they are resolved. If the programmer for a TSO based system wishes to enforce program order between a pair of load and store instructions, either the load or the store muse be the read or write of a read-modify-write instruction. Alternatively, a *dummy* read-modify-write instruction can be inserted between any two memory instructions to force sequentiality between them. A *dummy* read-modify-write instruction would read a value from memory and write the same value back to the same location.

3.5 Partial Store Ordering SPARC V8

The Sun Microsystems SPARC V8 also supports Partial Store Ordering (PSO) which guarantees that store, FLUSH and atomic load-store instructions of all processors appear to be executed by memory serially but not neccessarily in the order that they were issued from a processor. Unlike TSO, PSO allows store instructions to different addresses to be overlapped and potentially re-ordered. Local forwarding from stores that are not yet globally visible to consumer loads is still supported, and outstanding loads still block their issuing processor. PSO provides a STBAR instruction that functions as a Store Barrier. On encountering an STBAR instruction, a processor cannot issue any additional store instructions to the memory until all pending stores have completed execution in the memory hierarchy. This functionality can be implemented using a FIFO structure to buffer all not-yet-issued stores and a counter. The counter is incremented when a store instruction is issued to the memory hierarchy, and decremented when its execution in the memory hierarchy is completed. When as STBAR instruction is encountered, if the counter value is non-zero, new store instructions are buffered in the FIFO structure, but not issued to the memory hierarchy. Once the counter has been decremented to zero, stores from the FIFO structure may be once again issued to the memory hierarchy. Store instructions that occur between STBAR instructions may be re-ordered in any way. Atomic load-store instructions such as SWAP and LDSTUB are treated by the PSO mechanism as both a load and a store. The instruction is inserted into the store FIFO, but it also blocks the processor like a normal load until it is resolved.

3.6 Relaxed Memory Ordering SPARCV9

The Sun Microsystems SPARC V9 supports Relaxed Memory Ordering (RMO) [70] which places no ordering constraints on memory instructions beyond uni-processor consistency (programmatic correctness). SPARC V9 instead supports the MEMBAR instruction that functions as a memory barrier. When a memory barrier is encountered, all memory instructions issued prior to the MEMBAR must be completed. Memory instructions occuring between two MEMBAR instructions may be re-ordered in any way. Further, the MEMBAR instruction can be encoded with a 4-bit field to support any combination of specific memory ordering constraint, such as ordering only loads with loads, or loads before stores, or stores before loads, or stores with stores, or any union of those cases including all of them.

3.7 IBM-370

The IBM-370 system did not allow loads to consume the value of store instructions before they were globally visible, meaning that store-to-load forwarding in the processor was not allowed [1]. SYNC instructions were used much like compare and swap instructions to enforce read after write dependencies. Sequential Consistency was achievable on the IBM-370 by placing a serialization instruction after every memory instruction in a given program to enforce program order execution of memory instructions. Even without enforcing Sequential Consistency, the IBM-370 did not provide much flexibility to the compiler to reorder memory instructions since loads could not consume store data until stores were globally performed.

3.8 Alpha

The Digital Alpha processor [65] did not explictly enforce any memory operation order. Memory operations were free to be re-ordered in any way by the compiler or processor. Such a relaxed memory consistency model allowed compilers flexibility to re-order memory instructions for performance optimization. Additionally, the processor could re-order and/or overlap memory instructions to hide memory hierarchy latency to further increase performance. Programmers were given two instructions to force memory instructions' execution order. The first was the WMB or Write Memory Barrier instruction which functioned very similarly to the STBAR instruction from the PSO memory consistency model in the SPARC V8. Once the Digital Alpha processor encountered a WMB instruction, no additional store instructions could be issued until all outstanding store instructions' execution in the memory hierarchy was completed. The second memory ordering instruction the Digital Alpha processor provided was the MB instruction or Memory Barrier instruction. Once an MB instruction was encountered, no more memory instructions *of any type* could be issued to the memory hierarchy until all outstanding memory instructions completed their execution in the memory hierarchy.

3.9 PowerPC

The PowerPC architecture provides the SYNC intruction which is similar to the MEMBAR instruction from SPARC V9's RMO memory consistency model. The SYNC instruction will guarantee that memory operations that occur on either side of it are executed in sequence with one exception. Two reads to the same address can be executed out of program order even if a SYNC instruction is placed between them. To enforce an ordering between two load instructions accessing the same address on the PowerPC, a read-modify-write instruction is required. Further write instructions are not guaranteed atomicity when executing in the memory hierarchy. As with load instructions, store instructions must become the write operation of a read-modify-write instruction in order to guarantee atomicity of store instructions.

3.10 e-PDEMI Supports High Performance Sequential Consistency

Because e-PDEMI performs memory speculation with the VPC discussed in Chapter 4 Section 4.1.3.1, it provides performance equivalent to Release Consistency. e-PDEMI is designed to execute code compiled for the Release Consistency memory consistency model. As such, the compiler is free to re-arrange instructions in any way to improve performance. Further, e-PDEMI then executes memory instructions out-of-order to the VPC which enables greater flexibility to hide memory latencies related to outstanding inflight memory instructions. Finally, because each e-PDEMI core only commits store instructions and verifies load instructions to the memory hierarchy in program order, and each non-blocking cache processes each memory instruction to the same address in the order they were recieved, e-PDEMI maintains the high performance of Release Consistency while enforcing Sequential Consistency.

Chapter 4

The e-PDEMI Architecture

4.1 e-PDEMI Single-Core Architecture

The e-PDEMI single-core architecture re-conceives of memory speculation with energy consumption as the first design consideration while preserving the high performance provided by a modern super-scalar out-of-order processor.

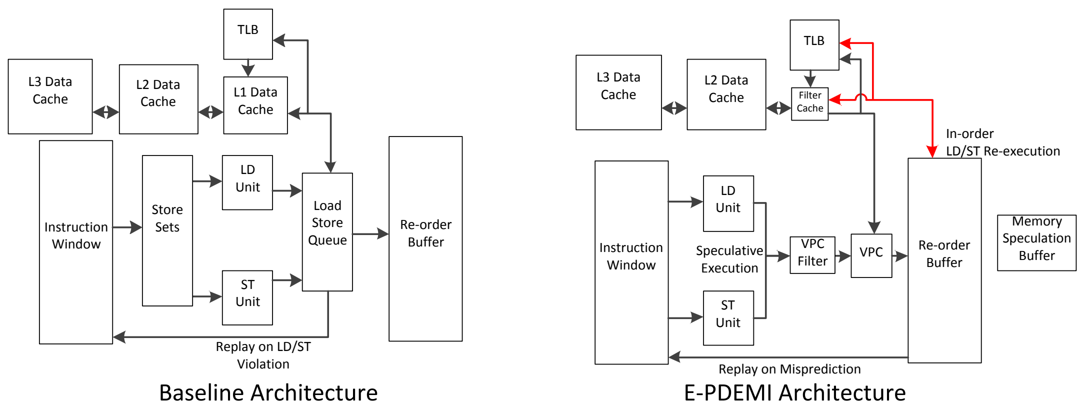


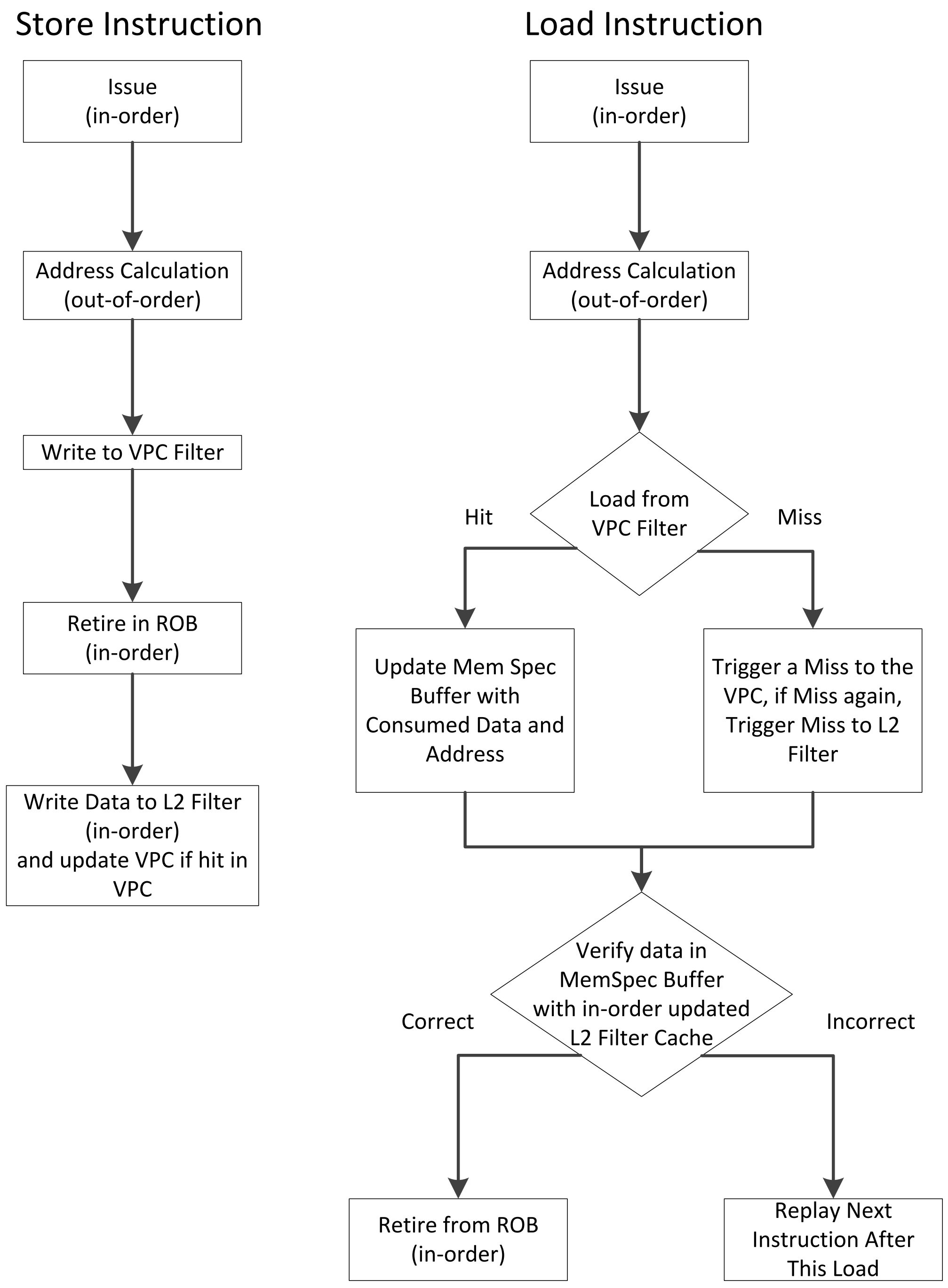
Figure 4.1: Baseline and Proposed Architectures

4.1.1 Baseline Architecture

Figure 4.1 shows the baseline architecture that e-PDEMI is proposed to modify. The baseline processor is an out-of-order processor with an instruction window that can dispatch one load and one store instruction per cycle to dedicated load and store units. The load and store units can then each execute one instruction per cycle by accessing the traditional Load Store Queue. The Load Store Queue orders store instructions to be committed to the memory hierarchy in order, performs appropriate store-load forwarding and detects memory order violations resulting in replays. The Load Store Queue also supports forwarding of unaligned store data to consumer load instructions. Additionally the Load Store Queue can issue one load and one store per cycle to the L1 data cache and TLB. The L1 data cache is virtually indexed and physically checked, meaning that the TLB is accessed in parallel with the tag bank and the successive data bank access is serialized. Table 4.1 lists the sizes and other relevant parameters for each of these structures.

4.1.2 Store Sets for Memory Speculation Correction

The baseline system uses StoreSets [19] to avoid replay loops resulting in slow forward progress. Every load and store instruction in the baseline architecture is assigned to a Store Set identified by its StoreSet ID. The StoreSets implementation requires the addition of an Store Set ID table to hash load and store instruction PC addresses to StoreSet IDs. If a memory ordering violation is detected by the Load Store Queue, the store that executed too early is identified by its StoreSet ID and the replayed load instruction's PC address is statically mapped to the store's StoreSet ID. Because the instruction rename stage of the baseline architecture is performed in program order, it is guaranteed that the rename logic will see the load instruction before the store instruction during the ensuing replay. The rename logic will see that the load instruction's PC address is already associated with a valid StoreSet ID and simply mark that load as the last fetched instruction for that StoreSet ID in an additional table called the Last Fetched Instruction table. The Last Fetched Instruction table is indexed by StoreSet ID. When the store instruction enters the rename stage in the baseline architecture's execution pipeline, the rename logic will also see that the store instruction's PC address also belongs to a valid StoreSet ID. The Last Fetched Instruction table will be indexed with that StoreSet ID and the load instruction will be found. The rename logic will then create a dependency chaining the store instruction to the load instruction such that the store instruction cannot be executed in the out-of-order pipeline until the load instruction has been retired. In this way, these instructions that previously caused a replay due to a read-after-write dependency are forced to execute serially to avoid a recurrent replay.



Store Instruction
Issue (in-order)
Address Calculation (out-of-order)
Write to VPC Filter
Retire in ROB (in-order)
Write Data to L2 Filter (in-order) and update VPC if hit in VPC
Load Instruction
Issue (in-order)
Address Calculation (out-of-order)
Load from VPC Filter
Hit
Miss
Update Mem Spec Buffer with Consumed Data and Address
Trigger a Miss to the VPC, if Miss again, Trigger Miss to L2 Filter
Verify data in MemSpec Buffer with in-order updated L2 Filter Cache
Correct
Incorrect
Retire from ROB (in-order)
Replay Next Instruction After This Load

Figure 4.2: e-PDEMI Flowchart for Load and Store Instructions

4.1.3 Proposed e-PDEMI Architecture

The right side of Figure 4.1 illustrates the proposed architecture and Figure 4.2 shows the pipeline flow of store and load instructions in the e-PDEMI architecture. e-PDEMI targets an architecture that can reduce system power by completely removing the Load Store Queue and StoreSets.

4.1.3.1 Value Prediction Cache

e-PDEMI includes a virtually indexed and virtually checked Virtual Predictor Cache (VPC) which avoids the access time of the TLB. Memory instructions are allowed to issue directly to the VPC out-of-order without any intermediary memory disambiguation, forwarding prediction, virtual to physical address translation, or other support logic. Stores are dispatched from the Store unit out-of-order, and speculatively update a decoupled VPC Buffer. The VPC Buffer does not ever displace data into the VPC. Loads are dispatched from the Load Unit out-of-order to the VPC Buffer first. If the VPC Buffer hits on the load's address, the load speculatively consumes the data contained in the VPC Buffer. The VPC Buffer is able to forward unaligned store data to consumer load instructions. If the VPC Buffer misses on the load's address, the load then accesses the VPC. If the VPC hits on the load's address, the load speculatively consumes the data contained in the VPC. If the VPC misses on the load's address, it triggers a miss request to the L2 Filter cache and obtains the requested data. It is important to note that the VPC replaces the L1 data cache and as shown in Section 4.2 is the same size as the L1 data cache in the baseline architecture. The only additional structures added are the small filter cache in front of the L2 cache and the small memory speculation buffer storing inflight memory instructions' addresses and data for off critical path verification.

4.1.3.2 L2 Filter Cache

The L2 Filter Cache is implemented as a phase cache [33] to provide energy savings by alleviating the L2 cache of additional re-execution accesses. It is important to note that the VPC never displaces speculative data to the L2 Filter cache. Further, the data in the L2 filter cache is always correct. The correctness of the L2 filter cache and verification of the speculative data consumed by loads are both handled off of the critical path. At retirement, stores are issued in-order from the re-order buffer and update a small filter cache placed before the larger L2 cache. This is to avoid accessing the big L2 structure for every store. The store also updates the VPC. To avoid increasing register file pressure, or adding register file ports, e-PDEMI includes a small memory speculation buffer that stores the data and address of each speculatively executed memory instruction. The memory speculation buffer is indexed by memory instructions' addresses.

4.1.3.3 No Write Allocation for VPC

Because store instructions are issued in order to the L2 cache through the filter cache, the content of the L2 cache is always correct. Additionally, because store instructions update the VPC in order at retirement, the VPC is considered mostly correct. To reduce the energy cost of re-executing store instructions to the memory hierarchy, write-miss requests are not allowed to be sent from the VPC to the filter cache. Therefore, when a store instruction is issued to the VPC at retirement, if the associated cache line is not present in the VPC, the request is dropped and no further action is taken. Because the VPC is merely used to predict values, it is not necessary for the store to write its data into the VPC. If a consumer load executes speculatively some time after the store such that the VPC Buffer no longer contains the store's data, then the VPC will trigger a miss request to the L2 filter cache and the correct, sequentially updated cache line will be provided. As can be seen in Section 4.3.4, the performance impact of this policy is negligible and the single-core e-PDEMI architecture is able to maintain the average energy reduction of approximately 16.4% discussed in 4.3.1.

4.1.3.4 In-Order Verification

When load instructions are re-executed at retirement, the data retrieved from the L2 Filter cache is compared against the data the load speculatively consumed during out-of-order execution. If the data matches, the L2 Filter Cache signals the re-order buffer that the load may retire, and if the data does not match, it signals a flash clear of the VPC Buffer, and signals the re-order buffer that a replay from the instruction immediately proceeding the load must be triggered. Because the triggering load receives the correct data from the L2 filter cache, it does not require replay. This is in contrast to Load Store Queue implementations where the load itself would need to be replayed. As a result, the proposed architecture can guarantee forward progress. As such, the StoreSet mechanism is not needed, and is therefore removed.

4.1.3.5 Avoiding Excessive Replay

In the place of StoreSet predictors, e-PDEMI simply implements a counter that can enforce periods of memory instruction serialization when replays begin to erode forward progress. Note that in benchmarks such as $bzip2$ there are replay interactions, particularly in loops, where a replay due to a particular load-store pair can trigger the replay of a nearby load-store pair, and although forward progress is guaranteed, it becomes very slow. To quickly combat this problem, if the proposed architecture detects forward progress below a threshold (less than 200 instructions between replays), all memory instructions are serially executed to the VPC for a set number of instructions. These simplifications may allow architects to easily scale up the surrounding structures (Instruction Window, Load and Store Units, re-order buffer etc.) to extract increased instruction level parallelism by supporting more in-flight memory instructions without complex scaling challenges. However, the focus of the ePDEMI architectural design is to sustain performance equivalent to the baseline architecture with lower energy per instruction while removing the complexity of coherence and supporting the simplicity of a sequential consistency memory model.

4.1.3.6 No Cache Checkpointing

Because store instructions are committed in-order and the VPC cannot displace speculative data to lower levels of the memory hierarchy, the e-PDEMI architecture does not pollute lower cache levels. As a result, replays remain decoupled from the memory hierarchy and do not require keeping checkpoints, or additional state based storage in order to repair the memory state. Avoiding memory pollution saves the dynamic energy associated with accessing the memory hierarchy for cache line displacements as well as the more costly multiple writes required to restore cache state during a replay.

4.1.3.7 Off The Critical Path

Because access to the main memory hierarchy is off the critical path, the e-PDEMI architecture is able to tolerate longer memory latencies. e-PDEMI is thus able to tolerate the additional latency associated with adding the small L2 Filter cache to the memory hierarchy, implementing the L2 Filter cache as a phase cache [33], and verifying all memory instructions at retirement. Section 4.3.6 presents performance results of varying L2 filter cache sizes and shows that the e-PDEMI architecture's performance results are fairly insensitive to the size of the L2 Filter Cache.

The e-PDEMI architecture simplifies the execution pipeline by removing the StoreSet predictors and Load Store Queue. The VPC replaces the L1 data cache and only a small L2 filter cache and memory speculation buffer are added.

4.1.4 Virtual Synonyms and Security

Unlike traditional caches, the VPC in the e-PDEMI architecture does not suffer from classic problems implied by virtual indexing and virtual tag checking. Due to its function as a predictor cache, the VPC is not required to handle virtual synonyms correctly. Consider the following example: A single process is running on the e-PDEMI architecture and obtains two virtual addresses that map to a single physical address. At an early point in time, the process modifies the contents pointed to by the first virtual address in the VPC. Some time later, the process modifies the second virtual address in the VPC. Finally, the process attempts to read from the first virtual address and receives the original data from the first virtual address in the VPC, however that data is now stale and not correct. Since each of the stores performed by the process would have updated the virtually indexed and physically checked L2 Filter cache at retirement in-order, when the load retires in the e-PDEMI system it would check the data it consumed against the data in the virtually checked and physically indexed L2 Filter Cache (where there are no virtual synonyms) and would find that it consumed the wrong data. The correct data received from the L2 Filter cache would be committed to the architectural state of the system and a replay would be triggered from the next instruction guaranteeing correct forward progress.

Another challenge that the e-PDEMI system could face, due to virtually indexing and checking the VPC, is process address space violations due to context switching. For example, if an e-PDEMI system were running a process known to store a password in virtual address 100, a malicious process could attempt to access virtual address 100 in a running loop hoping to aquire the first process's password. The malicious process could then attempt a brute force attack by continually accessing virtual address 100 and attempting to use the data loaded as the password. If the password were accepted, the malicious process would branch. Interestingly, in the e-PDEMI system, each time the malicious process attempted this load, it would fail

Baseline Architecture

Issue Width	4 Instructions
Re-order Buffer	256 Instructions
Load Queue	48 Loads
Store Queue	32 Stores
Branch Predictor	OGEHL
BTB	4K entries - 4-way
Integer ALUs	4 units
Floating Point ALUs	4 units
L1 TLB	64 entries / Fully Assoc. / 1 cycle
L1 Data Cache	16KB / 4-way / 4 cycles
L2 Cache	512KB / 16-way / 7 cycles
L3 Cache	8MB / 32-way / 14 cycles

Table 4.1: Baseline architectural parameters

verification at retirement, because the address would be translated to a physical address in the L2 Filter cache, which would yield a value from the malicious process's address space instead of the first process's address space. The malicious process could however read the branch predictor performance counter in the e-PDEMI system, and if it found that the last branch had been taken, then it would know the correct password had been found. In applications where such a security challenge may be present, the OS could perform a flash clear of only the VPC Buffer and main VPC on context switches.

4.2 Experimental Setup

This work used a simulation setup similar to TASS [38]. It modifies SESC [55], and uses QEMU [11] as the functional emulator executing ARM instructions. It uses a modified version of McPAT [47] for power estimation. Table 4.4 shows the TASS parameters used for single-core evaluation of the e-PDEMI architecture. Table 4.1 lists the parameters used to simulate each core of the baseline architecture, and Table 4.2 lists the parameters used to simulate each core of the proposed architecture. The single-core implementations of the baseline and e-PDEMI architectures were evaluated by running approximately 5 billion instructions from 16 SPEC2006 [34] benchmarks shown in Table 4.3.

e-PDEMI Architecture	
Issue Width	4 Instructions
Re-order Buffer	256 Instructions
Branch Predictor	OGEHL
BTB	4K entries - 4-way
Integer ALUs	4 units
Floating Point ALUs	4 units
L1 TLB	64 entries / Fully Assoc. / 1 cycle
VPC Cache	16KB / 4-way / 4 cycles
VPC Buffer	64 entries
Mem Spec Buffer	48 Load entries, 32 Store entries
L2 Filter Cache	4KB / 16-way / single-core 2 cycles
L2 Cache	512KB / 16-way / 7 cycles
L3 Cache	8MB / 32-way / 14 cycles

Table 4.2: Proposed e-PDEMI architectural parameters

Type	Benchmark
Integer	hmmer, astar, h264ref, sjeng libquantum, omnetpp, gcc, bzip2
Float	soplex, milc3, leslie3d, namd povray, bwaves, lbm, dealII

Table 4.3: Simulated SPEC2006 Benchmarks

TASS Periodic Sampler Parameter	Value
nInstSkip	3B
nInstSkipThreads	0
nSampleMax	480
nInstRabbit	2.53M
nInstWarmup	2.4M
nInstDetail	20K
nInstTiming	50K

Table 4.4: TASS Periodic Mode Sampler Parameter Values

4.3 Evaluation

4.3.1 Energy Per Instruction

Figure 4.3 illustrates the energy per instruction for each simulated benchmark running on both the baseline architecture and proposed e-PDEMI architecture as described in Sec-

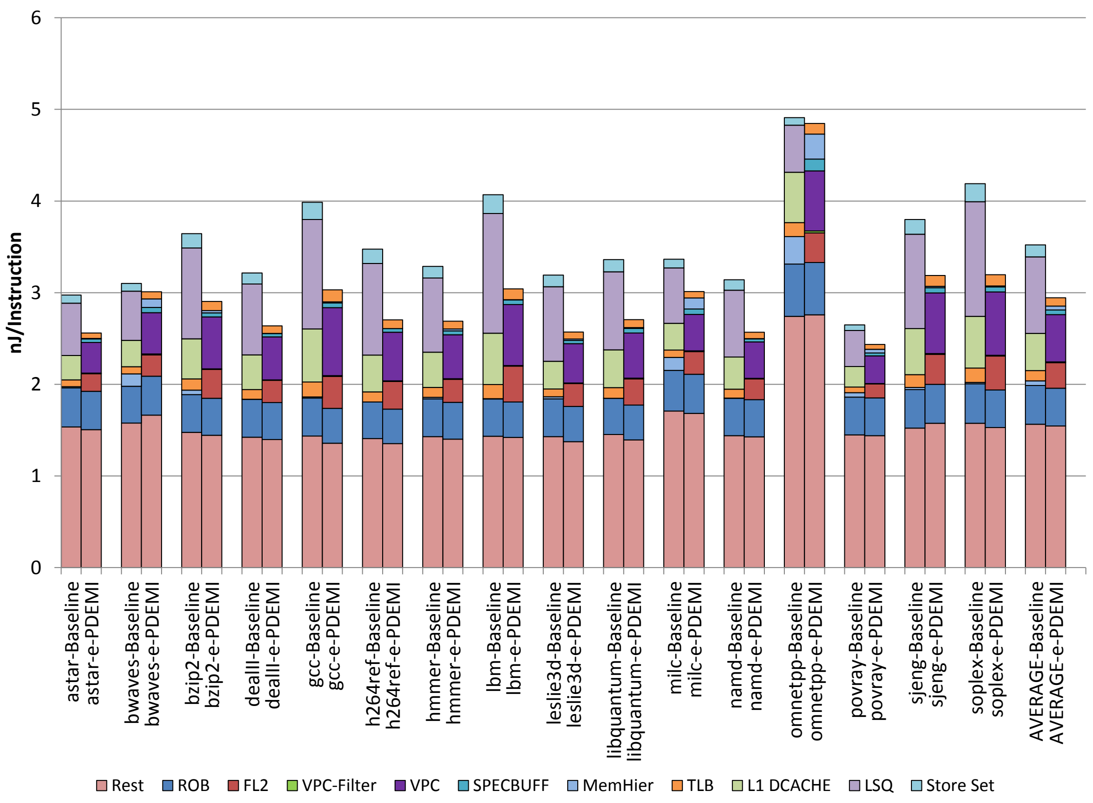


Figure 4.3: e-PDEMI has over 16% energy per Instruction savings

tion 4.1. The most important insight Figure 4.3 provides is that the e-PDEMI architecture reduces energy per instruction across all benchmarks. The average energy per instruction reduction is 16.4%. The "Rest" of the core dissipates about the same energy per instruction in both the Baseline and e-PDEMI systems due to the stall trading shown in Figure 4.8. This may seem counter intuitive because instructions need to be verified and e-PDEMI increases the amount of replayed instructions.

Verification does not impact the register file nor the re-order buffer energy consumption per instruction with respect to the baseline. Is also does not increase register file energy consumption per instruction because the instruction verification does not access the register file. e-PDEMI stores load and store instructions' out-of-order execution information in the memory speculation buffer (shown in Figure 4.3 as "SPECBUFF"). Due to its relatively small size, the memory speculation buffer does not significantly increase the energy per instruction on the e-

PDEMI architecture. e-PDEMI also does not increase the re-order buffer energy per instruction or other structures such as the rename logic because only a small percentage of instructions need to be replayed and the amortized cost is negligible.

The energy per instruction savings the e-PDEMI architecture achieves is maintained by carefully re-balancing other memory hierarchy structures to avoid increasing the energy dissipated per instruction. Since the memory hierarchy is off the critical path, the e-PDEMI architecture can tolerate longer latencies while maintaining performance similar to the baseline system. As a result, it is able to use a lower power mode of the L1 TLB to increase its efficiency at the cost of increasing its latency. The addition of a small L2 Filter cache avoids the potentially costly L2 verification access.

Neither the baseline nor e-PDEMI architectures experience any significant energy per instruction in the L2 or L3 caches (labeled "Mem_Hier").

The VPC is more energy efficient than the L1 cache even with the same size and organization. The VPC implements several optimizations only possible in a speculative cache. Specifically, the VPC includes a small buffer to simultaneously minimize speculative data pollution of the VPC, and to reduce the VPC's energy consumption per instruction. As shown in Figure 4.9, the VPC buffer is able to service a significant amount of memory accesses (over 20%), that would otherwise issue to the VPC and thus increase its activity rate. Additional small energy savings are due to the fact that the VPC does not allocate cache lines on writes and it never performs write backs to the next level of the memory hierarchy.

4.3.2 e-PDEMI Energy Comparison with SMDE

Figure 4.4 compares the average energy consumption by the memory hierarchy proposed for the e-PDEMI architecture and the memory hierarchy proposed in the SMDE architecture [27]. For fairness of comparison, each of the memory structures proposed in SMDE was sized to be the same size as their counterparts in e-PDEMI. CACTI 5.3 [52] was used to estimate the energy per access for each structure. Table 4.5 shows the structures from both SMDE and e-PDEMI and their selected sizes. To compute average energy dissipation over the set of benchmarks evaluated, the average activity rates for each memory structure were multiplied by the energy per access as estimated by CACTI.

In Figure 4.4, the SMDE TLB dissipates substantially more energy than the e-PDEMI

TLB. This is due to the need for the fully associative SMDE TLB to support virtual to physical address translations for every L0 Cache access. Because of this requirement, the SMDE TLB has a significantly increased energy consumption. Figure 4.4 also shows that because the VPC buffer is a direct mapped structure compared to the fully associative Fuzzy Disambiguation Queue used in SMDE, it dissipates less energy on average. Comparing the VPC with the L0 cache, the VPC dissipates slightly less energy than the L0 cache on average. This effect is due to the filtering mechanism of the VPC buffer. The VPC buffer prevents some accesses to the VPC which lowers the VPC's activity rate and thus it's average energy dissipation.

The most significant difference between e-PDEMI and SMDE in terms of energy dissipation comes from e-PDEMI's L2 Filter Cache. The SMDE L1 cache has the same activity rate as the L2 Filter Cache, but the L2 Filter Cache dissipates almost 50% less energy per access than the L1 cache in SMDE because it is much smaller. Finally, e-PDEMI's L2 cache dissipates slightly more energy on average than SMDE's L2 cache due to a slightly increased activity rate. This effect is attributable to the smaller L2 Filter cache, which has a higher miss rate than the much larger L1 cache in SMDE. However, the average energy dissipation savings found in the VPC and L2 Filter are enough to overcome this additional energy consumption, leading to an overall average energy consumption savings in the e-PDEMI memory hierarchy of almost 44% with respect to the SMDE memory hierarchy.

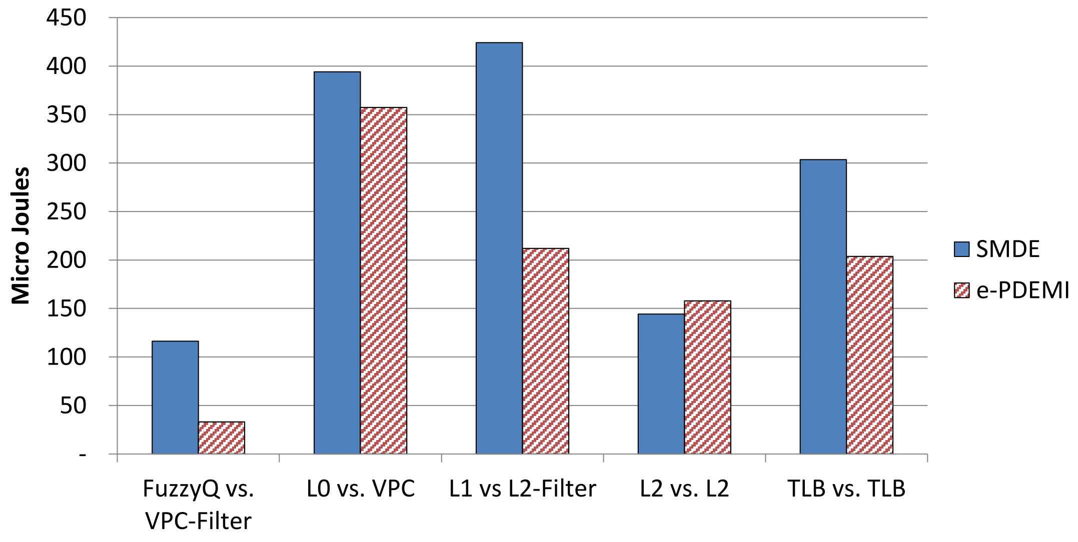


Figure 4.4: e-PDEMI consumes less overall energy than SMDE

e-PDEMI	**SMDE**
VPC Buffer 64 entry, direct mapped	Fuzzy Disambiguation Queue 16 entry fully associative
TLB 64 entry 4-way	TLB 64 entry fully associative
VPC 16KB 4-way	L0 Cache 16KB 4-way
L2 Filter Cache 4KB 16-way	L1 Cache 16KB 4-way
L2 Cache 512KB 16-way	L2 Cache 512KB 16-way

Table 4.5: e-PDEMI and SMDE Memory Hierarchies

4.3.3 Area Comparison

Using CACTI, the area required for the SMDE, e-PDEMI and Baseline memory hierarchies is estimated. Because only the memory hierarchies differ between e-PDEMI and the Baseline architecture, the same architectural parameters for SMDE are assumed, each architecture's memory hierarchy area cost is shown in Figure 4.5. The VPC and VPC buffer combine to make up 22% less area than the L1 and LSQ combined from the Baseline architecture. Since the VPC and L1 cache are the same size, this area savings is attributable to the large size of the fully associative LSQ with respect to the smaller VPC buffer. Having both an L0 and L1 cache that are the same size causes SMDE to be much bigger in area than either e-PDEMI or the Baseline architecture. e-PDEMI's major area savings over the SMDE architecture is the replacement of the full size L1 cache with the 62% smaller L2 Filter cache. Additionally, the fully associative Fuzzy Disambiguation Queue requires almost 53% more area than the VPC buffer. Additionally, e-PDEMI uses slightly less area than the Baseline architecture (about 2%).

4.3.4 Performance Analysis

Figure 4.6 shows the performance of the baseline and e-PDEMI architectures in terms of uIPC. uIPC is the retiring rate of micro-operations (result of instruction crack). It can be seen in Figure 4.6 that the e-PDEMI architecture performs at similar levels to the Baseline architecture and in some cases can provide a speedup of up to 6%. Because the e-PDEMI architecture does not rely on a Load Store Queue for memory disambiguation, it does not experience the

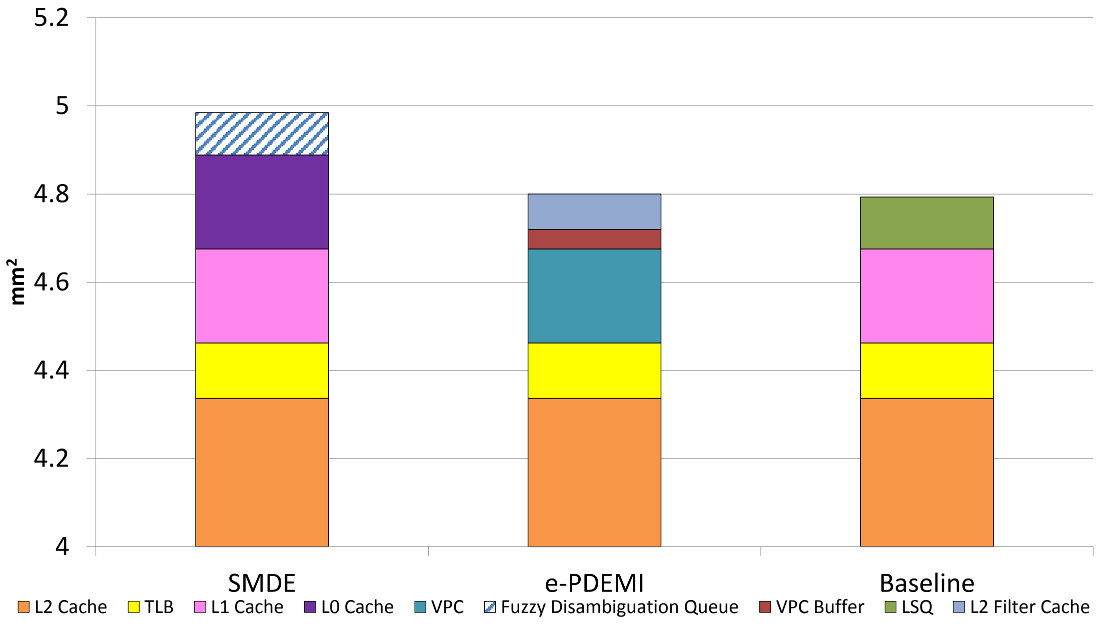


Figure 4.5: e-PDEMI memory hierarchy uses less area than SMDE memory hierarchy

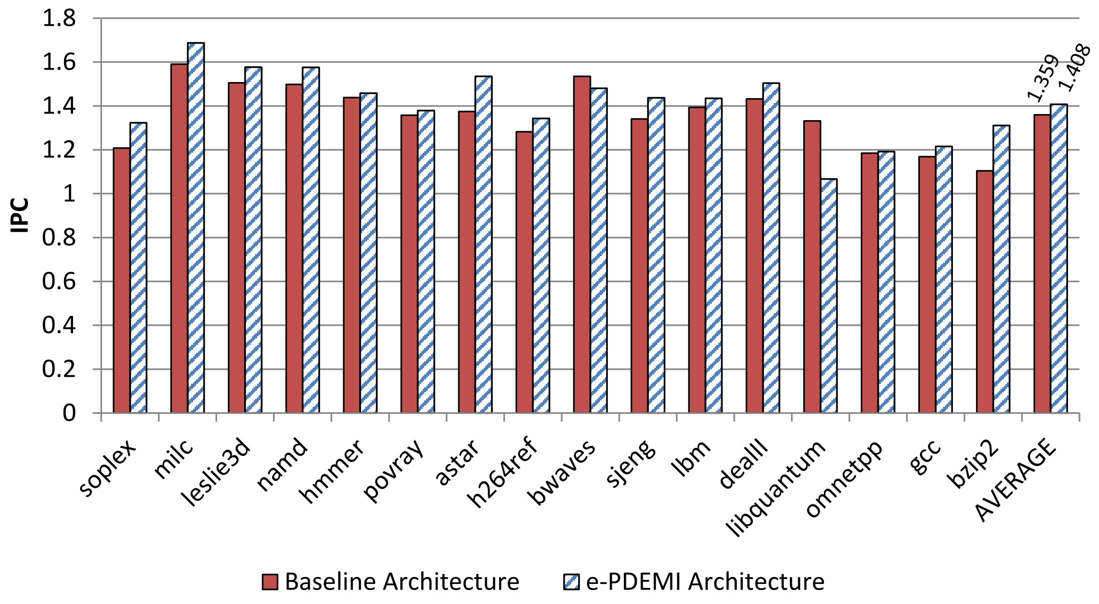


Figure 4.6: e-PDEMI has approximately the same performance as the baseline architecture comparing uIPCs

related stalls that come with those structures.

4.3.4.1 Instruction Replay Impact

While the e-PDEMI architecture can increase replays, the proportion of instructions that are actually replayed is small on average as shown in Figure 4.7. The $soplex$ benchmark is a good example of increased replays that do not necessarily slow performance. The e-PDEMI architecture achieves a speed up of 9% for $soplex$, although there is an increase of 4 times as many dynamic instructions executed due to replay with respect to the baseline architecture. However, the absolute total number of dynamic instructions executed due to replays is only 1.58% for $soplex$ running on the e-PDEMI system.

Another important observation can be seen in Figure 4.8 by looking at the pipeline stalls due to the Store and Load Queues. Since the e-PDEMI architecture does not have a Load or Store Queue, it cannot experience the pipeline stalls that the baseline architecture experiences due to those structures. This behavior is shown in the case of the $sjeng$ benchmark. It can be seen that 3.5% of the execution time of $sjeng$ on the baseline system is spent on stalls generated by the instruction window. In contrast, the e-PDEMI system only spends 0.8% of the execution time on stalls generated by the instruction window.

The additional instruction window stalls in the baseline system are due to the StoreSets' serialization of memory instructions. The baseline system experiences relatively high stalls while executing $sjeng$ due to replays triggered by the Store Queue. Each time the Store Queue triggers a replay the StoreSets structures are updated as described in Section 4.1.2. The ensuing dependency chaining behavior limits the Instruction Window's ability to freely issue memory instructions into the out-of-order pipeline creating back pressure from the StoreSets into the Instruction Window which thusly increases Instructions Window stalls as shown in Figure 4.8.

Although the baseline can have fewer instructions replayed than the e-PDEMI architecture, it achieves its low replay frequency by over-serializing memory instructions that may have triggered replays in the past, but were safe to execute out-of-order going forward. Because the e-PDEMI architecture uses a simple serialization technique, it only serializes instructions during periods of low forward progress (less than 200 instructions between replays) and thus will return to speculatively executing memory instructions out-of-order after a fixed period of time. The impact of lessening the instruction window pressure in the e-PDEMI architecture can provide more opportunity for the processor to extract increased ILP and thus mitigate the effects of increased replays.

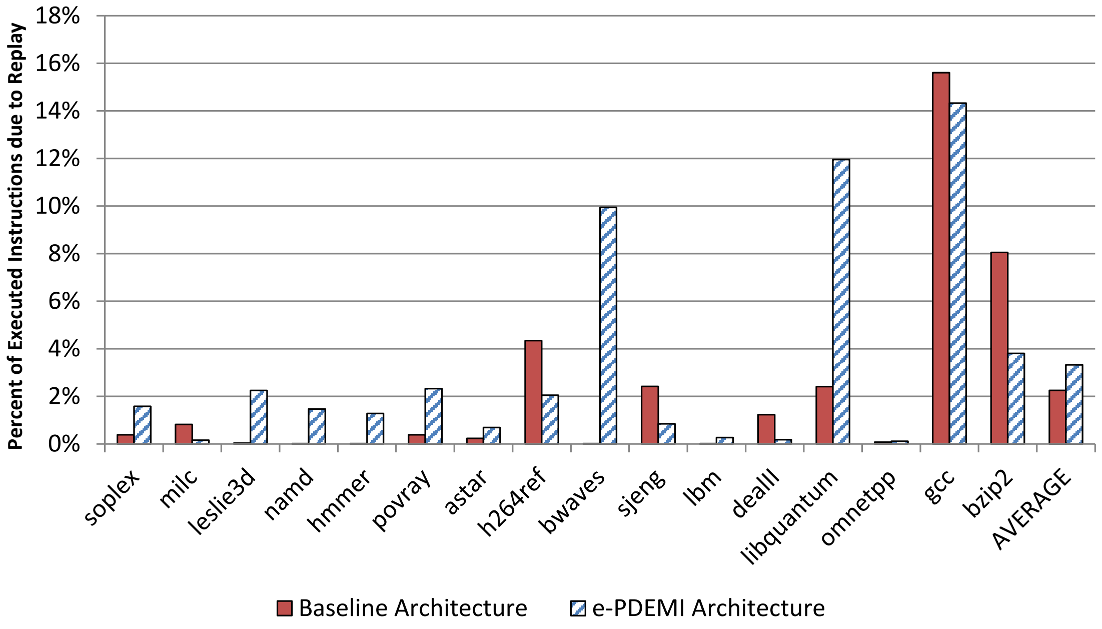


Figure 4.7: e-PDEMI has less than 3.5% increase in instructions executed as a result of memory replays

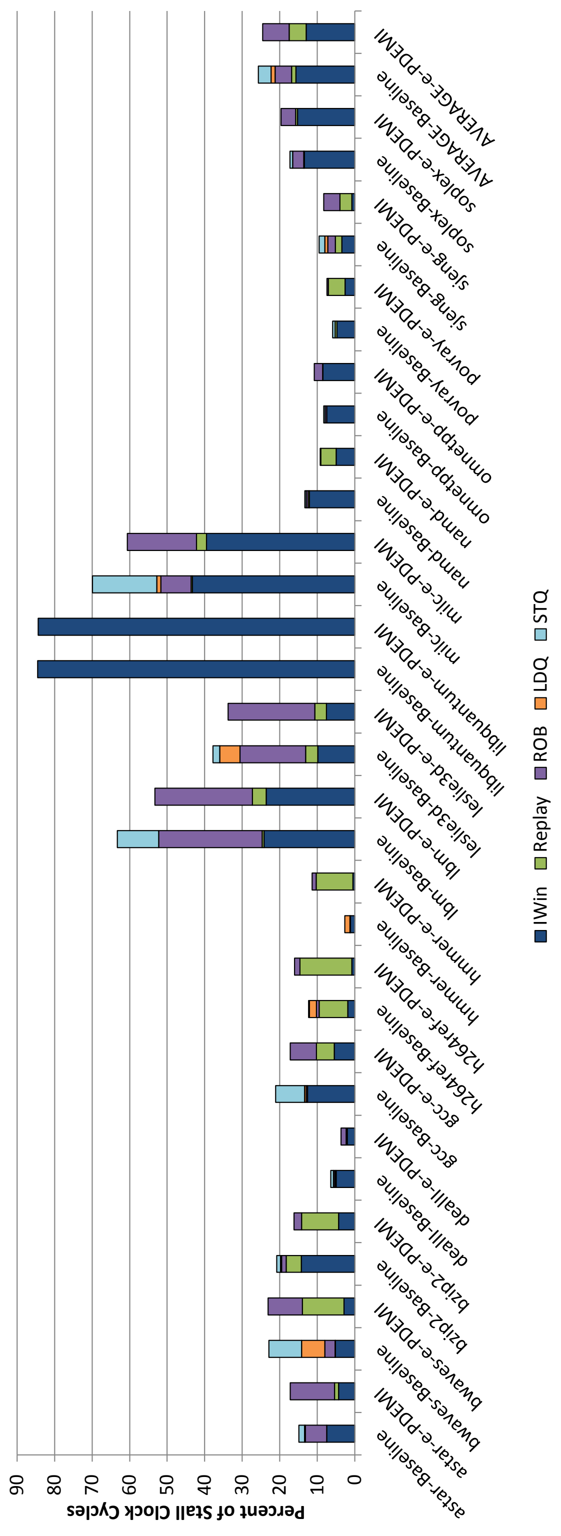


Figure 4.8: e-PDEMI trades replay stalls for store queue stalls

4.3.4.2 Stalls

Figure 4.8 shows the fraction of execution time spent on system stalls occurring from various functional units. Branch misprediction stalls are not shown in this figure for simplicity since they are out of the scope of this work and are not affected by the e-PDEMI architecture with respect to the Baseline architecture. The remainder of the execution time is spent in active operation. The e-PDEMI architecture reduces Instruction Window pressure in most cases. This is partially because both the Baseline and the e-PDEMI architectures stop fetching instructions during a replay, and since the e-PDEMI architecture uses commit time verification, it puts more pressure on the Re-order buffer rather than the Instruction Window in the average case. The e-PDEMI architecture clearly has more stalls due to replays, but on average it has slightly less stalls in total as seen in the plot, due to avoiding Load Store Queue stalls.

The e-PDEMI architecture reduces total stalls during the execution of $dealII$ with respect to the baseline architecture by almost 50%. By saving all store queue stalls and some instruction window stalls, the e-PDEMI architecture is able to tolerate increases in re-order buffer and replay stalls.

One method of potentially improving the baseline architecture's performance would be to increase the instruction window size. However, increasing the instruction window size is very costly because it would have to be significantly increased, and instruction windows are typically implemented using fully associative CAMs which would consume substantially more power. The e-PDEMI architecture in the average case nearly doubles re-order buffer stalls, but because it does not experience Load Store Queue related stalls, overall stalls are reduced. While in most cases where the e-PDEMI architecture saves Load Store Queue stalls, it gains them back in replays. However as shown in Figures 4.3 and 4.6, the e-PDEMI architecture is able to preserve the performance of the Baseline architecture and save energy.

4.3.5 VPC Performance

Approximately 17.8% of the memory reads result in correctly predicted data from the VPC buffer. This accounts for significant energy savings by simultaneously preventing read accesses to the VPC which consume much more energy than VPC buffer accesses while also correctly predicting the loads' required data thus also avoiding energy costly replays. $Soplex$ experiences the highest rate of VPC Buffer correct predictions which translate to less than av-

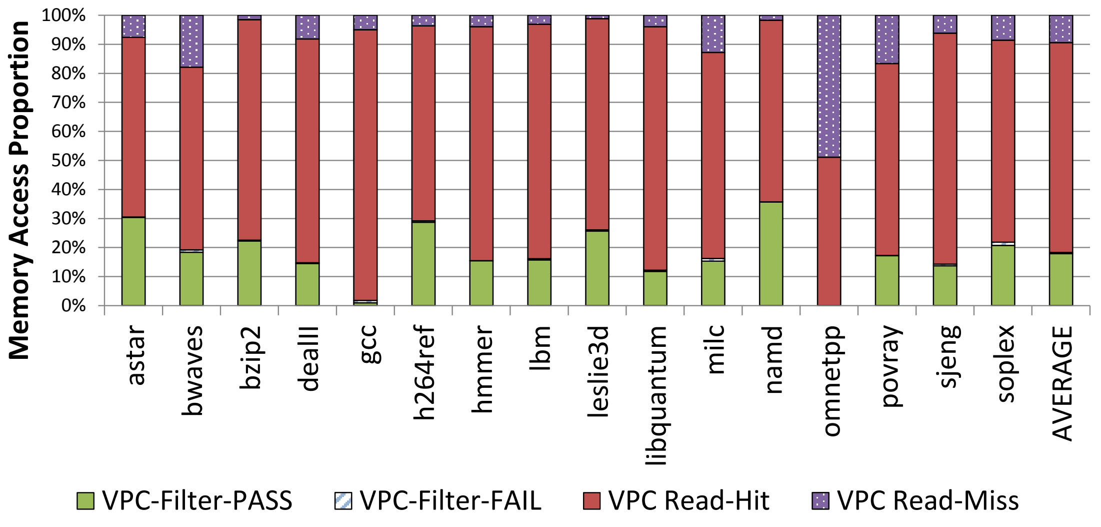


Figure 4.9: The majority of speculative accesses hit in the VPC

erage VPC energy consumption. The prediction accuracy of the VPC Buffer is very important for performance and energy consumption. If the VPC Buffer frequently predicts incorrectly, then the resulting replays will cause the e-PDEMI architecture's serialization mechanism to serialize most memory instructions and performance will be dramatically degraded as seen in Figure 4.13 and discussed in Section 4.3.7. On average, 0.4% of the accesses to the VPC Buffer receive incorrect data which then result in replays. 81% of reads miss in the VPC Buffer and thus access the VPC. 72% of reads result in a VPC hit and 9.4% of reads trigger a miss request to the L2 filter cache which will return a line of non-speculatively updated data (from the back-end in-order execution).

4.3.6 Sizing Characteristics

Figure 4.10 shows the effect of increasing the size of the L2 filter cache. Figure 4.10 presents instructions per cycle (IPC), energy per instruction and energy-delay product data normalized to the standard e-PDEMI architecture configuration of a 4KB filter cache. Although performance does not appreciably change as the L2 Filter Cache size is increased, energy per instruction of the overall e-PDEMI architecture does increase. The configuration of the e-PDEMI architecture evaluated throughout the rest of this work therefore uses an L2 filter cache size of 4KB to reduce the L2 filter cache's impact on energy without reducing performance.

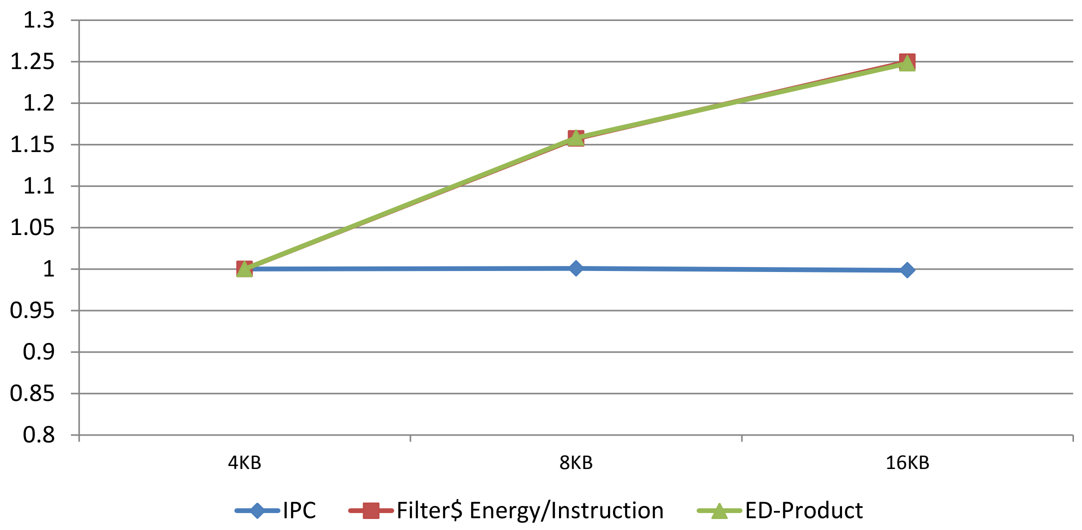


Figure 4.10: Small 4KB L2 Filter Cache Has The Best Energy-Performance Results

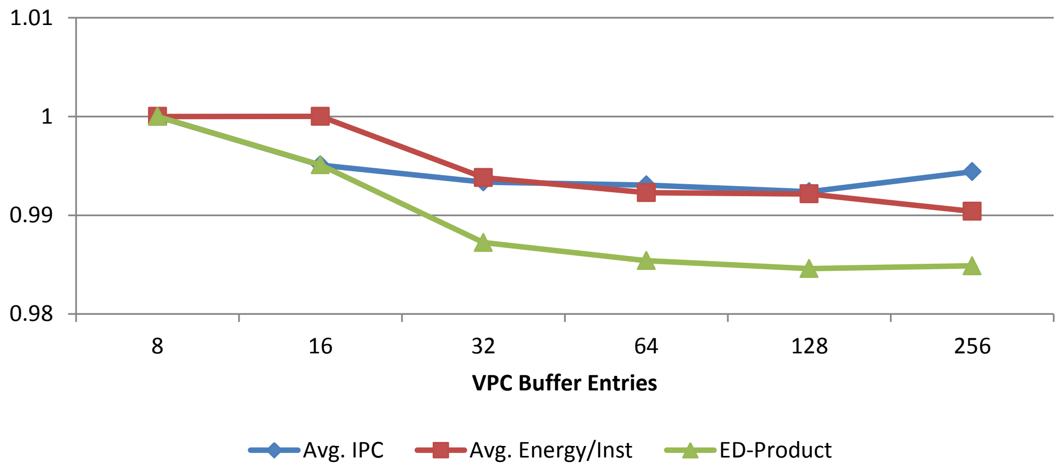


Figure 4.11: A small 32 or 64 VPC-Filter is Enough

The VPC buffer has been shown to be a critical component of the e-PDEMI architecture. It simultaneously filters accesses to the VPC for energy savings, and prevents speculative data pollution in the VPC which can result in severe replay increases degrading performance. Figure 4.11 shows energy per instruction and IPC as functions of the number of entries in the VPC buffer. IPC and energy per instruction almost do not change over the range of 8 to 256 entries, although there is a very small reduction of overall energy per instruction at 64 entries with little impact to IPC and thus 64 entries was selected as the standard VPC Buffer size.

4.3.7 Additional Characterization

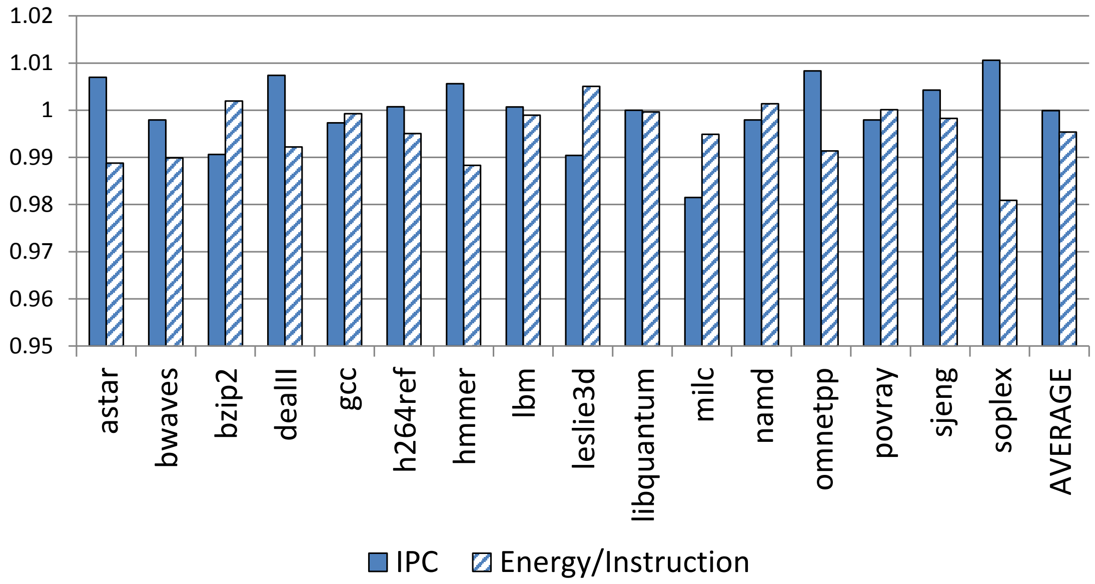


Figure 4.12: Enabling write allocate policy in VPC has negligible performance and energy impact

Figure 4.12 shows the IPC and energy per instruction results of allowing write allocation in the VPC normalized to the results of not allowing write allocation. The standard VPC does not allow allocation on write misses. When store instructions are issued to the VPC at retirement, if the cache line needed by a store is not present in the VPC, the write does not proceed. This avoids extra accesses to the L2 Filter Cache in order to service write misses from the VPC while the same store simultaneously executes to the L2 Filter Cache at retirement.

Figure 4.12 shows that by allowing write allocation in the VPC, IPC is on average 99.9% of the IPC when VPC write allocation is disallowed. Applications such as $astar$, $dealII$, $hmmer$, $omnet$ and $soplex$ show some IPC improvement along with some energy per instruction reduction, while other applications such as $milc$, $bzip2$, and $leslie3d$ do not because write allocation in the VPC is speculative and thus can increase the VPC miss rate as well as pollute the VPC with speculative data that can cause additional misprediction replays. In Figure 4.12, energy per instruction is on average 99.5% of the nonwrite-miss allowed VPC policy, so the VPC is not allowed write allocation resulting in implementation simplicity with no loss of energy efficiency or performance.

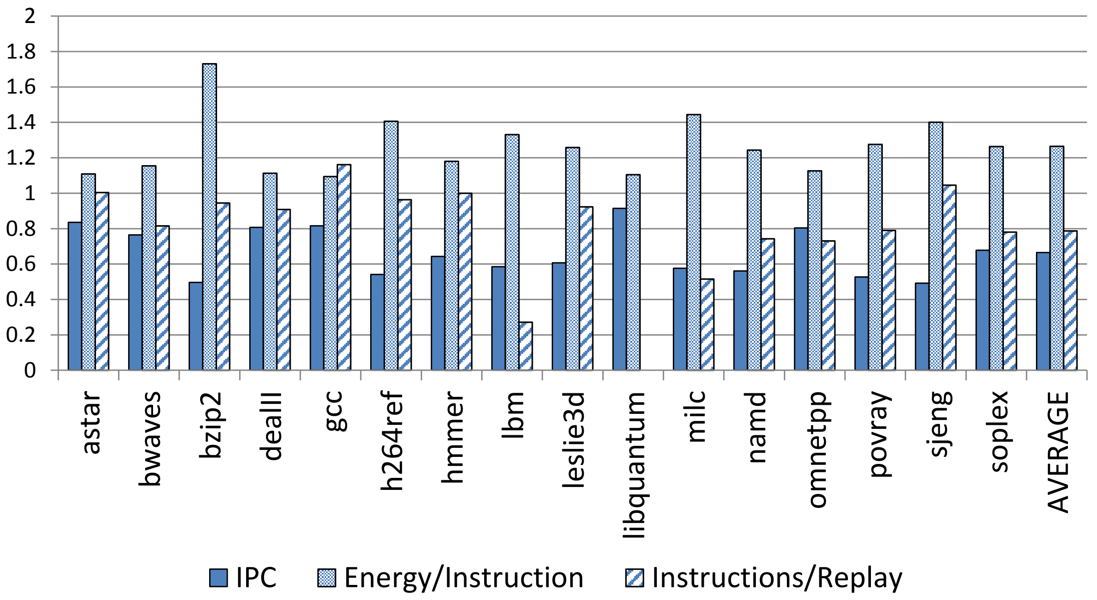


Figure 4.13: A VPC with VPC buffer displacements enabled degrades performance and increases EPI

Figure 4.13 illustrates that allowing the VPC buffer to displace into the VPC reduces IPC by 33.4% on average. This is because excessive out-of-order speculative data is polluting the VPC as opposed to being overwritten as program execution progresses. This effect can be seen by the increase in replays by 21.2% shown in Figure 4.13. Finally, this increase in replays results in 24.6% increased energy per instruction for the e-PDEMI architecture. As a result, e-PDEMI does not allow the VPC Buffer to displace into the VPC.

Chapter 5

The Multicore e-PDEMI Architecture

5.1 ePDEMI Multi-Core Architecture

The ePDEMI multi-core Architecture builds upon the energy efficient memory speculation proposed and evaluated in Chapter 4 by supporting the Sequential Consistency memory model and removing a traditional invalidation-based memory coherence protocol.

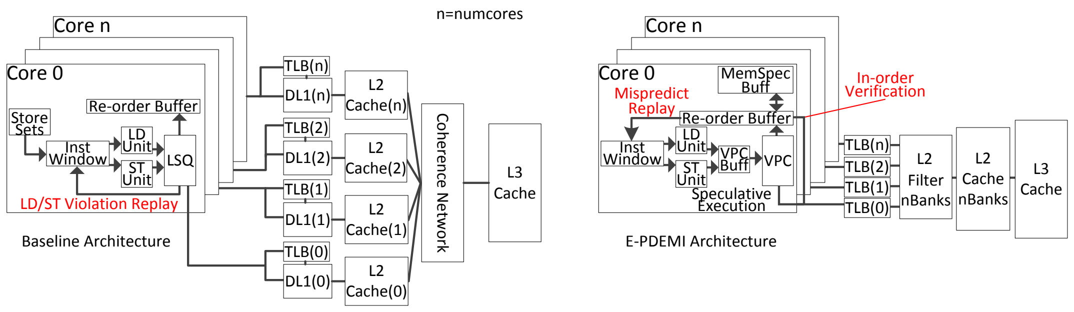


Figure 5.1: Baseline and proposed multi-core architectures

5.1.1 Baseline Multicore Memory Hierarchy

The multicore baseline implementation is composed of multiple instances of the baseline core discussed in Chapter 4 each with a private L1 and L2 data cache. Shown in Figure 5.1, each baseline core has a private TLB that is accessed in parallel with the L1 data cache. This is increases performance supported by configuring the private L1 data caches to be virtually indexed and physically checked. Each private L2 cache is connected to an interconnection network. Each private cache maintains coherence by implementing the Modified, Owned, Exclusive, Shared, or Inavlid (MOESI) [6] memory coherence protocol.

5.1.2 Cache Inclusion

In order for lower level caches to be aware of data present in higher level caches, the baseline system uses a policy of cache inclusion where a line residing in an L1 data cache must also reside in the L2 and L3 caches below it. A cache line placed in the L2 or L3 data cache by one processor, may be evicted due to the request of a conflicting cache line by another processor. Figure 5.2 illustrates this potential scenario. In step 1, Processor 1 issues a read of address A to its private L1 data cache, since this is a cold request, this triggers ensuing read requests to Processor 1's private L2 cache and the shared L3 cache in steps 2 and 3. Steps 4, 5, and 6 show the requested cache line being stored in the shared L3 cache, and Processor 1's private L2 and L1 caches.

Starting in the lower right corner of Figure 5.2 Processor 2 issues a read request for Address B in step 7. Again, this is a cold request for Address B and steps 8 and 9 show the ensuing read requests to higher cache levels. At step 10, the shared L3 cache determines that there is a line conflict between Address A and Address B. Address A must be evicted. Step 11 shows that the shared L3 cache evicts Address A and replaces it with Address B. However, because Address A no longer resides in the shared L3 cache, the cache inclusion policy prevents it from residing in any higher levels of the cache hierarchy. Thus, step 12a shows the shared L3 cache sending an invalidate message for the cache line corresponding to Address A (which also corresponds to the newly cached Address B) to Processor 1's private L2 cache. Cache inclusion again requires that if a line is invalidated in the private L2 cache, it must also be invalidated in the private L1 cache above it. Step 13a shows the final invalidation message sent from Processor 1's private L2 cache to its private L1 data cache. Simultaneously, steps 12b and 13b show Addr B being cached in Processor 2's private L2 and L1 caches.

5.1.3 MOESI Memory Coherence

As seen in Figure 5.1, each core of the baseline architecture has private L1 and L2 data caches. In order for data to be kept coherent between private data caches, a coherence protocol is required. Without such a protocol, data stored in private caches can become stale.

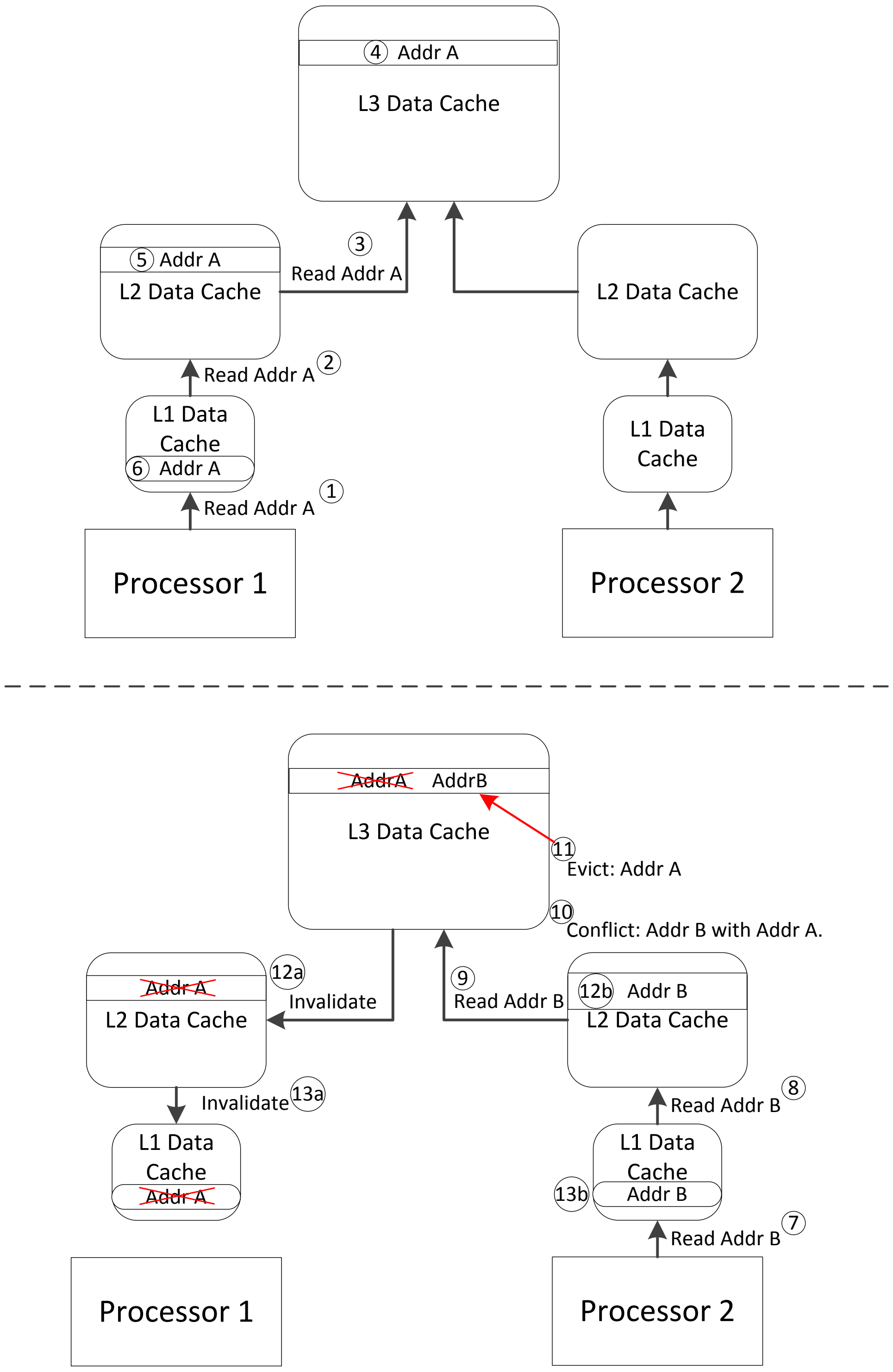


Figure 5.2: Baseline architecture cache hierarchy inclusion example

Figure 5.3 shows a case where private caches could become incoherence without a coherence mechanism in place.

Step 1 of Figure 5.3 shows Processor 1 issuing a cold read operation for Address A. Steps 2, 3, and 4 show the ensuing read requests issued to each next lower cache level until the shared L3 cache requests Address A's data from main memory. Address A's value is found to be *5*, and a *5* value is written back to each of the requesting caches in steps 5 and 6. Sometime later, Processor 2 issues a write request at Address A with value *7* to its L1 data cache in step 7. Later in steps 8 and 9 the value *7* propagates to Processor 2's private L2 cache and the shared L3 cache. At this point, Processor 2's complete memory hierarchy contains a different value at Address A than Processor 1's private L1 and L2 caches. Inclusion does not prevent this problem from occuring because all operations in this example were performed to the same memory address. Later in time if Processor 1 were to issue a read request for Address A to its L1 data cache, its L1 data cache would hit and return the value *5*. Processor 1's L1 and L2 data caches have no way of knowing that data they are caching has been modified in a higher level cache, or other private caches on the system. The MOESI coherence protocol and memory interconnect included in the baseline architecture are the mechanisms that provide caches on the system with this information. Figure 5.4 shows the same scenario again, this time corrected using the MOESI protocol.

As in Figure 5.3, steps 1 through 3 occur in the same way in Figure 5.4. Steps 4, 5, and 6 occur similarly to their counter parts in Figure 5.3 with the addition that each cached line is appended with state information. Because steps 1 through 3 are cold read requests, the resulting cache lines in the shared L3 cache, and Processor 1's L1 and L2 caches are marked as being in the "Exclusive" state of the MOESI protocol. Step 7 initiates the write operation from Processor 2 to Address A. Due to inclusion, that write operation must be also carried out in Processor 2's L2 cache (shown in step 8) and the shared L3 cache (shown in step 9). The difference in Figure 5.4's steps 7, 8, and 9 is that each of the resulting cache lines is marked in the "Modified" state from the MOESI protocol. Further, because of cache inclusion, the shared L3 cache would be aware that Processor 1's private L2 cache is storing now stale data at Address A and per the implemented coherence protocol it must issue an invalidation of those cached lines. Step 10 shows the invalidation of Processor 1's private L2 cache, and similarly due to inclusion, Processor 1's private L2 cache would be aware that Processor 1's private L1 data

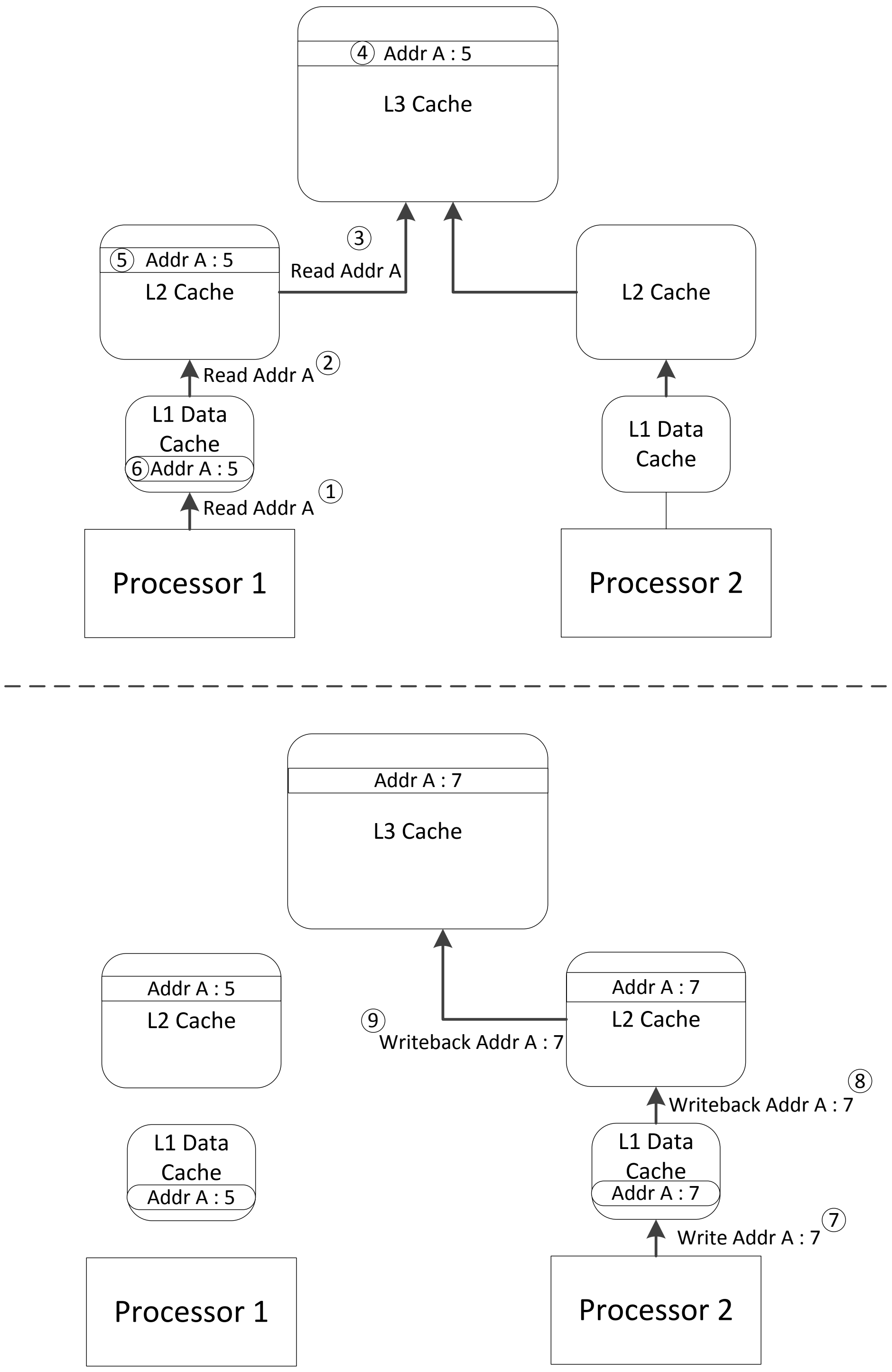


Figure 5.3: Baseline architecture memory incoherence example

cache is also caching stale data at Address A and would send an invalidation message for that cache line (shown in step 11). By the end of step 11, it is clear that if Processor 1 were to initiate a read of Address A, its L1 and L2 caches would miss and receive the updated Modified data from the shared L3 cache thus preventing the incoherence problem discussed above and shown in Figure 5.3. Once such a read request took place, all of the cache line copies corresponding to Address A on the system would be marked "Shared", and Processor 2's copy would change state to Owned.

As discussed previously, the multicore baseline system in this work uses the MOESI [6] protocol for memory coherence. The states for an MOESI protocol are:

- Invalid: A cache line in the invalid state does not hold a valid copy of the data. Valid copies of the data can be either in main memory or another processor cache.
- Exclusive: A cache line in the exclusive state holds the most recent, correct copy of the data. The copy in main memory is also the most recent, correct copy of the data. No other processor holds a copy of the data.
- Shared: A cache line in the shared state holds the most recent, correct copy of the data. Other processors in the system may hold copies of the data in the shared state, as well. If no other processor holds it in the owned state, then the copy in main memory is also the most recent.
- Modified: A cache line in the modified state holds the most recent, correct copy of the data. The copy in main memory is stale (incorrect), and no other processor holds a copy.
- Owned: A cache line in the owned state holds the most recent, correct copy of the data. The owned state is similar to the shared state in that other processors can hold a copy of the most recent, correct data. Unlike the shared state, however, the copy in main memory can be stale (incorrect). Only one processor can hold the data in the owned state all other processors must hold the data in the shared state.

Figure 5.5 shows the valid state transitions for the baseline architecture's coherence protocol. A key differentiator of the MOESI protocol compared to other coherence protocols such as MEI, MSI, or MESI is the presence of the "Owned" state representing data that is both owned and shared. This delays the need to write back modified data to main memory. Eventually such

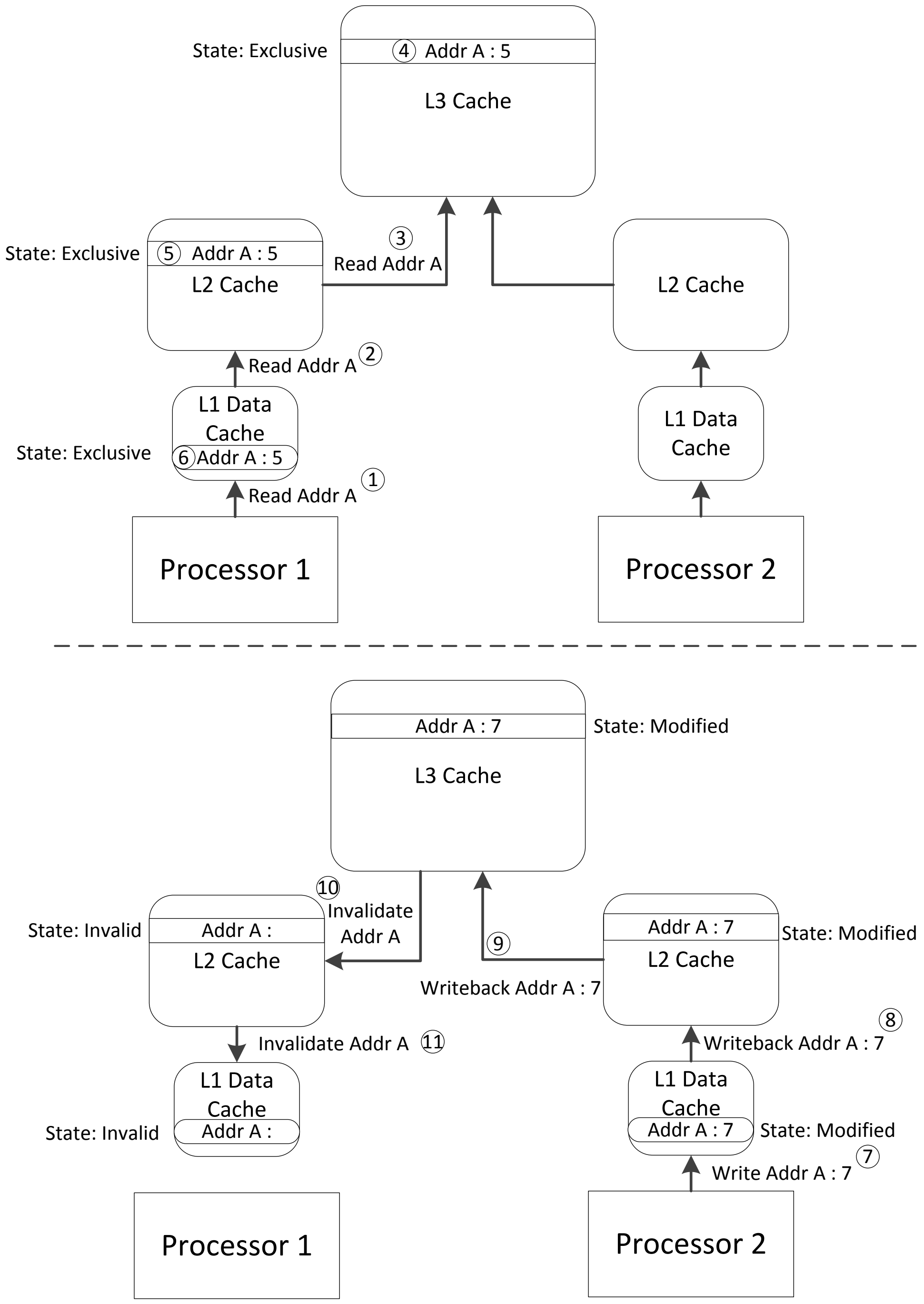


Figure 5.4: Baseline architecture memory with MOESI coherence protocol example

modified data will be written back to main memory on an eviction, but can reside in a cache coherent memory hierachy indefinitely until that event.

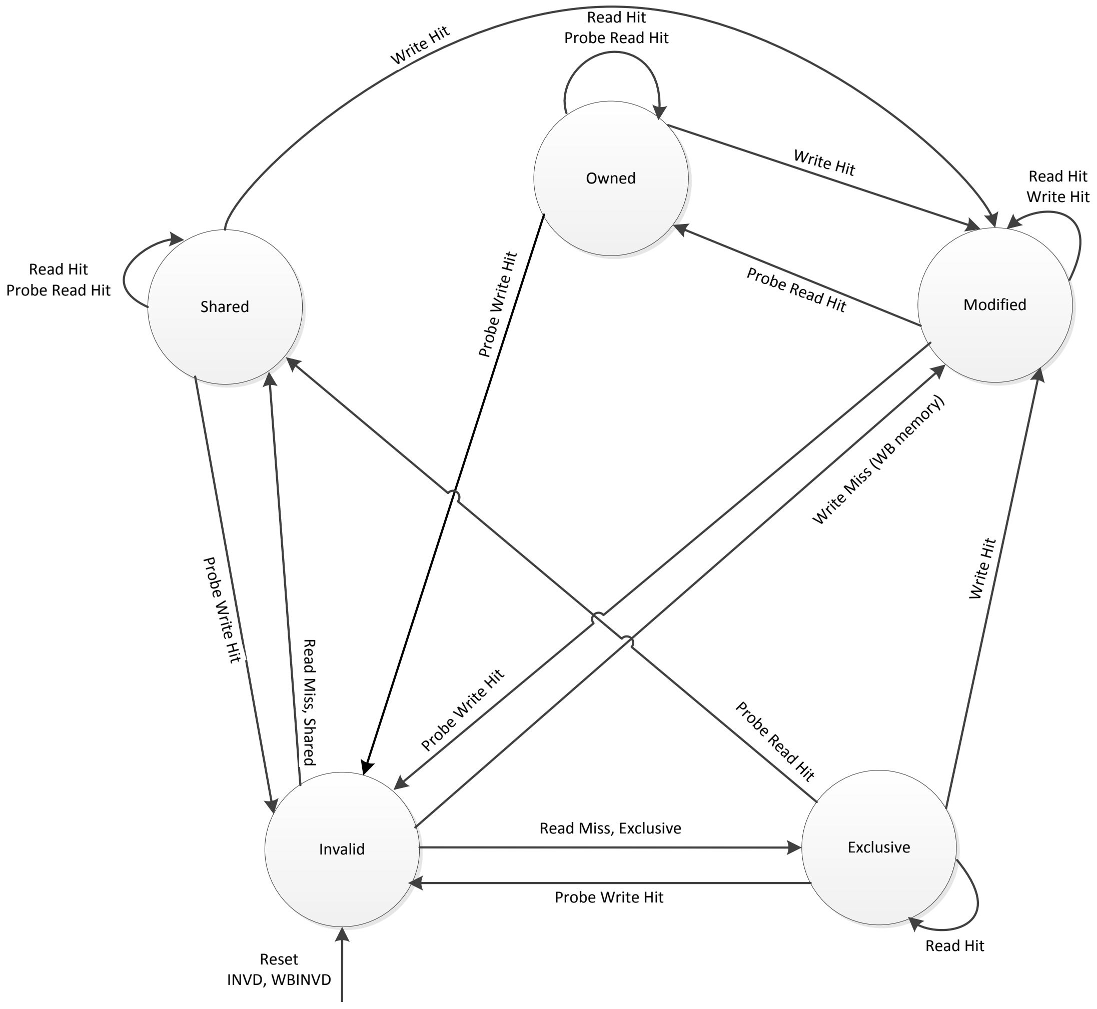


Figure 5.5: The MOESI protocol state transitions

The MOESI coherence protocol that the baseline system uses requires messages facilitating coherent cache line state changes to be sent between cache structures. This message passing is supported by a memory interconnection network. Many such networks have been proposed such as Intel's Quick Path Interconnect [43] and AMD's HyperTransport [4]. Although the energy consumption of such an interconnect is not considered in this work, the baseline architecture's total energy consumption will be increased due to its reliance on a coherence interconnection network. Further, as core densities increase, these networks are challenged to scale their bandwidths accordingly. Depending on the protocols implemented on the network, the available bandwidth can become insufficient. Additionally, the energy consumption of increasingly scaled networks can become significant. Many works investigating the tradeoffs of network topologies are motivated by these scaling challenges [14, 23, 53, 54, 63]. There has also been significant work investigating the acceleration or simplification of interconnect verification due to its complexity [22, 31, 68, 69]. In Section 5.3.1 it can be seen that including a coherence protocol and the required interconnect in the baseline architecture is crucial to support the baseline's high performance. However, the e-PDEMI multi-core architecture and memory hierarchy are able to tolerate its removal while sustaining the same high performance as the baseline architecture while avoiding the additional complexity and energy consumption that including a coherence mechanism would require.

5.1.4 e-PDEMI Multicore Memory Hierarchy

As discussed in Chapter 4, the ePDEMI architecture's speculative design allows it to tolerate extended latency in its memory hierarchy. The multi-core e-PDEMI architecture relies on that memory latency tolerance to support reduced coherence complexity. As also introduced in Chapter 4, the e-PDEMI architecture verifies speculatively executed memory operations at the out-of-order processor's retirement stage which is performed in-order. Because of this in-order issuance of memory operation to the multi-core e-PDEMI memory hierarchy, and details discussed below in Section 5.1.5, the multi-core e-PDEMI architecture also provides the Sequential Consistency Memory Model.

5.1.4.1 Off The Critical Path

Because access to the main memory hierarchy is off the critical path, the e-PDEMI architecture is able to tolerate longer memory latencies. As a result, from Figure 5.1 it can be seen that the e-PDEMI memory hierarchy is composed of a set of private TLBs accessed by each e-PDEMI core, which will access a single shared banked L2 Filter Cache, which can access a larger shared banked L2 cache before the last level shared L3 cache. Since there are no private data caches in the multi-core e-PDEMI system, any given cache line will only reside in each structure in exactly one location. As a result, data on the e-PDEMI system cannot become incoherent even in a multiple-reader/multiple-writer scenario. A coherence protocol, coherence support structures, and coherence interconnect are therefore not included in the multi-core e-PDEMI architecture. Finally, because of the multi-core e-PDEMI shared cache hierarchy and the in-order commit of stores from e-PDEMI cores discussed above, stores become globally visible in program order. Therefore, the multi-core e-PDEMI architecture meets the first requirement for Sequential Consistency set forth by Lamport [45], that memory operations from a given processor be issued to the memory hierarchy in program order.

Figure 5.6 shows the multi-core e-PDEMI architecture memory hierarchy providing memory coherence in the scenario shown in Figure 5.3 where the baseline architecture memory hiearchy became incoherent without the use of a coherence protocol and interconnect mechanism. In steps 1 through 3, Processor 1 makes a cold read request for address A to the shared banked L2 Filter Cache which results in cold read requests to the shared banked L2 Cache and shared L3 Cache. In steps 4, 5, and 6 the cache line containing the data (equal to *5* at that time) at address A is returned to each of the requesting caches. Sometime later in step 7, Processor 2 initiates a write operation to address A with value *7* to the shared banked L2 Filter Cache. Because the L2 Filter Cache banks are address indexed, there is only one bank that address A may reside and thus the old value *5* is overwritten with the new value *7*. Steps 8 through 12 carry out the update later in time of the higher level caches. The most important aspect of this operation to note, is that after step 8, if Processor 1 were to initiate a new read request for address A, it would read the correctly updated value *7*. Thus, the e-PDEMI memory hierarchy is always coherent without needing a coherence protocol or interconnect.

5.1.5 Sequential Consistency on e-PDEMI

As described above, the e-PDEMI architecture supports sequential consistency. At verification, the ROB issues instructions in program order to the memory hierarchy for verification. This meets Lamport's first requirement that all memory operations on a given processor be issued to the memory hierarchy in program order. Every cycle a memory operation can go to one of the address mapped L2 Filter cache banks, each of which is non-blocking and has a Miss Information Status Holding Register(MSHR) [42]. If a memory operation misses, it will pin down the corresponding cache line in the MSHR. Subsequent memory operations to the same address will also get pinned down in the same bank's MSHR, until the original store miss request is serviced. Once the original memory operation miss is resolved, all outstanding requests

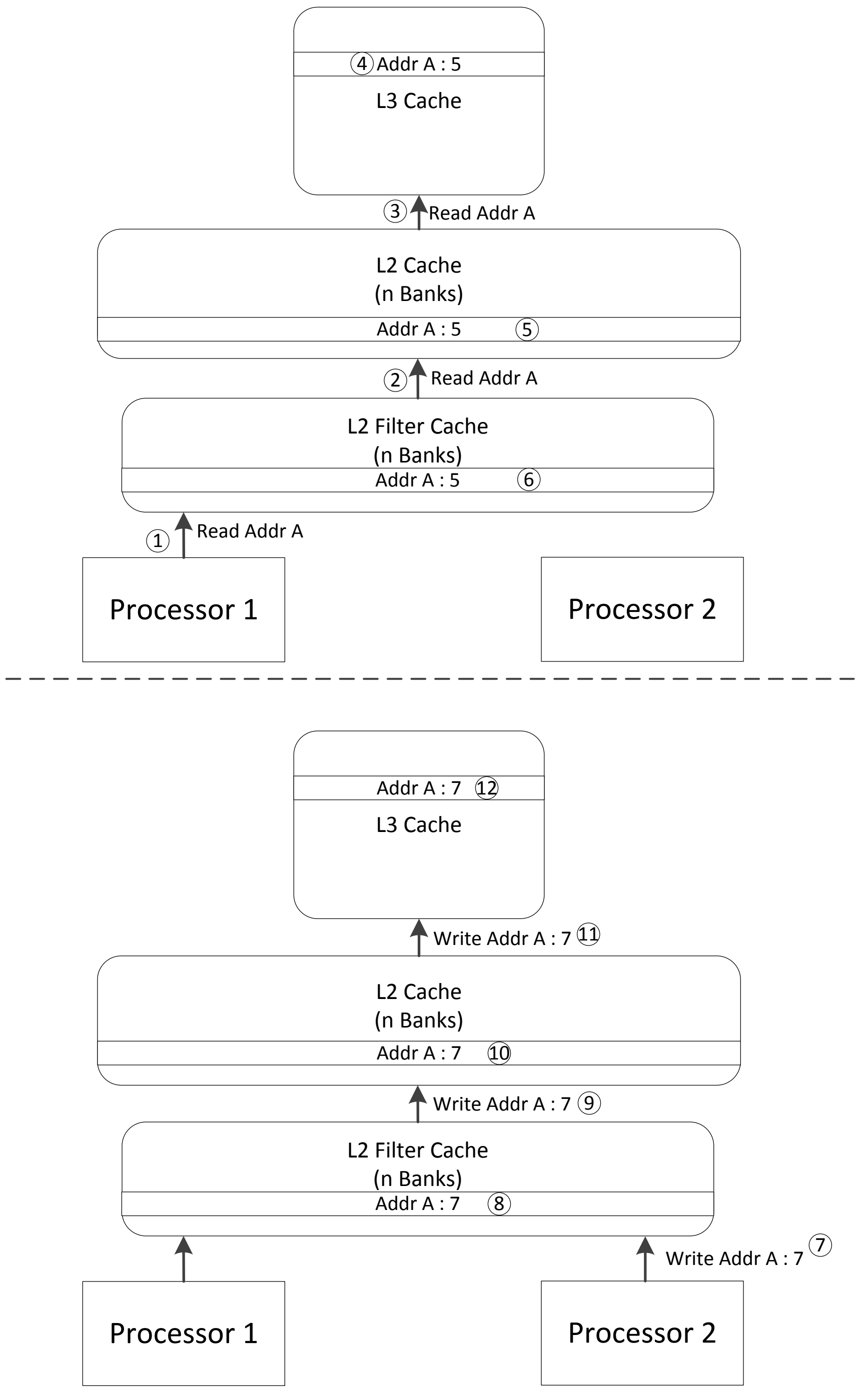


Figure 5.6: The e-PDEMI memory hierarchy provides memory coherence

for that line are fullfilled in the order in which they were received by the L2 Filter cache bank. This MSHR mechanism meets Lamport's [45] second requirement for Sequential Consisitency, that memory elements only process requests to the same datum in the order in which they were received.

5.1.6 Two Phase Commit Protocol

In order to support increased memory instruction retirement throughput, ePDEMI introduces a two phase commit protocol similar to the two phase commit protocol used in databases. Each cycle the ROB can send a memory verification request. If the request misses in the corresponding cache bank, it is held in the bank's MSHR, like any other request, until the earliest outstanding miss to the required cache line is serviced. Once a memory request is either a hit, or has its miss resolved, the MSHR sends a commit signal to the ROB which will then retire outstanding memory instructions in program order. Since the MSHR services all outstanding memory requests to the same line in the order in which they were received, and because memory verification requests are sent by the ROB in program order, the MSHR implicitly maintains the global visibility of stores in program order thus meeting the definition of sequential consistency.

Figure 5.7 shows an example of the two phase commit protocol that occurs between the Reorder Buffer (ROB) of each processor and the shared cache hierarchy in the multi-core e-PDEMI architecture. Note, that for simplicity the shared L2 cache and shared L3 cache are not shown although Figure 5.7 assumes that L2 Filter Cache misses are serviced by those lower cache levels. It can be seen in Figure 5.7 that each processor's ROB has a commit pointer and a head pointer. The head pointer points to the operation in the ROB to be retired and the commit pointer points to the last operation in the ROB that has been retired. It is important to note that the ROB can, and in practice will, contain any instruction type and is only restricted to memory operations in Figure 5.7 for simplicity of explanation. In time step 1, both processors' ROBs initiate the first phase of e-PDEMI's two phase protocol by issuing the operations pointed to by each ROB's head pointer to the shared cache hierarchy. For simplicity of explanation, Figure 5.7 assumes each request in time step 1 maps to a different bank of the L2 Filter Cache structure and thus the requests are processed in parallel. If both requests mapped to the same L2 Filter Cache bank, then the Bank arbitration logic will serialize these requests deterministically. This arbitration behavior is included in the simulations presented in this work. At the end of time step 1, each processor's ROB moves its head pointer up one entry. The ROB's primary function is to retire instructions in-order and it is assumed that the head pointer moves through buffered instruction in program order.

In time step 2, both processor's ROBs again issue their respective memory requests pointed to by their head pointers. In time step 3, both processor's ROBs again issue the memory operations pointed to by their head pointers. Additionally during time step 3, The L2 Filter Cache banks corresponding to the requests issued during time step 1 send back the second phase of the e-PDEMI two phase commit protocol by confirming the two requests as correct. Recall that the purpose of the in-order retire-time verification of memory instructions is to compare the speculatively consumed values of each memory instruction to the guaranteeed correct data stored in the e-PDEMI memory hierarchy, thus the second phase of the two phase commit protocol is crucial. In this case the requests were verified as correct. If either of them had been found incorrectly speculated, a replay would have been triggerd in the issuing ROB.

Since both processor's ROB's received a confirmation signal on their first requests, they advance their commit pointers by one entry, indicating that each of their respective first entries has now been retired. In time step 4, Processor 1's ROB receives a replay signal from the L2 Filter Cache for the Load from address D that it issued during time step 2. This signal indicates that the Load from address D consumed incorrect speculative data during out-of-order execution. Since the L2 Filter Cache provides correct data to the Load from address D, that instruction is allowed to retire and processor 1's ROB initiates a replay from the next instruction proceeding that Load. Thus processor 1's ROB is flash cleared of the instructions between that load and the current position of its head pointer by moving the commit pointer to the entry after the Load from address D, and moving the head pointer to be at the same entry as the commit pointer.

Processor 2's ROB receives a bulk phase two confirmation from the L2 Filter cache during time step 4. Bulk confirmations are allowed in the e-PDEMI two phase commit protocol. They are handled sequentially by the ROB. This means that the ROB will only move its commit pointer past the *earliest* confirmed memory operation. Because in time step 4 both the Load from address X and the Load from address L are confirmed to processor 2's ROB, its commit pointer can be moved to the instruction after the Load from address L. If the L2 Filter Cache had instead only confirmed the Load from address L, the commit pointer would not be permitted to advance, because the earlier Load from address X had not been confirmed. This mechanism enforces the in-order retirement of instructions from the ROB. Finally, although not shown in the example depicted in Figure 5.7, Store instructions are assumed by the processors' ROBs to be silently confirmed since they do not require verification as in the case of Load instructions.

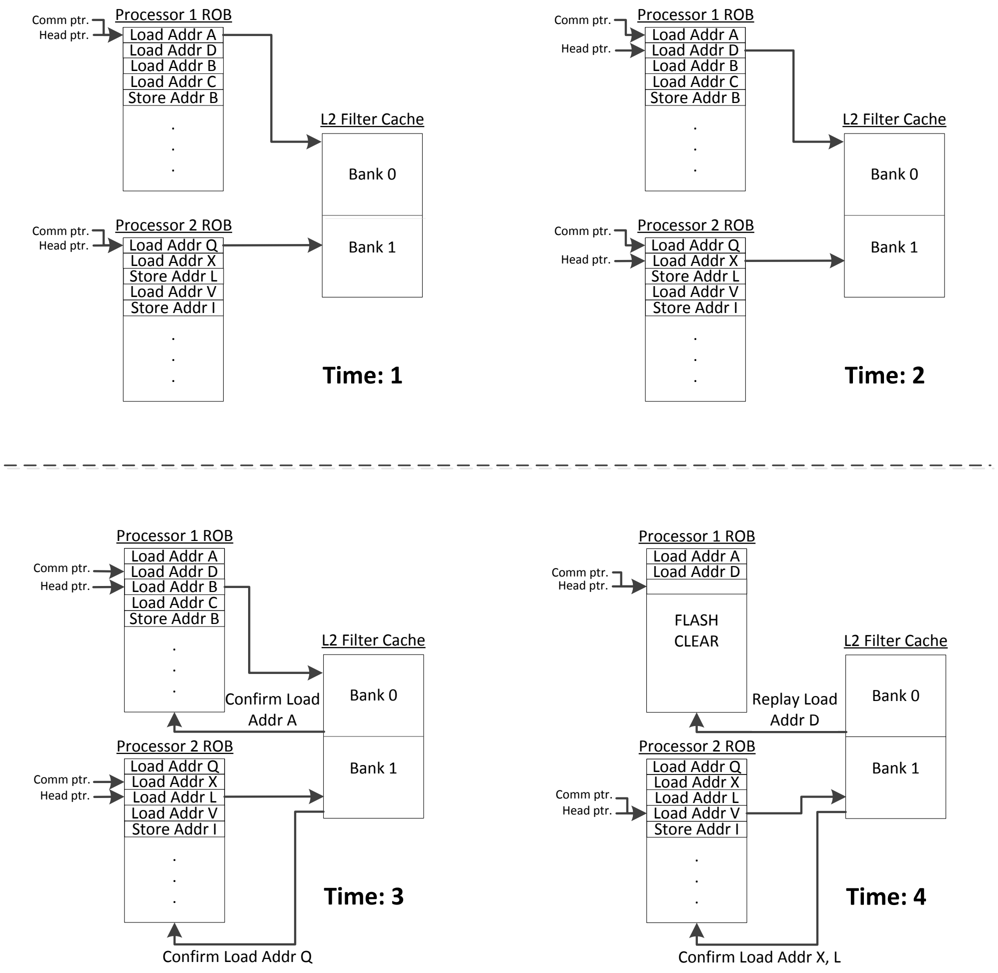


Figure 5.7: The e-PDEMI two-phase commit protocol for memory operation retirement

The e-PDEMI architecture is designed to perform memory speculation in the context of energy consumption as the primary design parameter, while also maintaining the high performance of a super-scalar out-of-order processor. The multi-core e-PDEMI architecture and

TBS Periodic Sampler Parameter	**Value**
nInstSkip	Tuned for each benchmark
nInstSkipThreads	0
nSampleMax	480M To allow for completion
nInstRabbit	2.53M
nInstWarmup	2.4M
nInstDetail	20K
nInstTiming	50K

Table 5.1: TBS Periodic Mode Sampler Parameter Values

memory hierarchy extend these benefits further by simplifying the complexity and energy cost typically associated with neccessary coherence protocols and mechanisms. As shown in the evaluation presented next in this chapter, while the e-PDEMI architecture can sustain the same high performance as the baseline without using a coherence protocol with private caches, adding a coherence protocol with private caches does not substantially improve the performance of the e-PDEMI system. In contrast, the baseline system cannot maintain its performance without the use of private caches and a coherence protocol over an interconnect. The baseline system cannot tolerate the additional latency associated with a shared cache hierarchy. Finally, the e-PDEMI architecture simplifies the programming model for programmers by supporting Sequential Consistency commonly accepted as the most intuitive memory consistency model.

5.2 Experimental Setup

The evaluation of the multi-core e-PDEMI system used a simulation setup similar to the infrastructure described in Chapter 4 execept that instead of a TASS-like [38] simulator, it instead used an extended multi-core version known as TBS [39]. TBS also modifies SESC [55] and uses a modified version of McPAT [47] for power estimation. The TBS sampler mode that was selected was the Periodic Sampler Mode. Table 5.1 shows the Periodic Sampler Mode parameters used for multi-core evaluation of the e-PDEMI architecture.

Table 5.2 lists the parameters used to simulate each core of the baseline architecture, and Table 5.3 lists the parameters used to simulate each core of the proposed architecture. All simulated multi-threaded benchmarks shown in Table 5.4 were run to completion, because using dynamic instruction based performance metrics such as Instructions per Cycle can often be

Baseline Architecture	
Issue Width	4 Instructions
Re-order Buffer	256 Instructions
Load Queue	48 Loads
Store Queue	32 Stores
Branch Predictor	OGEHL
BTB	4K entries - 4-way
Integer ALUs	4 units
Floating Point ALUs	4 units
L1 TLB	64 entries / Fully Assoc. / 1 cycle
L1 Data Cache	16KB / 4-way / 4 cycles Private, per-core
L2 Cache	512KB / 16-way / 7 cycles Private, per-core
L3 Cache	8MB / 32-way / 14 cycles 4 Banks

Table 5.2: Baseline architectural parameters

misleading for multi-threaded workloads due to dynamic generation of thread synchronization instructions such as locks. By simulating each benchmark to completion, performance is measured directly by comparing the execution time of each benchmark on the baseline architecture with the execution time of each benchmark on the e-PDEMI architecture.

5.3 Evaluation

Figure 5.8 shows the performance of the multi-core Baseline and e-PDEMI architectures in terms of benchmark completion times. Completion times are presented instead of IPC as noted in the previous section. Looking at Figure 5.8, the e-PDEMI architecture performance is essentially equivalent to the Baseline architecture, with a worst-case slowdown in $ocean$ of 8%. In some cases greater speed ups can be achieved, the greatest of which is $canneal$ at 14%. $canneal$ benefits from e-PDEMI's efficient handling of memory ordering instructions(membarrier, memfence, etc.). In the Baseline architecture, when a memory barrier, or memory fence instruction is encountered, the pipeline must be stalled, so that the ROB can be drained and all previously inflight instructions must be retired, before execution can continue. This mechanism is needed to guarantee that all store instructions prior to the memory barrier or

e-PDEMI Architecture	
Issue Width	4 Instructions
Re-order Buffer	256 Instructions
Branch Predictor	OGEHL
BTB	4K entries - 4-way
Integer ALUs	4 units
Floating Point ALUs	4 units
L1 TLB	64 entries / Fully Assoc. / 1 cycle
VPC Cache	16KB / 4-way / 4 cycles
VPC Buffer	64 entries
Mem Spec Buffer	48 Load entries, 32 Store entries
Filter Cache	4KB / 16-way /6 cycles
	Shared, Banked as many as cores
L2 Cache	512KB / 16-way / 7 cycles
	Shared, Banked as many as cores
L3 Cache	8MB / 32-way / 14 cycles
	4 banks

Table 5.3: Proposed e-PDEMI architectural parameters

Suite	Benchmark
PARSEC	blackscholes, bodytrack, canneal, x264
(simlarge input set)	fluidanimate, swaptions, facesim
SPLASH	fft, fmm, ocean, radix

Table 5.4: Simulated PARSEC and SPLASH Benchmarks

fence are globally visible to all instructions executing after the memory barrier or fence.

In the e-PDEMI architecture, when a memory barrier or memory fence instruction is encountered, e-PDEMI is not required to stall the pipeline to drain the ROB. The in-order commit of memory instructions to a shared memory hierarchy instead of private memory hierarchy guarantees that each Store instruction will be globally visible to any proceeding Load instruction at any point in time. This is the same mechanism that provides Sequential Consistency on the e-PDEMI architecture.

5.3.1 Private Cache Coherence Impact

Figure 5.9 shows that the Baseline architecture performance is particularly sensitive to the memory hierarchy configuration. Figure 5.9 explores the performance impact on both

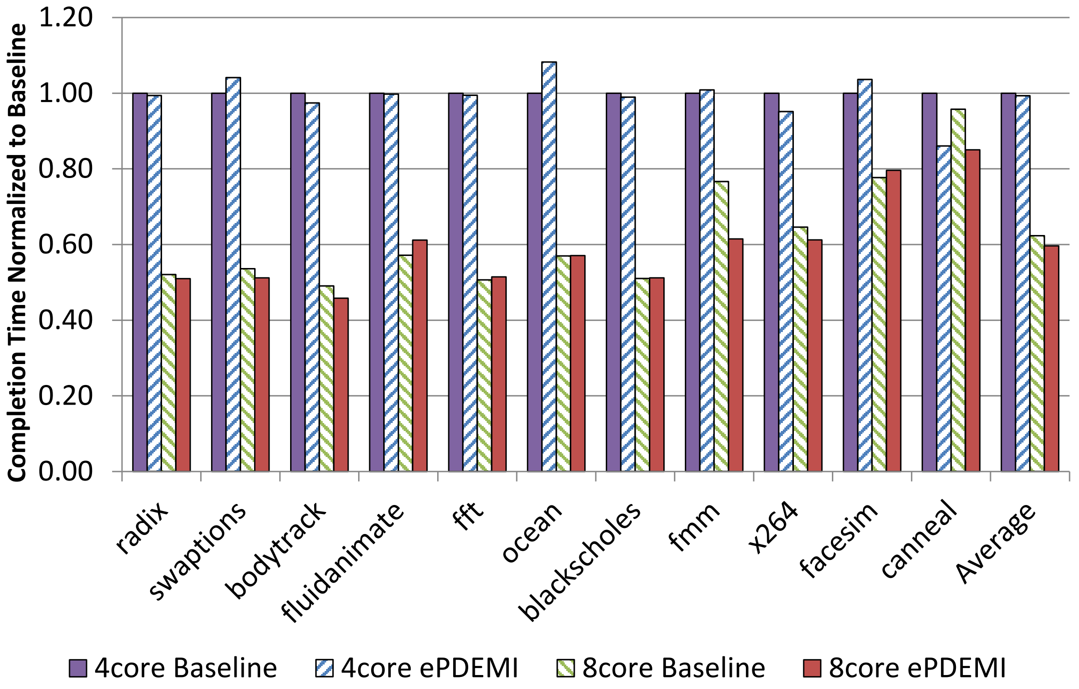


Figure 5.8: Multi-core e-PDEMI has the same performance as the baseline architecture

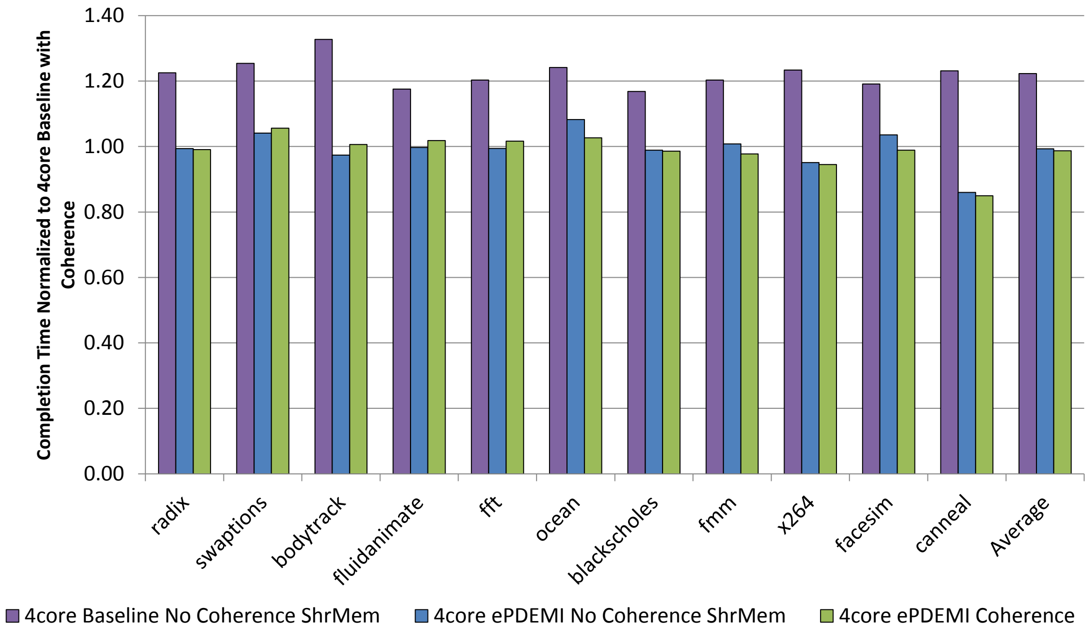


Figure 5.9: Private Cache Coherence doesn't improve e-PDEMI, but is critical for the Basline

architectures of having a shared memory hierarchy with no invalidation-based coherence as in the proposed e-PDEMI architecture, and having a private memory hierarchy with invalidation-based coherence as in the Baseline architecture. It can be seen that e-PDEMI appears mostly insensitive to this configuration choice, but the Baseline architecture performance suffers by over 20% without a private cache hierarchy with invalidation based coherence. The baseline architecture suffers because e-PDEMI's memory hierarchy accesses are off the critical path and therefore not sensitive to the additional contention latency inherent in a shared memory hierarchy. However, all of the Baseline architecture's memory hierarchy accesses are on the execution critical path and thus the additional latency of a shared memory hierarchy is not well tolerated.

5.3.2 Instruction Replay Impact

While the e-PDEMI architecture does increase replays, the proportion of instructions that are actually replayed is small on average, as shown in Figure 4.7 and Figure 5.10. Because the e-PDEMI architecture does not use StoreSets, it is more likely to trigger replays (due to misprediction) than the Baseline architecture. The e-PDEMI architecture increases the fraction of instructions replayed on average by 3.33% in the single-core implementation and 1.9% in the multi-core implementation. As shown in Figure 4.8, this increase in replays is mitigated by the e-PDEMI architecture experiencing slightly fewer stalls on average than the Baseline architecture. The e-PDEMI architecture experiences slightly fewer stalls because it does not experience Load Store Queue stalls, does not use StoreSets, and stops fetching new instructions in the event of a replay.

It is also important to note that, as was seen in Figure 4.3, the e-PDEMI architecture is able to recover any additional energy consumption it experiences due to increased replays by saving the additional energy per instruction associated with the Load Store Queue and StoreSets. Finally, Figure 5.8 shows that additional replays in the multi-core e-PDEMI architecture do not significantly impact its performance. In the case of $ocean$, an additional 6.3% dynamic instructions are executed due to replays, which only degrades performance by 8% in the worst case.

Figure 5.10 corresponds with Figure 4.7 showing that in the multi-core case e-PDEMI does not create significantly more dynamic instructions due to replay. On the multicore e-

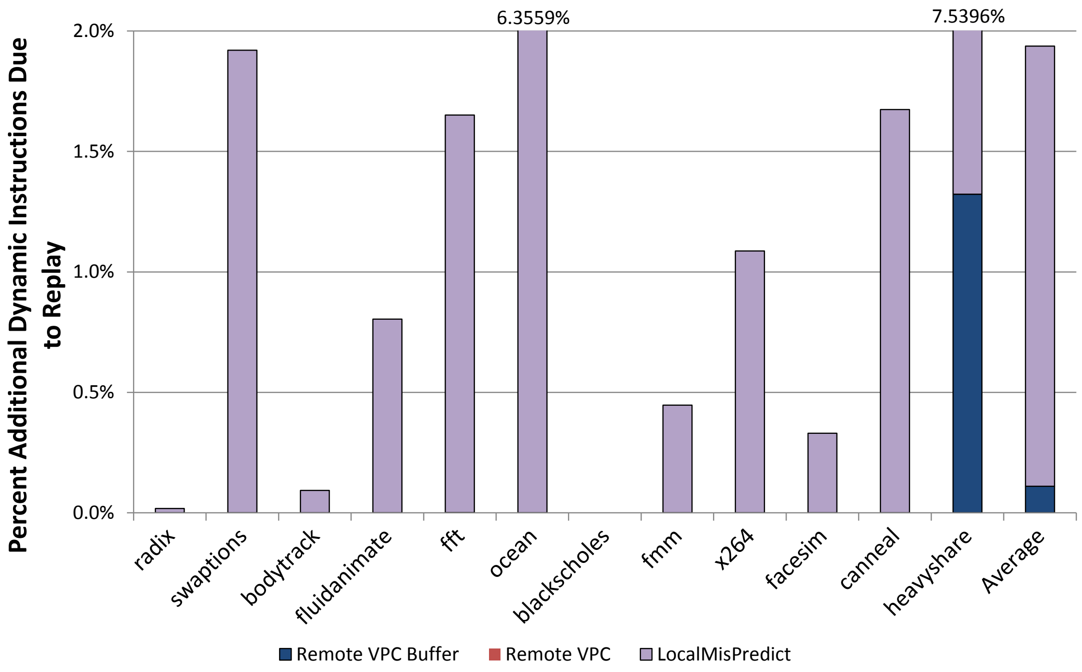


Figure 5.10: e-PDEMI has less than 2% for multi-core configurations increase in instructions executed on average as a result of memory replays

PDEMI architecture, there are three possible causes of replay. The first and most prevalent is the same as on the single-core e-PDEMI implementation, where local mispredictions in the VPC Buffer or VPC cause a load to consume incorrect speculative data. The other two replay types can occur due to the timing of remote data sharing. If a load on core 0 needs to consume the data stored by a store on core 1, because there is no communication or coherence between cores' VPCs, the load will only be able to consume the data once the Store has retired and committed its data to the shared memory hierarchy. The data the load requires at execution time could be in a remote VPC Buffer or VPC. Additionally, it is possible that the load could consume stale data from its own VPC that was written by a previous local store. These cases were specifically tracked and found to be rare.

In order to be sure there was no malfunction or oversight in the performed simulations, a a micro-benchmark was created (shown as "heavyshare" in Figure 5.10) to attempt to force this behavior by having threads simply increment the same counter in tight loop, requiring each thread to read the value of the counter and then write a new value. It was found that when executing this benchmark, replays due to data being present in a remote VPC Buffer or VPC account for 17.5% of total replays for a 4 threaded simulation as shown in Figure 5.10, and up to 23% of total replays for an 8 threaded simulation. For the rest of the simulated PARSEC benchmarks, we see that although local replays increase moderately, remote triggered replays do not become a substantial contributor. Because e-PDEMI supports sequential consistency and uses a shared memory hierarchy across cores, store data becomes globally visible in program order. As such, either a load will hit in its local VPC (either consuming the correct data or triggering a replay), or a load will miss in its local VPC and the data it consumes from the memory hierarchy will be sequentially updated data.

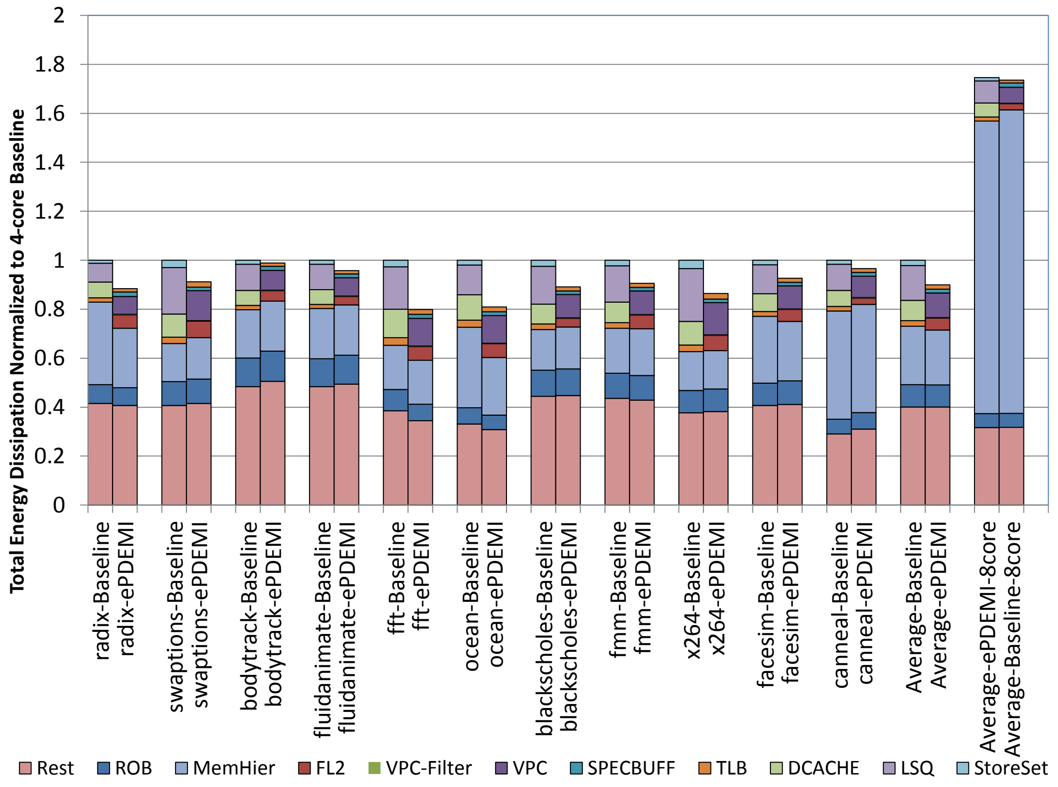


Figure 5.11: Quad-core e-PDEMI has 10% Average Energy Savings

Figure 5.11 illustrates total energy consumption for each multicore benchmark running on both the multi-core baseline architecture and proposed multi-core e-PDEMI architecture as described in Section 5.1. The most important insight Figure 5.11 provides is that the e-PDEMI architecture reduces total energy consumption across all benchmarks. The average energy per instruction reduction is 10% in the multicore case. The "Rest" of the core energy consumption is equivalent between the Baseline and e-PDEMI systems due to the stall trading behavior discussed in Chapter 4 and shown in Figure 4.8. As in the single-core case, this may seem counter intuitive because memory instructions must be verified, but also as in the single core case, verification does not impact register file nor re-order buffer energy consumption with respect to the baseline. The relatively small memory speculation buffer is again relied upon to store the additional data required by the memory operation verification process. This avoids increasing pressure and activity rates for the re-order buffer or register file which could increase energy consumption.

Because the multi-core e-PDEMI system generates a very small overhead of additional dynamic instructions executed due to replays as shown in Figure 5.10, the additional energy cost to the structures such as the re-order buffer and rename logic is small.

The multi-core e-PDEMI architecture again re-balances memory hierarchy structures to avoid increasing energy consumption. With memory hierarchy accesses off the critical path, the multi-core e-PDEMI architecture is able to tolerate the longer latencies associated with a unified shared cache hierachy as shown in Figure 5.1. As in the single core case, the addition of the L2 Filter cache avoids potentially energy costly accesses to the L2 cache during memory operation verification. To contend with contention from multiple cores in the multi-core e-PDEMI architecture, the L2 Filter cache becomes a banked structure, matching the number of banks to the number of cores as shown in Table 5.3

The baseline architecture was also simulated with an L2 Filter cache to evaluate its sensitivity to such a design choice. There was no appreciable performance impact, but a small increase in energy consumption. An L2 Filter cache is therefore not included in the Baseline architecture. In the e-PDEMI architecture, many of the L2 Filter accesses are memory verification requests which are off the critical path. Therefore, e-PDEMI is able to tolerate the additional potential latency associated with adding a new structure into the memory hierarchy. The VPC Buffer and VPC collectively have a high hit rate which, although does not guarantee that correct data is consumed by load instructions, does prevent accesses to the Filter L2 cache. In a multicore implementation, the Baseline architecture is required to maintain coherence between private L2 Filter caches whereas, because e-PDEMI uses a multi-banked shared L2 Filter cache, no coherence accesses are required. The drawback to making each structure in the memory hierarchy a multi-banked shared structure across the e-PDEMI cores is that the total leakage energy is increased, which accounts for the larger proportion of “MemHier” energy consumption shown in Figure 5.11.

Neither the baseline nor e-PDEMI architectures experience any significant energy per instruction in the L2 or L3 caches (labeled “MemHier”) in the single core case shown in Figure 4.3.

As in the single-core case, the multi-core e-PDEMI system benefits from the VPC being more energy efficient than the L1 cache with the same size and organization. Additionally, the same VPC optimizations discussed in Chapter 4 contribute to reducing the multi-core e-PDEMI system’s total energy consumption such as using the VPC buffer to reduce main VPC access rates, not allowing write allocations, and not displacing speculative data from the VPC buffer to the VPC.

Chapter 6

Conclusions

Memory speculation in modern out-of-order processors is vital to their high performance. Because of ordering requirements between memory instructions, many energy consuming structures have been proposed to avoid those hazards. The most common of these are the fully associative Load and Store Queues. In addition to consuming significant energy, their fully associative natures also make these structures difficult to scale.

The introduction of multi-core processors, with their need for coherent shared memory, has motivated many protocols and mechanisms to facilitate the management of coherent shared and private memory structures. That management is often carried out over an interconnection network.

In addition to the complexity of coherence protocols and networks, multi-core programmers must be accutely aware of the memory consistency models that their target architectures support. Commonly, programmers must use explicit program semantics in order to guarantee the ordering of their programs' memory operations. As core densities increase, the probability for heavily threaded programs to experience transient bugs that can sometimes manifest and sometimes not manifest becomes greater and the programmer is left with a choice between repeatable deterministic program execution or high-performance program execution with potentially non-deterministic results due to multi-core memory data races.

The e-PDEMI architecture has been designed as a novel approach to out-of-order memory speculation with energy consumption as the first design constraint. It simplifies out-of-order processors' memory sub-system implementation and delivers straight-forward memory disambiguation. Supporting energy efficiency, e-PDEMI focusses on simple, reduced-complexity mechanisms to support its ability to execute memory operations in a high-performance out-of-order Super Scalar processor.

e-PDEMI consists of a Virtual Predictive Cache and filter caches with low complexity. The Load Store Queue and StoreSets are removed in favor of a decoupled memory speculation mechanism with in-order commit and verification. e-PDEMI reduces overall processor energy by 16.4% on average with no average performance impact and up to a 14% speedup in multi-core applications with frequent memory fences or barriers.

In-order verification, novel two-phase commit, and an address mapped cache hierarchy make stores appear globally in program order, supporting Sequential Memory Consistency. This work simplifies the implementation of an out-of-order processor's memory sub-system and provides straight forward memory disambiguation, removes the coherence protocol and interconnect, and supports Sequential Consistency.

Simulation using the SPEC2006 and PARSEC benchmark suites demonstrates that the e-PDEMI architecture provides equivalent performance to a state of the art out-of-order processor and saves energy.

Chapter 7

Future Work

7.1 ePDEMI for High Core Denisities

This work shows that ePDEMI supports sequential consistency and does not require a coherence mechanism while maintaining the performance of the baseline architecture. The presented performance evaluations appear to scale over one, four, and eight cores. Additionally, ePDEMI's sequential memory consistency model and non-reliance on a coherence protocol are not impacted by the number of cores and therefore pose no scaling challenges to ePDEMI. However, ePDEMI's evaluated energy benefits appear to not to scale as well as the other metrics measured in this work.

Figure 5.11 in Section 5.3.2 shows that the energy benefits of ePDEMI begin to diminish as the number of cores increases. This effect is attributable to the total dynamic and leakage energy for the L2 and L3 caches which begin to dominate the total processor energy consumption as core counts incease. As a result, although ePDEMI continues to achieve significant power savings in the VPC and related structures with respect to the baseline processor's LSQ and related structures, the energy consumption that those structures account for on both ePDEMI and the baseline processor becomes a smaller fraction of the total processor energy dissipation due to the leakage power of the memory hierarchy. This trend is intuitive because as the number of cores increases on both architectures, the size of the L2 and L3 cache structures also increases, ultimately increasing the leakage energy of those larger memory structures. On the ePDEMI architecture, this is due to each shared memory structure's number of banks linearly scaling with the number of cores. On the baseline architecture, this trend comes from the addition of private L2 caches and the scaling up of the shared L3 cache as core count is increased. In both cases, the amount of total cache area increases with the number of cores and thus the total leakage power for the cache hierarchy increases as well. Because the memory speculation structures' energy consumption becomes a small fraction of total processor energy consumption as core counts increase, this suggests that the ePDEMI design may need to be extended to maintain the energy benefits shown in this work at higher core counts on a single multiprocessor die.

7.1.1 Cache Area Management

A potential area of investigation would be to evaluate the effect of not linearly scaling the number of banks for each structure in ePDEMI's shared cache hierarchy linearly with the number of cores. If this technique were to not significantly degrade performance, ePDEMI might be able to continue to scale its existing energy benefits effectively to higher core counts, and achieve greater energy benefits than shown in this work. This potential energy reduction opportunity is due to ePDEMI's shifting of memory hierarchy accesses off the critical path and the resultant memory access latency tolerance. Intuitively, the effect of not scaling the number of banks in ePDEMI's shared memory hierarchy with the number of cores in a multiprocessor ePDEMI system would be to increase contention for those banks amongst the cores. As the number of cores increased, so too would bank contention. The increased bank contention could lead to cores experiencing increased latency on each of their accesses to the memory hierarchy. However, because ePDEMI uses the two phase commit protocol described in Section 5.1.6, the ePDEMI retirement phase for memory instructions can tolerate increased latency and such a configuration might be able to maintain equivalent performance while not suffering the increased leakage energy consumption due to the additional banks scaling linearly with ePDEMI core counts. Such a study would need to examine the various bank provisioning options for ePDEMI multiprocessor systems of core count going beyond the eight core configuration shown in this work. Options for bank provisioning could include simple provisioning schemes such as having fixed ratios of core to banks i.e. 2:1, 3:1, etc. where several ratios could be evaluated.

Additionally, such a multiprocessor ePDEMI system could provide an equal number of banks to cores, but keep many of the banks in an "off" state until some criterion were met, stimulating the activation of additional banks under some circumstances but not all. Such an evaluation would need to consider several different activation criteria, and perhaps different methodologies of detecting those criteria.

Another related study could be the evaluation of allowing cores to have priority access to certain banks. If less banks than cores were provided, leading to increased bank contention as suggested above, one possible solution could be to give certain cores priority access to certain banks to maintain high performance. Several priority algorithms could be devised and evaluated, and potentially could compliment the bank partitioning schemes discussed above to not only maintain performance but also achieve higher energy savings for ePDEMI than shown in this work.

Each of the proposed evaluations above would not be complete without a similar investigation of the proposed techniques in the baseline multiprocessor architecture. Intuitively the baseline architecture cannot tolerate additional latency in its memory hierarchy because its memory hierarchy is on the critical path. However this assertion must be confirmed by completing the necessary evaluation. In the case of the baseline multiprocessor architecture, each core has a private L1 and L2 data cache and the shared L3 data cache size is scaled with the number of cores. This work has shown already that the baseline architecture cannot tolerate a fully shared memory hierarchy amongst cores. However, evaluating whether the baseline architecture could equivalently tolerate a smaller cache area, as proposed for ePDEMI above, would involve grouping cores to share particular private L1 and L2 data caches. Many permutations could be studied. For example, every core could have a private L1 data cache, but L1 caches could share L2 caches in pairs, or every two cores could share one L1 data cache, and so on. The completion of this study could provide insights into possible techniques to scale the benefits of the ePDEMI architecture to core counts greater than eight and possibly show that given a well-performing cache size configuration, ePDEMI could potentially increase its reduction of energy consumption more than shown in this work if the baseline processor could not tolerate a similar cache size configuration.

7.1.2 ePDEMI Clustering

ePDEMI might employ a clustering configuration to enable scaling of the energy benefits discussed in this work. Figure 7.1 shows such a clustered ePDEMI architecture. In this configuration, several ePDEMI cores are grouped together and share a memory hierarchy as described in Section 5.1.4. However, as shown in Figure 7.1, several of these ePDEMI clusters could be combined to form a much larger ePDEMI CMP consisting of total core counts beyond sixteen. Significant investigation would be required to evaluate such a scaling technique.

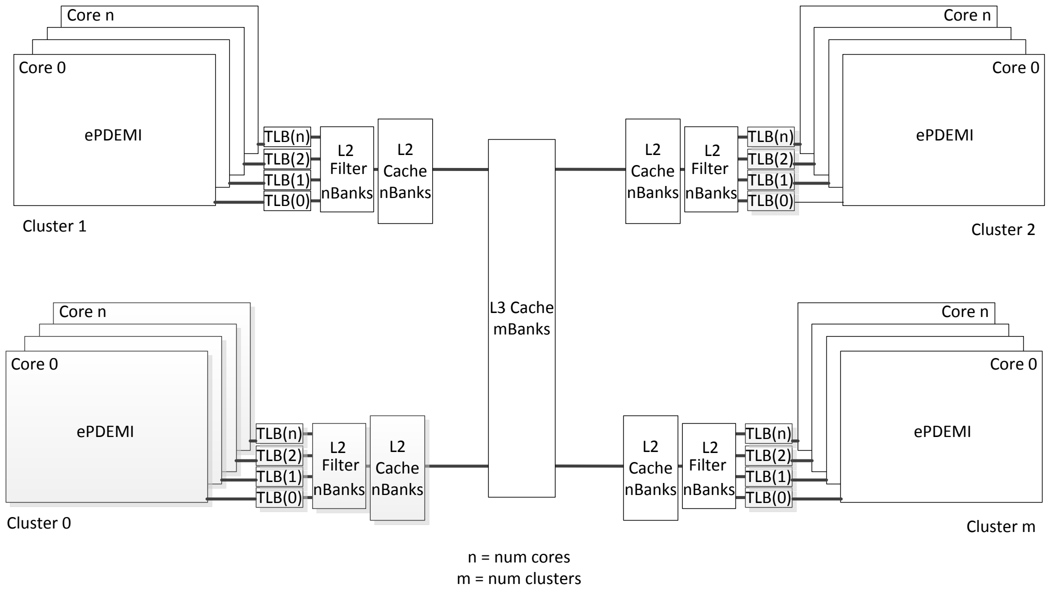


Figure 7.1: ePDEMI Clustered Architecture

One potential area of study could be an analysis of the tradeoffs between enabling a hierarchical address mapped shared memory hierarchy between clusters and a high level cache coherence protocol between clusters. Such systems are closely related to systems referred to as a *Distributed Shared Memory System* [50]. In the case of a hierarchical address mapped shared memory, each cluster would consist of several ePDEMI cores with the shared address mapped memory subsystem discussed in this work. Each cluster would be assigned a region of the system's total physical address space as shown in Figure 7.2. When the memory hierarchy for a given core missed on an address outside of its cluster's assigned address space, it would initiate a request to the home cluster for that memory address. The home cluster would then provide the requested data. In this way, the ePDEMI shared memory hierarchy paradigm would be maintained. However, it is likely that such a design could lead to severe memory access latencies for clusters accessing addresses outside of their home address space. The first evaluation step would consider whether ePDEMI's existing memory latency tolerance could tolerate that additional latency without performance degradation. If the increased memory access latency did degrade performance, additional investigation of designs to avoid that latency could be conducted.

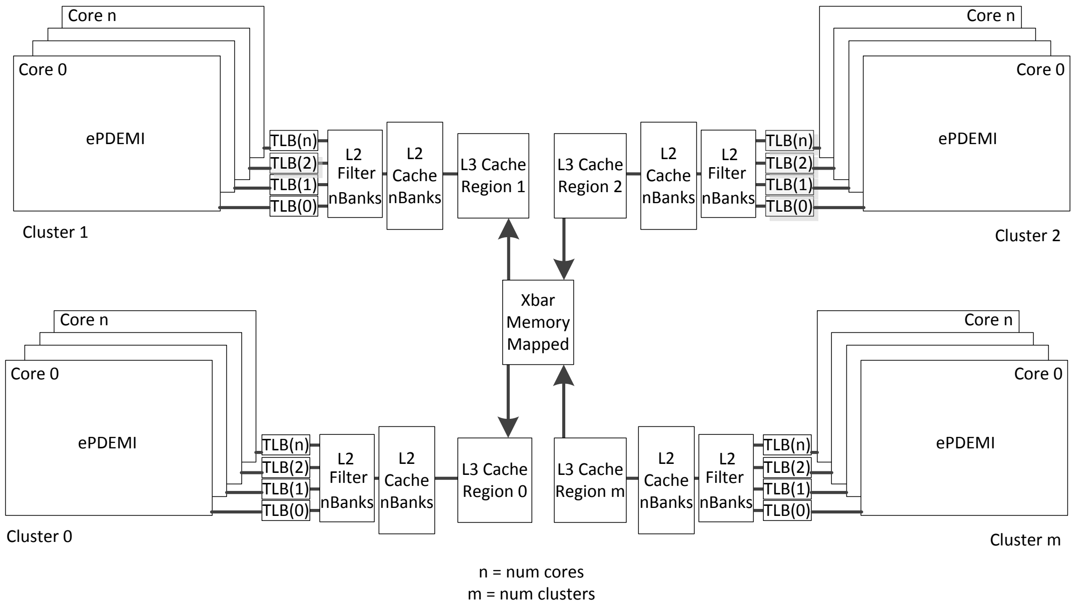


Figure 7.2: ePDEMI Clustered Architecture with Memory Mapping

Instead of mapping the ePDEMI clustered system's physical address space across each of the clusters, Figure 7.3 shows a configuration where each cluster could be allowed to cache copies of any accessed memory addresses, effectively making each cluster's memory hierarchy private to that cluster of cores. In such a case, as discussed in Section 5.1.3, cache coherence would become necessary between clusters. An investigation of which cache coherence implementation would provide the best performance and energy efficiency given ePDEMI's latency tolerance would be essential.

Additionally, it could be possible to arrange ePDEMI clusters into a hierarchal design where groups of clusters could share a private memory hierarchy, becoming members of the same coherence domain as shown in Figure 7.4, which would allow for a coarser grain cache coherence granularity at the expense of memory access latency inside a given coherence domain. Inside of each coherence domain, the full physical memory address space would be mapped across the banks of the shared memory structures. Accesses would proceed as discussed above

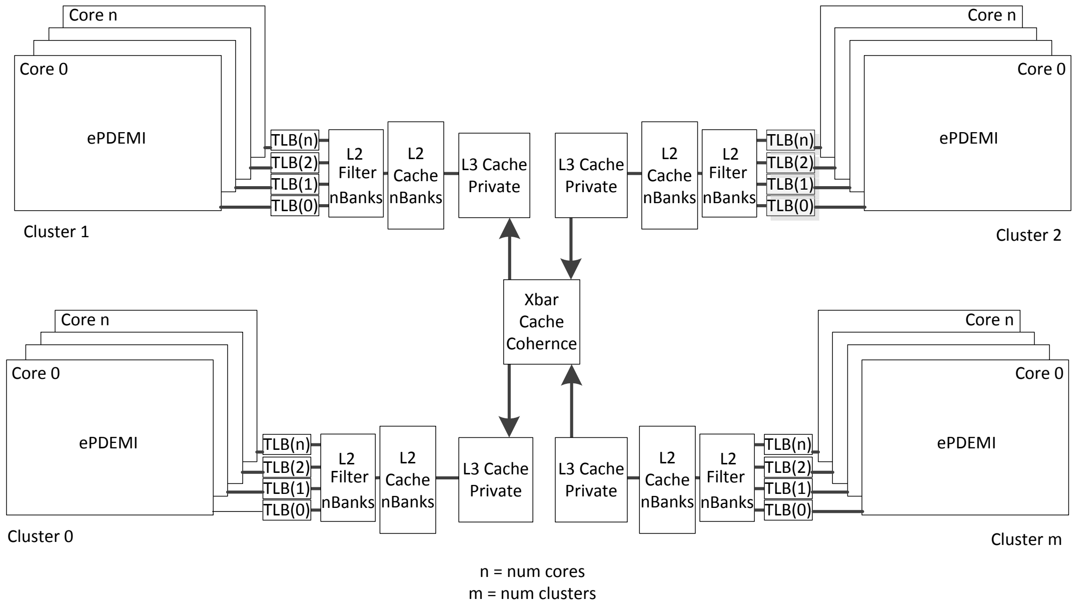


Figure 7.3: ePDEMI Clustered Architecture with Coherence

in the fully memory mapped clustered architecture, with the exception that for a given miss inside a coherence domain, that domain would first attempt a coherence operation with other coherence domains before triggering an access to main memory. Such a study could evaluate how many clusters could be members of a coherence domain before inter-domain cross-cluster memory access latency became too long to maintain performance.

7.1.3 Page Migration

A different approach to reducing memory access latency across clusters could be to migrate virtual memory pages between clusters actively working with the associated physical addresses. ePDEMI cores could be grouped into clusters with the full system physical address space mapped across the clusters. However, the additional access latency described previously could be mitigated by detecting performance degradation and changing the address map across the clusters appropriately. For example, if cores in cluster zero were continually accessing memory addresses from a virtual page assigned to cluster one, the system could migrate that virtual page to cluster zero's memory hierarchy and remap the associated addresses to cluster zero. The effect of such a migration would be that cluster zero's future accesses would be

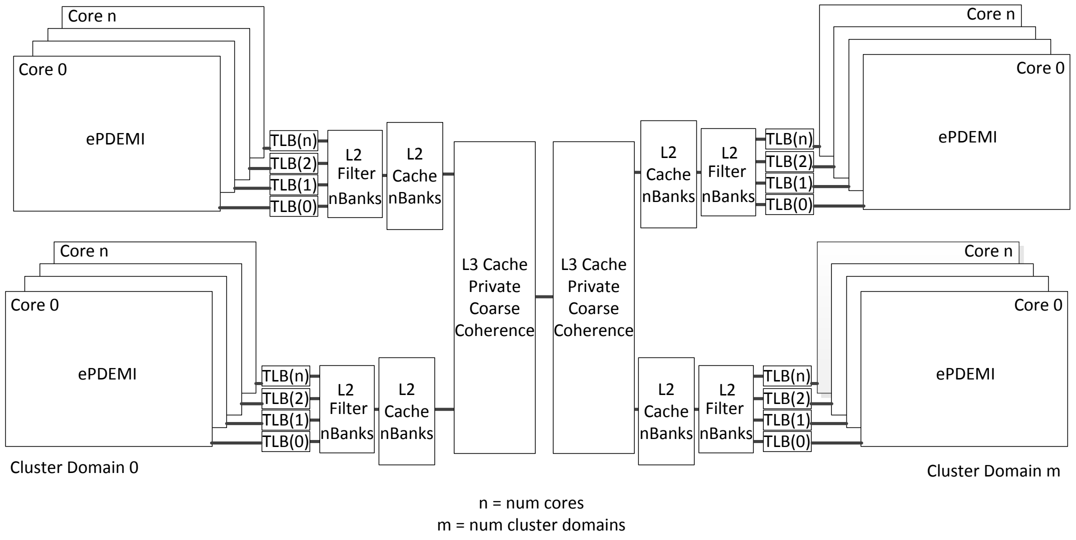


Figure 7.4: ePDEMI Clustered Domains Architecture with Coarse Grain Coherence

local accesses and would thus avoid long latency inter-cluster accesses. The tradeoff of such a technique would be that any future accesses from cluster one to that page would then be remote. The cost of this tradeoff as well as how to decide when to migrate a page would be the central questions to be answered by a future study.

7.1.4 Future Work Challenges

A challenge to further evaluating the scalability to ePDEMI beyond sixteen cores is the availability of benchmarks with workloads that would sufficiently stress such large systems. If the clustered systems discussed previously consisted of sixteen cores per cluster, a four cluster system would be composed of 64 cores with approximately 64MB of cache capacity between the L2 and L3 caches in the system. However, the multi-threaded shared-memory workloads presented in this work are not designed to scale to so many threads, particularly because their working set sizes could easily fit within such a sizable amount of cache storage, and are not large enough to require such a large number of worker threads. As a result, it would be crucial, before pursuing the future work outlined above, to identify several new workloads designed to scale beyond sixteen threads, ideally with very large working set sizes requiring substantially more worker threads than those presented in this work. A possible related future study could consider what applications could be scaled beyond sixteen threads with large workloads that would be relevant to various usage scenarios.

This work has shown that the ePDEMI architecture provides sequential consistency and removes coherence for single chip multiprocessor (CMP) dies consisting of one, four, and eight cores. As discussed previously, the ePDEMI energy benefit may face a scaling challenge due to leakage energy scaling as core count increases. However, although there exists today systems consisting of tens and even hundreds of processor cores, there are very few single die CMPs that consist of many more cores than have been studied in this work. This is most true for desktop CMPs, although server CMPs have not significantly surpassed eight cores in recent years. Figure 7.5 shows the progression of the highest core count available CMP processors from Intel, AMD, IBM, and Oracle (formerly Sun) over the last 20 years. Today, the most cores available in an Intel high-performance out-of-order superscalar CMP comes in the server-class Xeon E7-8870 with ten cores. AMD offers 16 cores in its Opteron 6386SE. Oracle offered up to 16 cores in the SPARC T3 processor released in 2006, but in 2011 Oracle decreased the number of cores provided in its new SPARC T4 processor to eight. In 2013, Oracle will release the SPARC T5 processor which will feature 16 cores once again. IBM currently offers its POWER7 processors with up to eight cores. From surveying today's high-end server processor market space and reviewing the market over the last 20 years as shown in Figure 7.5, it is clear that per-die CMP core counts are not increasing as rapidly as they did in the previous decade. Therefore, although scalability is an area of future investigation for ePDEMI, it is important to note that the work presented in this dissertation is not only relevant today, but appears to be relevant to the observed trend of the current high-performance multiprocessor marketplace.

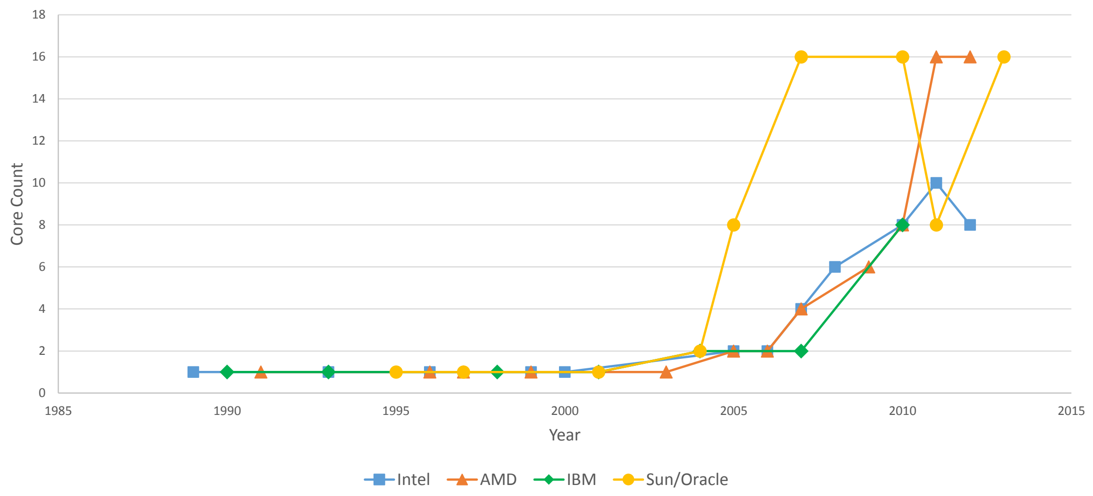


Figure 7.5: Core Counts in High Performance Processors Over Last 20 Years

Bibliography

[1] *IBM System/370 Principles of Operation*. Number GA22-7000-9. IBM, May 1983.

[2] *The SPARC Architecture Manual Version 8*. SPARC International Inc., 1992.

[3] 1003.1 standard for information technology portable operating system interface (posix) rationale (informative). pages i–310, 2001.

[4] HyperTransport technology I/O link - white paper. Whitepaper, July 2001.

[5] D. Abts, S. Scott, and D.J. Lilja. So many states, so little time: Verifying memory coherence in the cray x1. In *in the Cray X1;, International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[6] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Vol2 'System Programming'*, 2012.

[7] S. Adve and M. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 1–13, May 1990.

[8] S.V. Adve, M.D. Hill, et al. Implementing sequential consistency in cache-based systems. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume 1, pages 47–50, 1990.

[9] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Trans. Program. Lang. Syst.*, 15(1):182–205, Jan. 1993.

[10] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May. 1994.

[11] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX, 2005.

[12] Geoffrey M. Brown. Asynchronous multicaches. *Distributed Computing*, 4:31–36, 1990.

[13] H.W. Cain and M.H. Lipasti. Memory ordering: a value-based approach. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 90 – 101, june 2004.

[14] Aaron Carpenter, Jianyun Hu, Jie Xu, Michael Huang, and Hui Wu. A case for globally shared-medium on-chip interconnect. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 271–282, New York, NY, USA, 2011. ACM.

[15] F. Castro, L. Pinuel, D. Chaver, M. Prieto, M. Huang, and F. Tirado. DMDC: Delayed Memory Dependence Checking through Age-Based Filtering. In *International Symposium on Microarchitecture*, pages 297–308, Orlando, Florida, Dec. 2006.

[16] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: bulk enforcement of sequential consistency. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 278–289, New York, NY, USA, 2007. ACM.

[17] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[18] F. Castroand D. Chaver, L. Pinuel, M. Prieto, F. Tirado, and M. Huang. Load-store queue management: an energy-efficient design based on a state-filtering mechanism. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 617 – 624, oct. 2005.

[19] G. Z. Chrysos and J. S. Emer. Memory dependence Prediction Using Store Sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, pages 142–153. IEEE Computer Society, 1998.

[20] J.M. Corella, F. Stone and C.M. Barton. A formal specification of the powerpc shared memory architecture. *IBM*, 1993.

[21] ARM Corporation. Arm architecture reference manual, section a3.8, 2007.

[22] Érika Cota, Luigi Carro, and Marcelo Lubaszewski. Reusing an on-chip network for the test of core-based systems. *ACM Trans. Des. Autom. Electron. Syst.*, 9(4):471–499, Oct. 2004.

[23] W.J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Design Automation Conference, 2001. Proceedings*, pages 684 – 689, 2001.

[24] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th annual international symposium on Computer architecture*, ISCA '86, pages 434–442, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[25] C. Fensch and M. Cintra. An os-based alternative to full hardware coherence on tiled cmps. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 355 –366, feb. 2008.

[26] A. Gandhi, H. Akkary, R. Rajwar, S. Srinivasan, and K. Lai. Scalable Load and Store Processing in Latency Tolerant Processors. In *International Symposium on Computer Architecture*, pages 446–457, Madison, Wisconsin, Jun. 2005.

[27] A. Garg, M. Rashid, and M. Huang. Slackened Memory Dependence Enforcement: CombiningOpportunistic Forwarding with Decoupled Verification. In *International Symposium on Computer Architecture*, pages 142–153, Boston, Massachusetts, Jun. 2006.

[28] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 15–26, New York, NY, USA, 1990. ACM.

[29] Chris Gniady, Babake Falsafi, and T. N. Vijaykumar. Is sc + ilp = rc? *SIGARCH Comput. Archit. News*, 27(2):162–171, May. 1999.

[30] Jos Gonzlez and Antonio Gonzlez. Memory address prediction for data speculation. In *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 1084–1091. Springer Berlin / Heidelberg, 1997.

[31] Kees Goossens, John Dielissen, Om Prakash Gangwal, Santiago Gonzalez Pestana, Andrei Radulescu, and Edwin Rijpkema. A design flow for application-specific networks on chip with guaranteed performance to accelerate soc design and verification. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 1182–1187, Washington, DC, USA, 2005. IEEE Computer Society.

[32] L. Gwennap. Sandy bridge spans generations. *Microprocessor Report (www. MPRonline. com)*, 2010.

[33] A. Hasegawa, I. Kawasaki, K. Yamada, Si. Yoshioka, Si. Kawasaki, and Pi. Biswas. Sh3: High code density, low power. *IEEE Micro*, 15(6):11–19, Dec. 1995.

[34] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[35] Derek Hower. Acoherent shared memory. PhD Dissertation. University of Wisconsin-Madison, July 2012.

[36] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency : A bridge between release consistency and entry consistency. In *In Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 277–287, 1996.

[37] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2009.

[38] E K. Ardestani, E. Ebrahimi, G. Southern, and J. Renau. Thermal-aware Sampling in Architectural Simulation. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pages 33–38, New York, NY, USA, 2012. ACM.

[39] E. K. Ardestani and J. Renau. ESESC: A Fast Multicore Simulator Using Time-Based Sampling. In *International Symposium on High Performance Computer Architecture*, HPCA'19, 2013.

[40] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th annual international symposium on Computer architecture*, ISCA '92, pages 13–21, New York, NY, USA, 1992. ACM.

[41] J. Kin, M. Gupta, and W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. In *International Symposium on Microarchitecture*, pages 184–193, Research Triangle Park, North Carolina, Dec. 1997.

[42] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Annual International Symposium on Computer Architecture*, May 1981.

[43] Rajesh Kumar and Glenn Hinton. A family of 45nm IA processors. In *ISSCC*, pages 58–59. IEEE, 2009.

[44] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 408–419, Washington, DC, USA, 2005. IEEE Computer Society.

[45] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. Computers*, C-28(9):690–691, 1979.

[46] K. Lepak and M. Lipasti. Silent Stores for Free. In *International Symposium on Microarchitecture*, pages 22–31, Monterey, California, Dec. 2000.

[47] S. Li, J.H. Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 469–480. IEEE, 2009.

[48] Wei Lin, DongRui Fan, He Huang, Nan Yuan, and XiaoChun Ye. A low-complexity synchronization based cache coherence solution for many cores. In *Computer and Information Technology, 2009. CIT '09. Ninth IEEE International Conference on*, volume 1, pages 69 –75, Oct. 2009.

[49] Masaaki Mizuno, Michel Raynal, and Z. Zhou, Junhui. Sequential Consistency in Distributed Systems : Theory and Implementation. Rapport de recherche RR-2437, INRIA, 1995.

[50] C. Morin and I. Puaut. A survey of recoverable distributed shared virtual memory systems. *Parallel and Distributed Systems, IEEE Transactions on*, 8(9):959 –969, sep 1997.

[51] Andreas Moshovos and Gurindar S. Sohi. Speculative memory cloaking and bypassing. *International Journal of Parallel Programming*, 27:427–456, 1999.

[52] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical Report HPL-2009-85, April 2009.

[53] Gianluca Palermo and Cristina Silvano. Pirate: A framework for power/performance exploration of network-on-chip architectures. In Enrico Macii, Vassilis Paliouras, and Odysseas Koufopavlou, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, volume 3254 of *Lecture Notes in Computer Science*, pages 521–531. Springer Berlin Heidelberg, 2004.

[54] Yan Pan, Prabhat Kumar, John Kim, Gokhan Memik, Yu Zhang, and Alok Choudhary. Firefly: illuminating future network-on-chip with nanophotonics. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 429–440, New York, NY, USA, 2009. ACM.

[55] J. Renau, F. Basilio, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, 2005. http://sesc.sourceforge.net.

[56] Amir Roth. Store vulnerability window (svw): Re-execution filtering for enhanced load optimization. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 458–468, Washington, DC, USA, 2005. IEEE Computer Society.

[57] C. Scheurich and M. Dubois. Correct Memory Operation of Cache-Based Multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 234–243, Jun 1987.

[58] C. E. Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors*. PhD thesis, Dept. of Computer Engineering, University of Southern California, May. 1989.

[59] S. Sethumadhavan, R. Desikan, D. Burger, and C. Mooreand S. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *International Symposium on Microarchitecture*, pages 399–410, San Diego, California, Dec. 2003.

[60] S. Sethumadhavan, F. Roesner, D. Burger J. Emer, and S.Keckler. Late-Binding: Enabling Unordered Load-Store Queues. In *International Symposium on Computer Architecture*, pages 347–357, San Diego, California, Jun. 2007.

[61] Tingting Sha, Milo M. K. Martin, and Amir Roth. Scalable store-load forwarding via store queue index prediction. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 159–170, Washington, DC, USA, 2005. IEEE Computer Society.

[62] Tingting Sha, Milo M. K. Martin, and Amir Roth. Nosq: Store-load communication without a store queue. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 285–296, Washington, DC, USA, 2006. IEEE Computer Society.

[63] Wein-Tsung Shen, Chih-Hao Chao, Yu-Kuang Lien, and An-Yeu (Andy) Wu. A new binomial mapping and optimization algorithm for reduced-complexity mesh-based on-chip network. In *Proceedings of the First International Symposium on Networks-on-Chip*, NOCS '07, pages 317–322, Washington, DC, USA, 2007. IEEE Computer Society.

[64] K. Shim, M. Lis, O. Khan, and S. Devadas. Thread migration prediction for distributed shared caches. *Computer Architecture Letters*, PP(99):1, 2012.

[65] R. L. Sites. Alpha AXP Architecture. *Digital Technical Journal*, 4(4), 1992.

[66] S. Stone, K. Woley, and M. Frank. Address-Indexed Memory Disambiguation and Store-to-LoadForwarding. In *International Symposium on Microarchitecture*, Barcelona, Spain, Dec. 2005.

[67] Samantika Subramaniam and Gabriel H. Loh. Fire-and-forget: Load/store scheduling with no store queue at all. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 273 –284, dec. 2006.

[68] Xuan-Tu Tran, Yvain Thonnart, Jean Durupt, Vincent Beroulle, and Chantal Robach. A design-for-test implementation of an asynchronous network-on-chip architecture and its associated test pattern generation and application. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '08, pages 149–158, Washington, DC, USA, 2008. IEEE Computer Society.

[69] B. Vermeulen, J. Dielissen, K. Goossens, and C. Ciordas. Bringing communication networks on a chip: test and verification implications. *Comm. Mag.*, 41(9):74–81, Sept. 2003.

[70] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9.* Prentice Hall, Englewood Cliffs, N.J., 1994.

[71] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation Techniques for Improving Load Related InstructionScheduling. In *International Symposium on Computer Architecture*, pages 42–53, Atlanta, Georgia, May. 1999.

[72] X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Sahan. A case for software managed coherence in many-core processors. In *Poster on 2nd USENIX Workshop on Hot Topics in Parallelism HotPar10s*, 2010.