2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

Dynamic Memory Dependence Predication

Zhaoxiang Jin
Department of Computer Science
Michigan Technological University
Houghton, USA
zjin3@mtu.edu

Soner Önder
Department of Computer Science
Michigan Technological University
Houghton, USA
soner@mtu.edu

***Abstract*—Store-queue-free architectures remove the store queue and use memory cloaking to communicate in-flight stores instead. In these architectures, frequent mispredictions may occur when the store to load dependencies are inconsistent. We present DMDP (Dynamic Memory Dependence Predication) which modifies the microarchitecture behavior for such loads to mitigate memory dependence mispredictions. When a given dependence is hard to predict, i.e., a given load occasionally depends on a particular store, but it is independent at other times, DMDP predicates the load so that the address of the load is compared with the address of the predicted store to compute a predicate. This predicate guides the load to obtain the value from either the cache or the colliding store.**

The predication provided by DMDP i) enables the loads and their dependent instructions to execute much earlier, ii) reduces the hardware complexity of store-queue-free mechanisms, and iii) reduces the number of mispredictions. DMDP outperforms a state-of-the-art store-queue-free architecture by 7.17% on Integer benchmarks and 4.48% on Float benchmarks in our Spec 2006 evaluation. We further show that despite executing extra predication instructions, DMDP is power efficient as it saves about 6.7% on EDP.

I. Introduction

In superscalar processors, store instructions do not update the memory subsystem until they commit. Therefore, a mechanism to bridge in-flight stores and loads is necessary and critical. Without this mechanism, in-flight loads and their dependent instructions would have to wait until all prior stores commit. Most processors implement an associatively searched, age-ordered store queue to handle the store-load communication. When a load is executed, the store queue and the data cache are simultaneously accessed. If the store queue does not contain a store instruction with the same address, the data read from the cache is used. Otherwise, the youngest matching store data is selected. This search latency dramatically increases as the number of in-flight stores grows. Although processor manufacturers do not release the search latency of their mechanisms, it is unlikely to be one cycle [1].

Despite the difficulty of scaling, each processor generation incorporates a larger store queue than its predecessors in order to exploit more instruction-level parallelism (Sandy Bridge 36, Haswell 42, SkyLake 56). In order to improve scalability, several mechanisms were proposed to simplify the associative search operations [2], [3], [4], [5], [6], [7], [8], [9] as well as completely removing the store queue [10], [11], [12].

*This work is supported in part by NSF grant CCF-1533828.

NoSQ [10] is one of these mechanisms which completely eliminates the store queue. The store in **NoSQ** is executed at the commit stage and then updates the cache. Therefore, the store is never issued to the out-of-order core. The in-flight store-load communication is accomplished through a memory dependence predictor. A load which is predicted to be dependent on a prior store is renamed to the physical register of that store (memory cloaking [13]). This collapses the DEF-store-load-USE dependence chain into a DEF-USE chain. In order to verify the memory dependence prediction, the load is re-executed at the retire stage if necessary. Given prior history, if the memory dependence prediction confidence is low, **NoSQ** delays the execution of the load until the store is committed and the cache is updated. Therefore, these delayed loads need to be kept in a reservation station-like structure and woken up by committed stores.

In this paper, we present Dynamic Memory Dependence Predication (**DMDP**) to completely eliminate the unnecessary delays and the overhead of keeping delayed loads. The basic idea is to create a new MicroOp to compare the addresses of the predicted load-store pair. If the addresses match, the load uses the store data directly. Otherwise, the data read from the cache is used. As a result, the false dependence between the load execution and the store commit is removed.

Removal of this false dependence is significant because the store commit latency increases drastically when memory consistency models [14] are considered. Store-queue-free architectures eliminate the store queue which holds speculative stores before they are retired, but a store buffer still has to be provided to hold the retired stores until they update the cache. This buffer is essential for overlapping the penalty of store misses and properly implementing consistency models.

II. Motivation

It is possible to classify load-store dependencies into three categories: 1. Never Colliding (NC); 2. Always Colliding (AC); 3. Occasionally Colliding (OC). In NC, loads can always read from the cache without touching the store queue. For example, sweeping accesses through an array without changing the array values will generate many NC loads. Store-queue-free mechanisms work perfectly also in AC scenario due to the high dependence prediction accuracy. Examples of AC include register spilling, global variable accesses, stack accesses, etc. In contrast, OC is hard to predict since a correct prediction can

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00029

not be achieved by simply observing the history of the memory dependencies. Figure 1(a) shows a code example of OC cases. In each iteration, a new pointer is read from an array and the pointed content is incremented. Figure 1(b) shows two nearby iterations in which the increment instructions collide whenever the two pointers are the same.

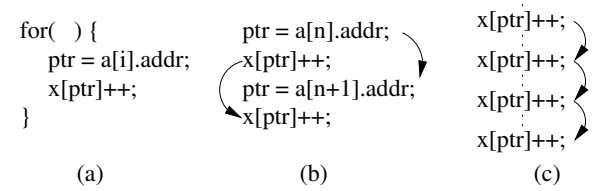

Fig. 1. OC dependence caused strict ordering.

A common store queue free mechanism such as **NoSQ** initially would always read from the cache. When the first collision happens, the dependence is added and the increment instruction will be predicted to read the value from the previous iteration instead of the cache. However, if the pointers mismatch the next time, the forwarded data is probably wrong and the memory dependence is mispredicted as well. Frequent mispredictions on a certain load would impose a strict memory ordering. That is the load can not execute until the potentially aliasing store is committed. Figure 1(c) shows that the increment instruction will not execute until the previous one commits. This strict ordering makes sure the load reads the correct value regardless of whether or not the store and load addresses match. However, this strict ordering may significantly hurt the program performance. First of all, if the store and the load addresses are different, the load and its dependent instructions suffer unnecessary latency. Even if the addresses are identical, the load has to wait until the store commits. Many unrelated events such as cache misses may delay the store committing and consequently affect the load execution time.

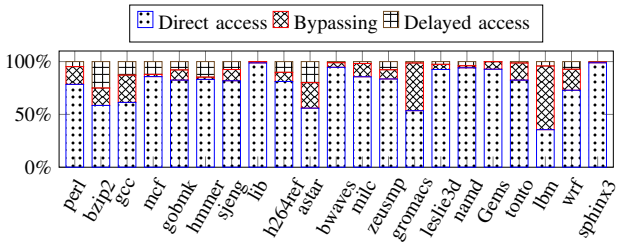

Fig. 2. Load instruction distribution

Figure 2 illustrates a breakdown of load instructions based on how they get their values in **NoSQ**. The configuration of the evaluation is described in detail in Section V. In the figure, **Direct access** means the load gets its value directly from the cache. **Bypassing** means the load gets its value through memory cloaking. **Delayed access** means the load cannot get its value from the cache until the conflicting store commits. Note that *bzip2*, *gcc*, *mcf*, *hmmer*, *h264ref* and *astar* have more than 10% **Delayed access** loads. We also compare the average execution time of **Bypassing** and **Delayed access** loads where the load execution time is defined to be the number of cycles spent between renaming of the load and the load result becoming available. In **Bypassing** cases, the execution time might be negative since the store data is available even before the load is renamed. We set the execution time to be zero for these cases. The comparison is illustrated in Figure 3.

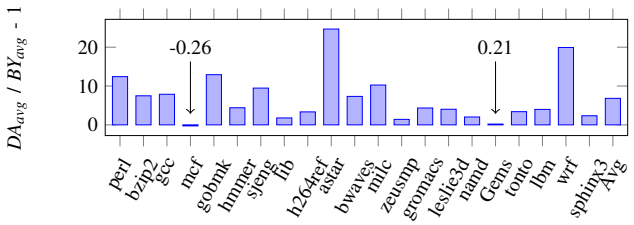

Fig. 3. Delayed loads vs. bypassing loads

In this figure, if the ratio is greater than zero, **Delayed access** loads have longer average execution time. Otherwise, **Bypassing** loads are longer. The figure shows that **Delayed access** loads take significantly more cycles to execute in most benchmarks, except *mcf*. In *mcf*, the average execution time of **Delayed access** loads is 117.6 cycles and is 159.3 for **Bypassing** loads. This is because the colliding stores are always dependent on some other cache miss loads so that memory cloaking does not effectively work in these cases. Overall, the execution time of **Delayed access** loads is about 7 times longer than the **Bypassing** loads. If these **Delayed access** loads are on the critical path, the program will be forced to execute in an in-order manner.

III. Concept

Predication is a technique to convert control dependencies into data dependencies. It is widely applied in branch elimination where the branch result is computed as a predicate which can then be used to select the correct operand [15], [16], [17]. When the store and load dependence is not consistent, **NoSQ** needs to conservatively wait until the data source becomes unambiguous or risk frequent mispredictions. Clearly, this problem is analogous to branch prediction where a difficult to predict branch is encountered. **DMDP** therefore dynamically inserts a predicate to compare the store and load addresses. The comparison result can then be used to guide the load to obtain the correct operand from either the cache or the in-flight store in a manner similar to a conditional move instruction.

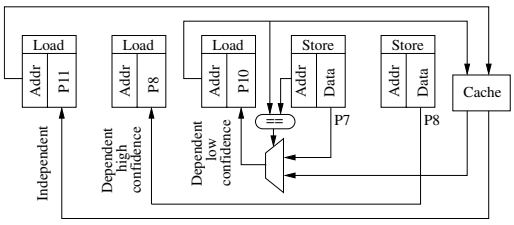

Fig. 4. Three different ways to read data for loads.

Figure 4 shows how the load gets its data in three different ways. The first load does not find any dependent store so it gets its data from the cache. The second load has a colliding store and the predictor is confident. Therefore, memory cloaking is applied and the load reuses the same physical register (P8) as

its own destination physical register. The third load also has a colliding store except its prediction is not confident. The store and the load addresses are compared to drive a multiplexer for selecting the correct data. Since the multiplexer's output is a new definition, the load is assigned a new physical register (P10) as usual.

Table I illustrates the difference between **NoSQ** and **DMDP**. The primary difference is how inconsistent store-load dependencies are handled. To be specific, the first row describes the situation when the load has never observed any dependence violation or the aliased store has committed and updated the cache when the load is renamed. **DMDP** converts all memory dependence into register data dependence so that loads do not need to check any store when it commits.

TABLE I
THE DIFFERENCE BETWEEN NOSQ AND DMDP ON DIFFERENT LOADS

	NoSQ	DMDP
No dependence or the store has committed	The data is read from the cache directly.	
Aliased store is predicted (high confidence)	The load reuses the physical register from the store. No cache read is necessary.	
Aliased store is predicted (low confidence)	The data is not read from the cache until the store is committed.	Predication is inserted so that both the store's data and the data from the cache are forwarded to the predicate instruction. The correct one is selected to bypass to the load's physical register.

The use of the store data and address physical registers are not included in the original program semantics. These registers might have been released and re-allocated to other instructions at the time the predicated instruction is created. **DMDP** delays the release time of store registers until the store is committed and updates the cache so that any in-flight store can be utilized to create the predicate. For this purpose, **DMDP** assigns an extra physical register to each memory instruction to hold the computed address. This change simplifies the address comparison and the resulting microarchitecture as well, since the memory address can be directly read from the physical register file instead of doing a base register plus offset computation. The details of the implementation are described in the next section.

Note, when a load is predicted to be dependent on a store, there are two possible types of mispredictions. Either the load is independent of any in-flight store or the load is dependent on a different in-flight store. **DMDP** can only handle the former case for low confidence loads, not the latter one. We evaluated the Spec2006 benchmark suite and found that the first type of misprediction dominates the mispredictions. Therefore, applying predication can save most of the memory dependence mispredictions.

Figure 5 illustrates the memory dependence prediction results for low confidence loads. In this figure, **IndepStore** means the load is predicted to be dependent on a store but it is actually independent of any in-flight store; **DiffStore** means the load is dependent on a different in-flight store. **Correct** means the prediction is correct. It is apparent that **IndepStore** dominates

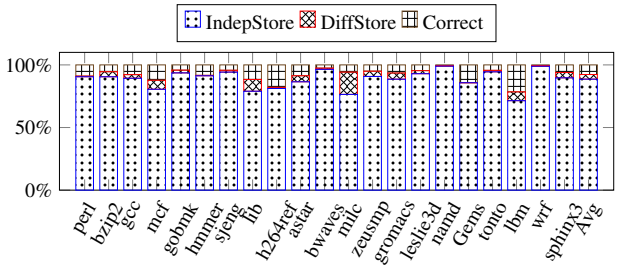

Fig. 5. Memory dependence prediction results over low confidence loads

every benchmark. In other words, if a load has a low confidence prediction, the load is most likely independent of any in-flight store. A naive solution would treat low confidence loads as independent loads and make them read directly from the cache. However, **DiffStore** and **Correct** would be mispredicted in this algorithm and the misprediction rate is considerable in some benchmarks such as *lbm*(28.6%) and *milc*(23.5%). The average misprediction rate is 11.4%. **DMDP** can correctly execute **IndepStore** and **Correct** which results in a misprediction rate of 3.7%. The delayed execution in **NoSQ** can also cover **IndepStore** and **Correct**. It can also cover some of **DiffStore** if the actual colliding store is older than the predicted store at the cost of drastically increased load latency.

IV. MICROARCHITECTURE

The microarchitecture of **DMDP** is shown in Figure 6. It tracks each store using a unique *store sequence number* (SSN) [18] and three globally observable registers, SSN_{rename}, SSN_{retire} and SSN_{commit} are used to track the store instruction states.

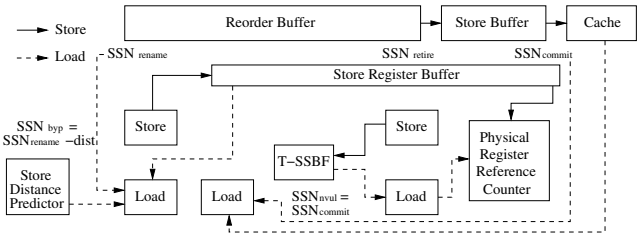

Fig. 6. DMDP Microarchitecture

When a store is renamed, the SSN_{rename} is incremented and set as the store's SSN, hence a younger store has a larger SSN. When a store retires and commits, its SSN updates SSN_{retire} and SSN_{commit} respectively. The store buffer works like a queue to hold retired stores before they commit and loads never search the store buffer. *Store Register Buffer* holds the physical register numbers of every in-flight store instruction before they are committed. When a predicated MicroOp is created, the store's data and address physical register numbers are read from this buffer. In **DMDP**, store instructions have an extended physical register lifetime since the register might be read even after the store is retired. Therefore **DMDP** includes a *Physical Register Reference Counter* to manage the register release operations. For a load, its colliding store's SSN (SSN_{byp}) is predicted when the load is renamed. The relative store distance is predicted by the *Store Distance Predictor* and SSN_{byp} equals SSN_{rename} minus the store distance. When a load is executed and reads the

data from the cache, the current SSN_{commit} is kept with the load as SSN_{nvul}. SSN_{nvul} indicates the youngest store to which the load is not vulnerable. At the retire stage, the speculative load needs to verify its value by re-execution. In order to minimize the number of re-executions, a *Tagged Store Sequence Bloom Filter* (T-SSBF) is added. In the rest of the section, we are going to elaborate on the microarchitecture details starting with the basic operations.

A. Basic Operations

In order to minimize the overhead of speculative load verification, we adopted the mechanism of Store Vulnerability Window (SVW) [18] which only re-executes the load if necessary. The second part is a path-sensitive store distance predictor [10].

a) Store Vulnerability Window: When a load is speculative, its value has to be verified before the load is retired. A simple mechanism re-executes every load at the retire stage which doubles the bandwidth requirement for the cache. SVW substantially reduces the number of re-executions by only re-executing the load if the colliding store committed after the load was executed.

When a store commits, it writes the data to the cache and updates the SSN_{commit}. Therefore, any store whose SSN is smaller than or equal to SSN_{commit} has updated the cache. When a load reads the data from the cache, it also reads SSN_{commit} and keeps it as SSN_{nvul}. During the retire time, if the colliding store's SSN is greater than the load's SSN_{nvul}, that means the colliding store updated the cache after the load read from the cache. A potential RAW hazard is possible and the load needs to re-execute. If the re-execution provides a different value, a full penalty recovery is initiated. Otherwise, if the colliding store's SSN is smaller than or equal to the load's SSN_{nvul}, that means the colliding store has committed its change to the cache and the load read the correct value. Hence, no re-execution is required.

b) Tagged Store Sequence Bloom Filter: As it is described before, when a load is retired, we need to identify its colliding store's SSN. Tagged Store Sequence Bloom Filter [19] (T-SSBF) is used to efficiently detect the store's SSN. T-SSBF is similar to an N-way set-associative cache indexed by the (hashed) memory address. The difference is each set in T-SSBF is organized like a FIFO, containing the last N store's SSNs which map to that set. When a store is retired (not committed), it writes its SSN into the T-SSBF (T-SSBF[st.addr] = st.SSN). When a load is retired, it reads the T-SSBF to find its colliding store's SSN. If multiple instances with the same address are found, the largest SSN (the youngest) is selected. On the other hand, if no matching address is found, the smallest SSN in the same set is selected.

In order to detect collisions on partial-word accesses, each memory access maintains a word address and a Byte Access Bits (BAB). BAB is used to indicate which bytes in that word are accessed. BAB is also written into the T-SSBF along with the SSN. When the word address matches and $store_{BAB}$ & $load_{BAB}$ is greater than zero, this load-store pair collides.

c) Load Re-execution: When the colliding store's SSN is larger than the load's SSN_{nvul}, a load re-execution is scheduled. Since the store buffer still holds some pending stores which have not yet updated the cache, the load re-execution is not issued until the store buffer is drained. This impact caused by the store buffer can significantly deteriorate the overall performance. Optimization in reducing the number of load re-executions is desirable and we will describe it later.

d) Memory Dependence Prediction: We need a mechanism to predict the colliding store's SSN when a load is renamed. In **DMDP**, we use *Store Distance Predictor* [20] to predict the memory dependence. This structure is indexed using the load's PC and predicts how many stores there are between the aliased store and the load, assuming this distance is constant so the same load will always collide with the store at the same distance. With the predicted store distance, we can compute the colliding store's SSN as $ld.SSN_{byp} = SSN_{rename} - ld.dist_{byp}$. At the retire stage, the store distance prediction needs to be verified. The actual store distance is computed as SSN_{retire} - T-SSBF[ld.addr]. If the prediction is wrong, the store distance predictor is updated with the actual distance.

Multiple branches between the store and the dependent load may cause the store distance to vary if the branches lead to different paths. A path-sensitive memory dependence predictor [10] is used to handle the change in control flow which uses an identical structure except indexing its table by an XOR of load's PC and branch history bits. The path-sensitive predictor and the path-insensitive predictor are read simultaneously. Prediction from the path-sensitive predictor is selected if it is available. Otherwise, the path-insensitive prediction is used. If the load misses both predictors, the load is predicted to be independent and can directly read the cache when its address is available.

e) Memory Cloaking: Once a colliding store's SSN is predicted, the physical register which produces the store data is read from the *Store Register Buffer*. The load uses this physical register as its destination register. As a result, this load does not need to access the cache and only computes its address. In this approach, the DEF-store-load-USE dependence chain now is collapsed to DEF-USE and this is called memory cloaking [13]. Memory Cloaking is a very powerful method, because the data is forwarded to the load even without knowing the address.

We implement a mechanism which splits memory operations at the decode stage into two MicroOps: an address computation and a memory access. In this design, the store queue and the load queue are completely eliminated. Figure 7 illustrates the MicroOp creation and the renaming procedure. Figure 7(a) shows the original assembly code of the store and the load. In Figure 7(b), an address generation MicroOp, ADDI, is added before each memory access, which eliminates the offset field in the memory MicroOps. Note that the logical destination register of the ADDI is $32, which is not defined in MIPS ISA ($0-$31). This register is only visible in the hardware and facilitates physical register renaming and release in the same way as a normal superscalar processor.

Figure 7(c) shows the renamed MicroOps. The store does

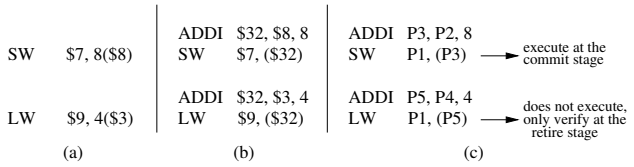

Fig. 7. Memory Cloaking

not write to the cache until it is committed, hence it is not dispatched to the out-of-order core. Both the data physical register identity (P1) and the address physical register identity (P3) are kept in the *Store Register Buffer* so that when this store is committed it can read these two values from the register file and update the cache. The store queue is removed at the cost of an additional address physical register (in a typical superscalar processor, the address does not require a dedicated physical register but instead an entry in the store queue).

Load instructions operate in a similar manner such that a dedicated address physical register is allocated. At the retire stage, both the data and the address physical registers are read to verify the memory dependence prediction (the data is required if a load re-execution verification is issued, and the address is required to read T-SSBF). The load queue is removed since the address is kept in the register file. Figure 7(c) shows a bypassing load which reuses the store data register (P1) as its own destination physical register. Hence, this load is not dispatched to the out-of-order core either. The processor only verifies its value at the retire stage.

Our mechanism is different from **NoSQ** in how memory addresses are stored. In **NoSQ**, the address offset is kept in the ROB and the address has to be calculated at the retire stage (for non-bypassing loads, these are extra computations). In **DMDP** the address is kept in the register file and directly read at the retire/commit stage. Furthermore, the address generation instruction (AGI) translates the virtual address to a physical address by setting a special MicroOp flag, making it different from a normal ADDI. When the AGI is computed, it searches the TLB to find the physical address and stores the physical address in the address register. Therefore, at the retire/commit stage, physical addresses are available for memory ordering violation detection and no extra translation is needed. The drawback of this approach is the non-bypassing loads have to wait for the address translation and an extra delay is imposed. In order to remove this side effect, we use the virtual address to read the data array and the tag array simultaneously with the address translation. In the next cycle, the translated physical address is compared with the tag array output and the correct data is selected. This approach takes advantage of a VIPT cache organization.

B. Predication insertion

A predicate is inserted at the load if the memory dependence prediction is not confident. Figure 8 shows an example of predicate creation and renaming. Figure 8(a) shows the original assembly code of the store and the load. Figure 8(b) presents the decoded MicroOps alongside with address generation MicroOps. Figure 8(c) illustrates the predicate creation before renaming. Note that, in reality the predication insertion uses physical registers. We use logical registers in Figure 8(c) to improve readability. The insertion happens at the decode stage since the dependence is predicted at the same stage.

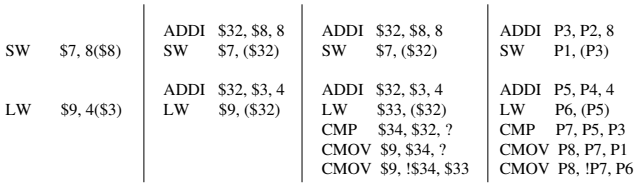

Fig. 8. Memory predication insertion

In the figure, there are three new MicroOps inserted after the load: a comparison, CMP, which produces the predicate $34 and two conditional moves, one of which would update the load destination register $9. Logical register $33 keeps the data read from the cache and $34 serves as the predicate. The store address and the store data sections are marked with a question mark in the figure. Because logical register $7 and $32 may be modified after the store, only physical registers are available during the predication insertion process. The CMP instruction compares the store address with the load address and sets the predicate $34 to one if the addresses match. If the predicate is set, the CMOV instruction forwards the store data to $9. Otherwise, the loaded data $33 is forwarded.

Figure 8(d) shows the renamed predication code. Note that the two CMOVs are mapped to the same physical register P8 since only one of them will write to the RF. Both CMOVs are dispatched to the out-of-order core. When the CMOV is woken up to execute, it first checks the predicate and only executes if the predicate is set. Otherwise, the CMOV is treated as a NOP and does not update the RF. The benefit of sharing one physical register is it reduces the data dependence and the number of the operands is two instead of three (a predicate and two operands selected by the predicate). The physical register P8 is defined twice, similar to a memory cloaking destination register. Its release algorithm is the same and we elaborate on the details next.

a) Physical Register Reference Counter: In superscalar processors, a physical register is defined once in its lifetime and is never read again after it is released. Since neither of these conditions are valid in **DMDP**, we incorporate *Physical Register Reference Counters* to guide the physical register allocation and release mechanism.

In **DMDP**, a given physical register might be defined multiple times in its lifetime. For example, a load might reuse the colliding store's data register as its own destination register in memory cloaking. A predication insertion creates two conditional moves which have the same destination register. A counter based algorithm [21] is implemented in **DMDP** to track the number of definitions through the register's lifetime. This producer counter is incremented when the register is defined and decremented when the register is virtually released. Figure 9 demonstrates a simple example. P7 is defined twice so the counter number is two at the beginning. When the second instruction is retired, it virtually releases the previous definition,

P7. Hence, the counter value is decremented to one. When the last instruction is retired, it also decrements the producer count of P7 and the counter becomes zero at that point.

Fig. 9. The producer counter

On the other hand, a physical register might be read even after it is released. For example, in Figure 8(d), P1 contains the store data and could be released before it is read by the conditional move. Another case happens in the store buffer. When a store is retired and transferred to the store buffer, its data and address physical registers might be released before the store is committed. We have to extend the lifetime of these physical registers in order to guarantee correct memory forwarding. A consumer counter is added for each physical register. When a consumer operand is renamed, the renamed physical register's counter is incremented. When the instruction which has that operand executes, the counter is decremented. The store executes when it is committed. The consumer counter was first proposed in [22] which is used to make early physical register release. But in our scenario, it is used to delay the register release.

Overall, when a physical register has a producer counter of zero and a consumer counter of zero, it is released. We use a mechanism walking through squashed instructions to recover the counters during recoveries [1].

C. Load Re-execution Filter

When a speculative load is about to retire, its loaded data needs to be verified by a re-execution. The cost of load re-execution is too high as loads can not be issued until the store buffer is drained. As a result, we have to minimize the number of such re-executions. At the same time, we have to make sure that every potential collision is checked. Based on the data source, we can classify the loads into two categories: i) the loads which read their data from the cache; ii) the loads which get their data from an in-flight store.

TABLE II
LOAD RE-EXECUTION POLICY FOR DIFFERENT LOADS.

	re-execution condition
Data from a cache	T-SSBF[ld.addr] > $ld.SSN_{nvul}$
Data from a store	T-SSBF[ld.addr] != $ld.SSN_{byp}$

Table II lists the re-execution policy for these two kinds of loads. If the data is coming from the cache, $ld.SSN_{nvul}$ indicates the SSN_{commit} when the load is executed. Therefore, if the actual colliding store's SSN is larger than $ld.SSN_{nvul}$, a re-execution verification is required. If the data is forwarded by a predicted colliding store, then the actual colliding store's SSN must match with the predicted one. Otherwise, a re-execution verification is issued.

a) Silent Stores Effect: Silent stores [23] can be detected in many of the programs, in which multiple stores write the same value into a particular memory location. This is mainly caused by the program redundancy. In **DMDP**, silent stores impose a lot of unnecessary load re-executions. Figure 10 shows a simple example, in which the two stores write the same value to the same address. The load also reads this address. This figure displays the status when the load is executed. Since store1 has committed (st1.SSN < SSN_{commit}), the load reads the data updated by store1 from the cache. When the load is about to retire, it finds itself colliding with store2 (st2.SSN > $ld.SSN_{nvul}$). Thus, a load re-execution is issued and no exception is set since the reloaded data is the same. The original design does not update the *Store Distance Predictor* unless an exception is reached. Consequently, this load will incur re-execution many times without creating any exception.

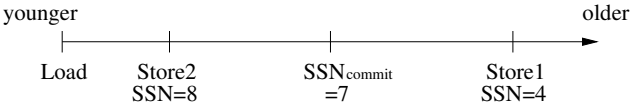

Fig. 10. Load re-execution incurred by silent store

In order to solve this problem, the memory dependence should be created even when no exception is observed. Whenever a load re-execution occurs, the *Store Distance Predictor* is updated. In Figure 10, the store distance between store2 and the load is kept in the predictor. Thus, store2 will forward the data to the load and no load re-execution is required based on the policy in Table II.

D. Partial-word Forwarding

For memory predication to work, it must handle any kind of store-load forwarding, including partial-word forwarding. We use word address plus byte access bits (BAB) to detect partial-word forwarding violation. On a 32-bit machine, a word is composed of 4 bytes so 4 bits (BAB) are used to indicate which bytes are accessed. This BAB can be expanded to 8 bits for a 64-bit machine. When a store retires, its BAB, $store_{BAB}$ is written into the T-SSBF with other information. When a load retires, not only the word address but also the BAB is compared. Figure 11 shows how a partial-word forwarding is verified.

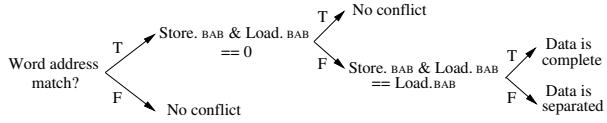

Fig. 11. The decision tree of partial-word forwarding detection

If $store_{BAB}$ & $load_{BAB}$ is equivalent to $load_{BAB}$, that means the prior store contains the whole data section required by the load. The data forwarding is correct. On the other hand, if the store only modifies part of the data section, which means the needed data is separated by multiple stores, then a load re-execution is triggered as the load is retired.

A partial-word forwarding may cause the forwarded data to be shifted. For example, a store writes a whole word into the cache and the dependent load only reads the upper half word. In which case, the store data is right shifted 16 bits before the forwarding. In MIPS, the shift amount is decided by the

least significant two bits of the memory address. The value of these two bits times eight is the shift amount. The store shifts left and the load shifts right. **DMDP** has a dedicated MicroOp, CMP, to compare the store and load address. When a CMP is executed, **DMDP** also puts the shift amount and direction information into the predicate (the predicate is a word-wide register, only one bit is used to guard the predicated instruction, other bits can be used). Therefore, the CMOV knows how to shift the operand before it is forwarded.

Other than address alignment, the forwarded data may be masked and sign/zero extended based on the load type. **DMDP** keeps the load type in the CMOV when it is created. So when CMOV is executed, the operand is trimmed properly. Moreover, partial-word loads, such as half word load or byte load, are prohibited from doing memory cloaking due to the alignment or sign/zero extension. Thus, these loads are forced to use predication for store-load communications. **NoSQ** inserts special "shift & mask instruction" for partial-word communication. However, the store and the load addresses are unknown at the rename stage, thus the shift amount has to be predicted. Our partial-word mechanism can be easily adopted to other ISAs, since the store address, load address, store type and load type can be integrated into the predicate.

E. Confidence Predictor

When a load is predicted to be dependent on a store, **DMDP** consults the confidence predictor to decide whether memory cloaking or memory predication should be used. The confidence predictor is embedded in the *Store Distance Predictor* and updated at the retire stage. The loads which cause load re-executions or are predicted to be dependent on a store can update the confidence predictor. When the prediction is correct, the corresponding confidence counter is increased. Otherwise, the confidence counter is decreased. A common confidence predictor modifies the counter with a balanced strategy. In other words, it increases and decreases the counter by the same amount, for example, one. In **DMDP**, pushing instructions to predication only causes a trivial data dependence increase, but a dependence misprediction results in a full recovery penalty. Because the cost is biased, the confidence counter update should be biased as well. **DMDP** right shifts the counter by one bit (divided by two) whenever a misprediction is detected and only increments the counter by one in other cases. By applying this strategy, fewer mispredictions are experienced at the cost of more predications.

F. Memory Consistency

In a multi-core system, the cache lines might be invalidated by other cores. Therefore, even if the load satisfies the local memory dependence check, it may still need to be re-executed due to the stores from other cores [19]. This makes two changes to the design: i) When a cache line is invalidated by another core, all the words in that cache line should update the T-SSBF as the invalidation message usually only contains the cache line address without word offset; ii) The word invalidated by another core should write SSN_{commit}+1 as its SSN in the T-SSBF, i.e., all the in-flight loads which were executed before the invalidation should re-execute if their addresses match.

When we were designing our store buffer mechanism, we considered different memory consistency models, such as total store order (TSO) and relaxed memory order (RMO). In TSO, the stores in the store buffer are committed following the program order. When the store buffer is full, stores are not allowed to retire from the ROB. RMO mitigates the pressure of the store buffer and permits the stores to commit in any order. In either model, the load in **NoSQ** has to wait for the colliding store and its preceding stores to commit if the memory dependence prediction is not confident. **DMDP** is not constrained by the committing of the stores.

V. Simulation Methodology

We simulated **DMDP** by using MIPS-I ISA without delayed branching. This ISA is very similar to PISA ISA, used by SimpleScalar [24]. GCC 4.9.2 tailored to this ISA is used to compile the benchmarks and generate binary code with the highest optimization ("-O3") set. We choose Spec 2006 as our benchmark suite. All simulation models were designed with Architectural Description Language (ADL) [25]. The ADL compiler can automatically generate the assembler, the disassembler and a cycle-accurate simulator which respects timing at the register transfer level from the description of the microarchitecture and its ISA specified in ADL language.

In order to efficiently simulate our mechanisms, we incorporated Simpoint 3.2 [26], [27] to minimize the simulation time. For each benchmark, a set of checkpoint images were generated and each checkpoint image contained the complete memory data segment, the register file and the program counter (PC). Other architecture related structures were not included, such as cache, branch predictor, memory dependence predictor, etc. Hence, the simulation of each interval had a cold start. In order to compensate for this effect, we selected a large size, 100 million retired instructions, to simulate each interval. Since each interval simulation is independent of others, we simulated all of the intervals simultaneously to further shorten the simulation time. Currently, the file descriptors are not kept in the checkpoint. Therefore, we could not simulate an interval if it had file operations and the file descriptor was created before the checkpoint. When this happened, we replaced that checkpoint interval with the dominant checkpoint in that benchmark. In *h264ref*, we substituted one checkpoint (0.92% weight) for the dominant checkpoint (18.14% weight) and in *hmmer*, we substituted two checkpoints (0.22%, 0.43% weight) for the dominant checkpoint (98.9% weight). As the replaced intervals have very limited weights (<1.0 %), we expect the impact of this substitution to be minimal. We also modified McPAT 1.4 [28] to evaluate the dynamic energy consumption. T-SSBF and the memory dependence predictor which replace the load queue and the store queue were both modeled. DRAMSim2 [29] was also embedded to evaluate the memory subsystem behavior. The simulated benchmarks are: **Integer**: *perl*, *bzip*, *gcc*, *mcf*, *gobmk*, *hmmer*, *sjeng*, *lib*,

h264ref, *astar*; **Float**: *bwaves*, *milc*, *zeusmp*, *gromacs*, *leslie3d*, *namd*, *Gems*, *tonto*, *lbm*, *wrf*, *sphinx3*.

We simulated these benchmarks with the "ref" input. The remaining missing benchmarks were not included due to our linker's inability to link them. The processor configuration of the baseline architecture is listed in Table III which we used as a reference for comparing with other models. The baseline architecture has a store queue and a load queue with unlimited entries. A store buffer is also implemented following TSO model. Store coalescing was implemented to alleviate the write port pressure. Since TSO is considered, only consecutive stores are coalesced. We used **Store Set** [30] algorithm to predict the memory dependences in the baseline model.

TABLE III
BASELINE PROCESSOR CONFIGURATION

ROB / RS / PRF	256 / 64 / 320
Fetch / Decode / Issue	8 / 8 / 8
Store Queue	unlimited entries, 4 cycles search latency
Store Buffer	16 entries, store coalesce
Cache	32KB 8-way set associative iL1; 32KB 8-way set associative dL1, 4 cycles hit latency, 2 read ports, 1 write port; 512KB 8-way set associative L2, 10 cycles hit latency;
Memory	16GB DDR3L-1600, 2 channels, 2 ranks, 8 banks, open page, up to 64 pending requests [31]
Recovery Penalty	minimum 15 cycles
Int ALU / Int Mul	1 cycle / 3 cycles
Int Div, FP ALU	7 cycles
Branch Predictor	8 kB TAGE [32]
Tech node	22nm
Clock frequency	3.2GHz

We simulated the following models which differ from the baseline in their store-load communication mechanism.

1. NoSQ: This architecture [10] has no store and load queue which are replaced with a 4-way set associative, 128 entry T-SSBF. Each entry contains a 20 bit SSN, a 4 bit BAB and a 25 bit tag. The total size of T-SSBF is 6.125 Kbits. The *Store Distance Predictor* uses two 4-way associative, 1K entry tables. One of the tables is for path-insensitive predictions and the other is for path-sensitive predictions. The path-insensitive table is indexed by the load PC. The path-sensitive table is indexed by the XOR of the load PC and an 8 bit branch history. Each table entry contains a 7 bit confidence counter, a 22 bit tag and a 6 bit distance part. The predictor's total size is 8.75KB. The confidence counter is set to 64 by default. If the value is greater than 63, memory cloaking is used, or the load has to wait for the colliding store to commit. The number of delayed loads which can be kept in the core is unlimited. Silent-store-aware predictor update policy is used to match **DMDP**.

2. DMDP: **DMDP** has the same T-SSBF and dependence predictor as **NoSQ**. The only difference is that **NoSQ** decreases the confidence counter by one if a misprediction is detected. But **DMDP** divides the counter by two in the same case. Predicate is added when a low confidence prediction is made.

3. Perfect: This model has a perfect memory dependence predictor so that every load gets its data from either a colliding store or the cache. No delayed executions or mispredictions are experienced.

VI. EXPERIMENTAL RESULTS

Figure 12 illustrates the IPC performance of **NoSQ**, **DMDP** and **Perfect** models normalized to the baseline model. The geometric means of the Integer benchmarks are 0.975, 1.045 and 1.068, for **NoSQ**, **DMDP** and **Perfect** respectively, and the corresponding floating point benchmark performances are 1.008, 1.053 and 1.066. Clearly, **DMDP** is much closer to **Perfect** in terms of IPC performance.

a) NoSQ VS. Baseline: **NoSQ** can forward store data to the load by memory cloaking. That is the reason why it outperforms the baseline in some of the benchmarks. On the other hand, **NoSQ** has to stall the retire stage when a load re-execution is issued when the store buffer is not drained. Moreover, **NoSQ** may experience more memory dependence mispredictions due to its aggressive prediction strategy. Figure 12 shows **NoSQ** works more than 20% worse in *hmmer*. Analyzing the result, we found that **NoSQ** has a significant amount of memory dependence mispredictions (3.06 Mispredictions Per 1k Instructions). Further analysis showed that the silent store predictor update policy, which was described before, had a substantial impact in this benchmark. This policy would update the **Store Distance Predictor** whenever a load re-execution was triggered. If we change back to the original mechanism which only updates the predictor if the re-execution leads to an exception (the reloaded data is different), **NoSQ** has fewer mispredictions and higher performance in *hmmer*. However, it reduces the performance of other benchmarks.

Silent-store-aware predictor update policy is a double-edged sword. It reduces the number of load re-executions immensely but also might cause the increase of mispredictions. **DMDP** could compensate this shortcoming by using predication. From our evaluation, **DMDP** had much fewer mispredictions (1.03MPKI VS. 3.06MPKI) and it only had 1% lower performance than the baseline in *hmmer*.

b) DMDP VS. Baseline: **DMDP** surpasses the baseline in every benchmark except *hmmer* which is caused by the nontrivial memory dependence mispredictions as mentioned before. In the baseline model, loads have to read the data from the cache, store queue or store buffer. All of these structures have a constant access latency (4 cycles in our simulation). **DMDP** can use memory cloaking to bridge stores with loads if the predictions are confident. Even a low confidence load can obtain its data quicker if the data is from an in-flight store.

Table IV lists the average execution time of the loads in baseline and **DMDP**. The execution time is the number of cycles spent between renaming and the load result becoming available. **DMDP** has a shorter execution time in every single benchmark and on average, saves more than 20% of the execution time for loads. Figure 12 shows that **DMDP** has the most improvements in *wrf* and *bzip2* and in these two benchmarks, **DMDP** saves about half of the execution time of the loads.

c) DMDP VS. NoSQ: **DMDP** outperforms **NoSQ** in every single benchmark. The geometric mean of the speed-up is 7.17% (Int) and 4.48% (FP). If a low confidence load is renamed, **NoSQ** has to wait until the predicted colliding store

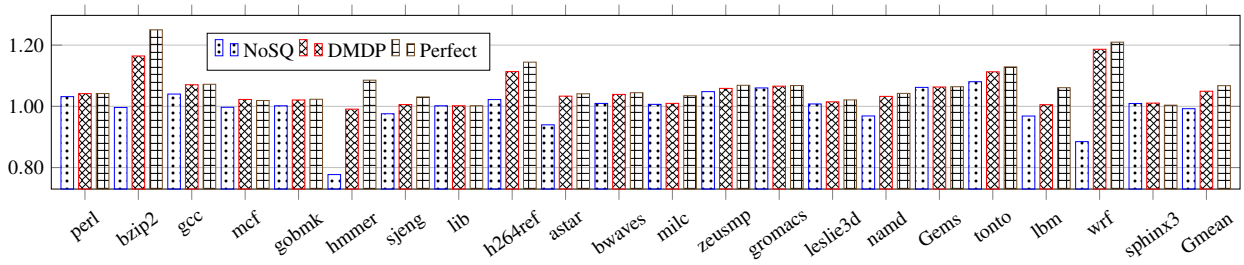

Fig. 12. Spec 2006 Speedup over the baseline

TABLE IV
AVERAGE EXECUTION TIME OF ALL LOADS

	baseline (Cycles)	DMDP (Cycles)		baseline (Cycles)	DMDP (Cycles)
perl	15.86	12.45	bzip2	36.67	19.48
gcc	44.98	35.04	mcf	112.44	104.00
gobmk	13.51	11.52	hmmer	11.20	7.47
sjeng	12.60	10.62	lib	125.23	124.73
h264ref	22.68	17.32	astar	21.18	13.77
bwaves	42.56	36.76	milc	73.40	61.18
zeusmp	26.97	21.21	gromacs	32.13	11.41
leslie3d	36.55	32.91	namd	20.22	18.94
Gems	14.78	11.62	tonto	20.31	12.89
lbm	72.17	31.15	wrf	18.17	9.19
sphinx3	51.95	50.47	Average	39.31	31.15

commits. **DMDP** can steer the load to find its correct data by predication, disregarding the store commit states. Table V lists the average execution time of the low confidence loads in **NoSQ** and **DMDP**. **DMDP** saves up to 79.25% execution time and the average is 54.48%. *Lib* is the only benchmark in which **DMDP** has a longer execution time. This data is not representative due to the fact *lib* has so few low confidence loads.

TABLE V
AVERAGE EXECUTION TIME OF LOW CONFIDENCE LOADS

	NoSQ (Cycles)	DMDP (Cycles)		NoSQ (Cycles)	DMDP (Cycles)
perl	63.79	14.54	bzip2	65.29	22.34
gcc	60.69	18.14	mcf	111.40	52.39
gobmk	23.73	11.58	hmmer	37.19	8.91
sjeng	24.54	12.36	lib	9.11	13.38
h264ref	41.29	19.17	astar	85.47	25.16
bwaves	103.90	36.17	milc	141.28	85.72
zeusmp	118.72	24.66	gromacs	68.10	65.94
leslie3d	53.76	20.59	namd	22.53	16.33
Gems	11.11	9.98	tonto	35.85	10.75
lbm	129.69	100.76	wrf	61.59	12.78
sphinx3	49.93	18.68	Average	62.81	28.59

In Figure 12, **DMDP** surpasses **NoSQ** in *wrf* by 34.1%, which is the highest improvement. **NoSQ** works worse than not only **DMDP** but also the baseline. From our evaluation, the average execution time of all loads is 18.17 cycles, 13.85 cycles and 9.19 cycles for the baseline, **NoSQ** and **DMDP** respectively. The loads in **NoSQ** has a shorter execution time, but the overall performance is still worse than the baseline. One possible reason is the delayed loads in **NoSQ** are on the critical path of the program and slow down the rest of the instructions. Therefore, even the average load execution time is saved, the whole program runs slower. We evaluated the average execution time of all instructions and it is 19.53 cycles, 21.47 cycles and 12.74 cycles for the baseline, **NoSQ** and **DMDP** respectively.

d) DMDP VS. Perfect: On geometric mean, **DMDP** loses 2.19% (Int) and 1.25% (FP) IPC compared to **Perfect**. There are three reasons why **Perfect** outperforms **DMDP**: i) **DMDP** has a large amount of memory dependence mispredictions in some benchmarks which **Perfect** does not have; ii) When a load verification triggers a load re-execution, **DMDP** has to wait for the store buffer to be drained whereas **Perfect** never re-executes any loads; iii) For low confidence loads, **DMDP** still needs to wait for the addresses being computed even if the addresses match. **Perfect** only has high confidence loads which receive their data by memory cloaking.

TABLE VI
MEMORY DEPENDENCE MISPREDICTION RATE

	NoSQ (MPKI)	DMDP (MPKI)		NoSQ (MPKI)	DMDP (MPKI)
perl	0.144	0.141	bzip2	0.784	1.409
gcc	0.301	0.250	mcf	0.232	0.147
gobmk	0.305	0.198	hmmer	3.061	1.029
sjeng	0.420	0.357	lib	0.000	0.000
h264ref	0.226	0.118	astar	0.110	0.086
bwaves	0.039	0.002	milc	0.363	0.363
zeusmp	0.021	0.024	gromacs	0.070	0.057
leslie3d	0.063	0.056	namd	0.004	0.003
Gems	0.007	0.001	tonto	0.160	0.130
lbm	0.101	0.089	wrf	0.057	0.066
sphinx3	0.020	0.046	Average	0.309	0.218

Table VI lists the memory dependence misprediction rate in **NoSQ** and **DMDP** measured in terms of Mispredictions Per 1k Instructions (MPKI). On one hand, **DMDP** would have more low confidence loads since the confidence predictor has a biased update policy (Section IV-E). As a result, **DMDP** should have fewer mispredictions. On the other hand, **NoSQ** would delay the execution of low confidence loads. As a result, some mispredictions which can not be covered by **DMDP** can be covered by **NoSQ**. For example, if the load depends on a different in-flight store and that store is older than the predicted one, **NoSQ** can read the correct data through the cache. In the same case, **DMDP** might read the cache earlier but potentially the wrong value.

Note that **DMDP** has more dependence mispredictions than **NoSQ** in *bzip2*. A snapshot of the code which causes the most mispredictions is demonstrated in Figure 13. In this code, **LHU** sequentially reads an array which contains a pointer ($9). After a series of computation, the pointed value is incremented by one. If the array has two identical values, they point to the same memory location and the increment operations collide

with each other. During the execution, the distance between the colliding store and the load keeps changing. Therefore, a lot of mispredictions are observed in *bzip2*.

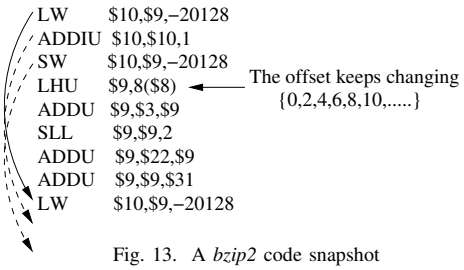

Fig. 13. A *bzip2* code snapshot

Let us assume the colliding store is randomly distributed. Thus, when a misprediction happens, half of the time the actual colliding store is older than the predicted colliding store and the other half is younger. **NoSQ** can cover the former cases and **DMDP** mispredicts both cases. Table VI shows **NoSQ** has about half of the mispredictions of **DMDP**.

TABLE VII
LOAD RE-EXECUTION RELATED STALLS PER 1K COMMITTED INSTRUCTIONS

	NoSQ (cycles)	DMDP (cycles)		NoSQ (cycles)	DMDP (cycles)
perl	4.532	11.258	bzip2	3.999	30.235
gcc	5.031	12.182	mcf	4.205	9.173
gobmk	0.679	0.951	hmmer	32.675	101.631
sjeng	1.118	1.485	lib	0.001	0.001
h264ref	1.002	15.038	astar	0.881	54.548
bwaves	0.720	6.231	milc	16.997	32.113
zeusmp	5.629	18.389	gromacs	0.262	0.526
leslie3d	2.430	3.925	namd	0.040	0.338
Gems	0.004	0.028	tonto	0.419	0.882
lbm	154.956	155.260	wrf	0.575	5.027
sphinx3	0.161	0.200			

When a load is re-executed, the retire stage is stalled until the store buffer drains. Table VII lists the number of stalled cycles per 1000 committed instructions in **NoSQ** and **DMDP**. **DMDP** has more stalled cycles in every benchmark due to its early load execution. Since **NoSQ** delays the low confidence load execution, it has a much narrower vulnerable window and fewer load re-executions are issued.

DMDP loses the most performance in these three benchmarks: *hmmer* (-8.71%), *bzip2* (-6.83%) and *lbm* (-5.23%) when compared with **Perfect**. The first two benchmarks have the most memory dependence mispredictions (Table VI) and *lbm* has the most re-execution related stalls (Table VII). These are the two major obstructions impeding **DMDP** to reach a higher performance.

e) Store Buffer Size: The loads in **DMDP** and **NoSQ** do not associatively search the store buffer which simplifies the design significantly. As a result, we can build a much larger store buffer and hide more cache misses imposed by stores. The store buffer size has a substantial impact on performance, especially for a multiprocessor [33], [14]. We believe mechanisms, such as **DMDP**, can be easily adopted to a multiprocessor design and boost the performance of multi-threaded workloads.

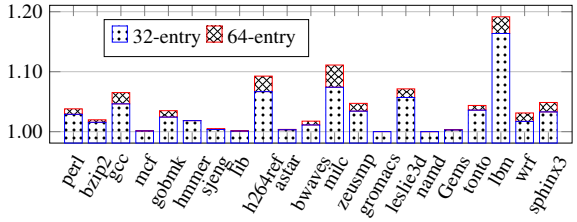

Fig. 14. 32,64-entry SB VS. 16-entry SB

Figure 14 depicts the performance of **DMDP** with a 32-entry store buffer and a 64-entry store buffer normalized to another one with a 16-entry store buffer. Using a geometric mean, the 32-entry model outperforms the 16-entry model by 2.07% (Int) and 3.81% (FP), the 64-entry model outperforms the 16-entry model by 2.77% (Int) and 5.01% (FP). Through all the benchmarks, *lbm* has the highest performance improvement with a larger store buffer. Moreover, we estimated the number of stalled cycles incurred by a full store buffer. They are 503.1 cycles, 220.5 cycles and 75.0 cycles per 1000 committed instructions for a 16-entry, 32-entry and 64-entry store buffer accordingly. Apparently, a larger store buffer can immensely help with single-thread performance. We expect the ability to implement a larger store buffer to be more fruitful in multi-threaded workloads.

f) Register File Pressure: The physical registers occupied by the store instructions are not released until these store instructions are committed from the store buffer. This change adds more pressure on the physical register file. On the other hand, memory cloaking shares the physical registers among multiple load instructions, which reduces the pressure on the physical register file. Our quantitative evaluation shows that the performance improvement of **DMDP** over the baseline is reduced to 4.24% from 4.94% when the physical register file size is cut in half (320→160).

g) Alternative Configurations: We also simulated **DMDP** with a 4-issue width. The IPC improvement over **NoSQ** shrunk to 4.56% (Int) and 2.41% (FP). This is because with a smaller issue width, the vulnerable window gets narrower and the in-flight store-load communication is reduced. The number of the low confidence loads drops as well (23.4% is removed). As a result, it is less likely for **DMDP** to save delayed load executions.

A 512-entry ROB is also simulated, which yields more IPC improvement (7.56% Int, 6.35%). A larger ROB would help **DMDP** to bridge long distance store-load communications. This long distance store-load dependence is more difficult to predict and **DMDP** has slightly fewer mispredictions.

Consistency model RMO is simulated as well. Stores are allowed to commit in an out-of-order manner. SSN_{commit} is set to the one preceding the oldest store in store buffer. When a store is committed, its corresponding entry in *Store Register Buffer* is invalidated and forwarding is prohibited. The results shows **DMDP** surpasses **NoSQ** by 7.67% (Int) and 4.08% (FP).

h) Power Efficiency: The original motivation to design **DMDP** was to eliminate unnecessary load delays. So we

did not consider the power efficiency when we designed the microarchitecture. We wanted to make sure that the extra predicated instructions do not negatively affect the power efficiency. Figure 15 illustrates the EDP (Energy Delay Product) of **DMDP** normalized to **NoSQ**. Additionally, **DMDP** saves 4.83% of EDP on average from the baseline model.

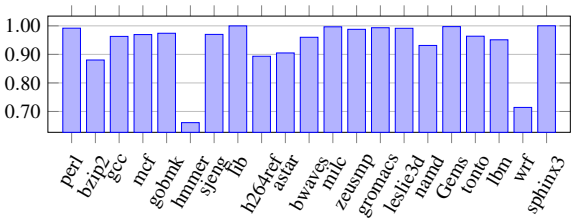

Fig. 15. The EDP of **DMDP**, normalized to **NoSQ**

As it can be seen, **DMDP** reduces the total execution time in every benchmark (Figure 12) and slightly consumes more energy due to the extra predicated instructions. Overall, **DMDP** is more power efficient than **NoSQ** and saves 8.5% (Int) and 5.1% (FP) EDP on average. If we refer the EDP result to the memory dependence misprediction rate (Table VI), we find more mispredictions result in more energy consumption. **DMDP** has more mispredictions in *bzip2* and that costs **DMDP** to consume about 3% more energy. On the other hand, **DMDP** saves around 2 MPKI in *hmmer* which leads to a 15% energy saving.

VII. Related Work

Instruction predication was initially used to convert control dependencies to data dependencies [17], [15], so that vectorization could be applied even if a loop contained conditional branches. With the mechanism, qualified branches are converted to predicate computation instructions statically during the compilation. The control dependent instructions are guarded by these predicates to eliminate conditional branches. No branches means no mispredictions and no recovery penalties at all, especially if the eliminated branches are hard to predict. Dynamic predication is also proposed to insert predicates during the run time [16] for processors which do not have a predicated Instruction Set Architecture (ISA).

Store-queue-free architectures were proposed when memory dependence prediction accuracy became acceptable. Sha et al. designed **NoSQ** [10] to completely remove the store queue by using memory cloaking to communicate the in-flight stores and loads. When the dependence is inconsistent, **NoSQ** delays the load execution, therefore, **NoSQ** still needs a structure to hold all the delayed loads until they are ready to execute. In contrast, **DMDP** inserts predication which converts load-store dependencies into simple data dependencies, therefore it does not need this extra storage at the expense of executing additional operations.

Subramaniam et al. proposed Fire-and-Forget (FnF) [11] to eliminate the store queue in a different way. Instead of predicting the load, FnF predicts the store so that when a store is decoded, its consumer load is predicted and the store forwards its data to that load. We selected **NoSQ** as our reference design since some memory dependencies are path sensitive. When FnF is predicting a store, the branches between the store and the dependent load are not considered.

Several related works attempt to reduce the size of store queue or simplify it. The mechanism proposed by Sha et al. predicted the index of the colliding store in the store queue [4]. This approach eliminates the need to search the store queue when a load is executed. Their mechanism is similar to **DMDP** in that both techniques require address comparison. The load retrieves its data from either the store queue or the data cache. The difference is that the address comparison is not triggered until the store is executed in their mechanism, whereas **DMDP** only waits for the store address computation.

Stone et al. separated the function of the store queue into three different structures [3]: store forwarding cache (SFC) for memory forwarding; memory disambiguation table (MDT) for memory ordering violation detection; and a store FIFO for in-order store retirement. CAM structures are completely replaced by RAM structures since no associative search is needed. Garg et al. had a different view of the store queue and proposed SMDE (Slackened Memory Dependence Enforcement) [7]. Their mechanism uses an L0 cache to bridge the in-flight loads and stores similar to a store queue. This simple design does not need any associative search so the size scales well. Sethumadhavan et al. proposed *Late-Binding* [34] to allocate LSQ at the issue stage instead of the dispatch stage which reduces the demand for LSQ entries. Since the LSQ allocation is completely out-of-order, an age tag is explicitly integrated into each LSQ entry. The current commercial processors are still using store queue to disambiguate memory ordering. Kim et al. designed a mechanism, store dependence prediction (SDP) [35], to indicate which store is not likely to collide with any younger load so that the load does not need to wait for this store address computation.

Perais et al. expanded the store-load bypassing to load-load bypassing with a new register sharing mechanism [1]. They introduced an Instruction Distance Predictor to predict the instruction which produces the load result. A TAGE-like [36] predictor was designed and could also be tuned as a Store Distance Predictor and adopted to **DMDP**.

VIII. Conclusion

To the best of our knowledge, predication has never been employed to convert data dependencies through memory to register dependencies. **DMDP** is the first mechanism which does so without employing special buffers to hold memory instructions. **DMDP** converts a load to i) direct access to cache; ii) reuse of the colliding store data; iii) predication between cache and the colliding store data. Therefore, the only hardware change is predication insertion at the rename stage and a consumer counter for expanding the lifetime of store's physical registers. Our evaluation of the mechanism shows that **DMDP** surpasses **NoSQ** in all benchmarks (7.17% Int, 4.48% FP). Meanwhile, **DMDP** works more power efficiently as well and saves EDP (8.5% Int, 5.1% FP).

REFERENCES

[1] A. Perais and A. Seznec, "Cost effective physical register sharing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 694–706.

[2] E. F. Torres, P. Ibanez, V. Vinals, and J. M. Llaberia, "Store buffer design in first-level multibanked data caches," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 469–480. [Online]. Available: https://doi.org/10.1109/ISCA.2005.47

[3] S. S. Stone, K. M. Woley, and M. I. Frank, "Address-indexed memory disambiguation and store-to-load forwarding," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 38. Washington, DC, USA: IEEE Computer Society, 2005, pp. 171–182. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2005.10

[4] T. Sha, M. M. K. Martin, and A. Roth, "Scalable store-load forwarding via store queue index prediction," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 38. Washington, DC, USA: IEEE Computer Society, 2005, pp. 159–170.

[5] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, "Scalable hardware memory disambiguation for high ilp processors," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 399–. [Online]. Available: http://dl.acm.org/citation.cfm?id=956417.956553

[6] I. Park, C. L. Ooi, and T. N. Vijaykumar, "Reducing design complexity of the load/store queue," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 411–.

[7] A. Garg, M. W. Rashid, and M. Huang, "Slackened memory dependence enforcement: Combining opportunistic forwarding with decoupled verification," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ser. ISCA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 142–154.

[8] A. Gandhi, H. Akkary, R. Rajwar, S. T. Srinivasan, and K. Lai, "Scalable load and store processing in latency tolerant processors," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 446–457.

[9] L. Baugh and C. Zilles, "Decomposing the load-store queue by function for power reduction and scalability," *IBM J. Res. Dev.*, vol. 50, no. 2/3, pp. 287–297, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1147/rd.502.0287

[10] T. Sha, M. M. K. Martin, and A. Roth, "Nosq: Store-load communication without a store queue," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 285–296. [Online]. Available: https://doi.org/10.1109/MICRO.2006.39

[11] S. Subramaniam and G. H. Loh, "Fire-and-forget: Load/store scheduling with no store queue at all," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 273–284.

[12] A. Hilton and A. Roth, "Decoupled store completion/silent deterministic replay: Enabling scalable data memory for cpr/cfp processors," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 245–254. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555786

[13] A. Moshovos and G. S. Sohi, "Speculative memory cloaking and bypassing," *International Journal of Parallel Programming*, vol. 27, no. 6, pp. 427–456, 1999.

[14] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 266–277.

[15] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, ser. MICRO 25. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 45–54.

[16] A. Klauser, T. Austin, D. Grunwald, and B. Calder, "Dynamic hammock predication for non-predicated instruction set architectures," in *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*, Oct 1998, pp. 278–285.

[17] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '83. New York, NY, USA: ACM, 1983, pp. 177–189. [Online]. Available: http://doi.acm.org/10.1145/567067.567085

[18] A. Roth, "Store vulnerability window (svw): Re-execution filtering for enhanced load optimization," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 458–468.

[19] ——, "Store vulnerability window (svw): A filter and potential replacement for load re-execution," *Journal of Instruction Level Parallelism*, vol. 8, no. 1, pp. 1–22, Sep 2006.

[20] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ser. ISCA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 42–53. [Online]. Available: http://dx.doi.org/10.1145/300979.300983

[21] A. Roth, "Physical register reference counting," *IEEE Computer Architecture Letters*, vol. 7, no. 1, pp. 9–12, Jan 2008.

[22] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose, "Increasing processor performance through early register release," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.*, Oct 2004, pp. 480–487.

[23] K. M. Lepak, G. B. Bell, and M. H. Lipasti, "Silent stores and store value locality," *IEEE Transactions on Computers*, vol. 50, no. 11, pp. 1174–1190, Nov 2001.

[24] D. Burger, T. M. Austin, and S. Bennett, *Evaluating future microprocessors: The simplescalar tool set*. University of Wisconsin-Madison, Computer Sciences Department, 1996.

[25] S. Önder and R. Gupta, "Automatic generation of microarchitecture simulators," in *IEEE International Conference on Computer Languages*, Chicago, May 1998, pp. 80–89.

[26] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 318–319, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/885651.781076

[27] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14.

[28] 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 *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 469–480. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669172

[29] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[30] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ser. ISCA '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 142–153.

[31] Micron, https://www.micron.com/resource-details/e570d65b-2664-4037-a141-620a6f2e58e7.

[32] T. INRIA, "Branch prediction research," https://team.inria.fr/alf/members/andre-seznec/branch-prediction-research/, the 1st Championship Branch Prediction.

[33] Y. Chou, L. Spracklen, and S. G. Abraham, "Store memory-level parallelism optimizations for commercial applications," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 38. Washington, DC, USA: IEEE Computer Society, 2005, pp. 183–196.

[34] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler, "Late-binding: Enabling unordered load-store queues," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 347–357.

[35] H.-S. Kim, R. S. Chappell, and C. Y. Soo, "Method and apparatus for store dependence prediction," Apr. 11 2017, uS Patent 9,619,750.

[36] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *Journal of Instruction Level Parallelism*, vol. 8, pp. 1–23, 2006.