Perceptron Prefetch Filtering with Static Features

Madison Bradley and Sarah Thomson
Department of Computing Science
University of Alberta
Edmonton, AB, Canada
{mlbradle, smt}@ualberta.ca

***Abstract*— Prefetcher design is complex - many state-of-the-art prefetchers such as signature path prefetching (SPP) [1] aim to predict complex memory access patterns. Many current prefetchers focus on balancing the opposed metrics of coverage and accuracy through methods such as throttling prefetch depth. However, this balancing act is a very delicate process. Perceptron prefetch filters (PPF) [2] aim to remove this balancing act between accuracy and coverage by allowing an underlying prefetcher to maximize coverage, while a filtering mechanism prunes out prefetches predicted to not be useful. This makes accuracy the responsibility of the prefetcher, separating the two metrics. PPF uses information that is readily available via the hardware at the prefetching stage - however, many program semantics that may be useful for prefetching are lost at this level. We implement a mechanism for introducing static analysis information to the perceptron prefetch mechanism. This is done by inserting instructions that store static analysis information to a register before a memory access instruction. This information is then retrieved during a prefetch request, and used as a feature value for PPF - essentially allowing static analysis information to be considered in PPF.**

I. Introduction

Processors require data stored in memory to execute tasks. While a processor is waiting for data to be retrieved from main memory, valuable computation time is lost. Prefetching schemes aim to hide this latency for future memory operations, by predicting data likely to be accessed in the future and preemptively fetching it into the cache.

Prefetcher design is complex; memory access patterns from one instruction may range from easy to predict to very difficult. This motivates specialized prefetching mechanisms that account for this complexity. The performance of a prefetcher is typically measured using **coverage** and **accuracy**, which are two metrics that are typically at odds with each other.

Existing prefetch schemes such as Signature Path Prefetching (SPP) [1] have complex prefetching mechanisms that aim to predict prefetching patterns with high coverage, while throttling prefetch aggressiveness to avoid wasting bandwidth and polluting the cache. However, these prefetching schemes try to find a sweet-spot between accuracy and coverage, by reducing prefetch aggressiveness (and subsequently coverage) in an attempt to increase accuracy.

An alternate approach is to have a highly aggressive prefetcher with a predictive filter, where the filter takes on the responsibility of eliminating inaccurate prefetches. This allows the prefetcher to focus on coverage, delegating accuracy to a separate predictor. One example of these schemes can be found in Perceptron Prefetch Filtering (PPF) [2], which uses a perceptron to automatically learn weights for a predictive filtering mechanism based on prefetching feedback.

PPF uses information that is readily available via the hardware at the prefetching stage. However, many program semantics that may be useful for prefetching are lost at this level, such as loop information. We posit that propagating static analysis information to PPF will allow for improved filtering, and thus greater prefetch accuracy.

Static analysis features can be propagated as hints to the hardware via instructions inserted at compile-time. We investigate the potential of a target-specific LLVM machine code pass as shown in Figure 1, where static analysis information is placed in a designated hardware register before each memory access instruction.

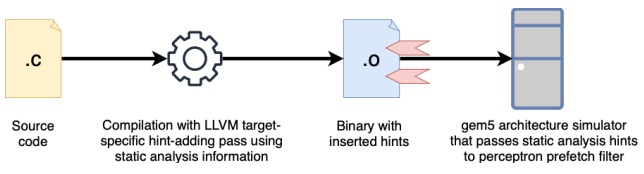

Figure 1: The proposed process for the modeling of passing static analysis hints for prefetching to the hardware.

II. Background

A. Signature Path Prefetcher

SPP [1] is a state-of-the-art prefetching scheme that stores memory access patterns and performs lookahead prefetching using these patterns. Four deltas can be compressed into one signature, which is used to correlate with likely future patterns. This allows for the learning of both simple and complex delta patterns, and allows for the continuing of prefetching off of one page and into the next.

Coverage and **accuracy** are two metrics that are often opposed in prefetching schemes. Coverage represents cache misses that are eliminated by prefetching, while accuracy represents the cache prefetches that are actually useful (referenced by the program). Highly aggressive prefetchers forgo accuracy for high coverage, opting to be less conservative with prefetching decisions by prefetching often and far into the future (with deep lookahead prefetching). However, highly aggressive prefetchers have drawbacks in that they utilize more resources and lead to unnecessary cache pollution. To mitigate the opposition of coverage and accuracy, SPP implements a throttling mechanism, which controls prefetching lookahead depth dynamically based on the predictions made on the current signature path (series of signatures). This allows for prefetch aggressiveness to be throttled according to the current confidence in prediction, allowing for mitigation of the loss in accuracy associated with highly aggressive prefetchers.

B. Perceptron Prefetch Filter

While many prefetchers (such as SPP) aim to balance accuracy and coverage through controlling prefetching decisions, the mechanisms used to balance these are often rudimentary [2]. Additionally, the two metrics are still at odds with each other. Increased aggressiveness with greater coverage still leads to decreased accuracy; schemes such as SPP do not mitigate this fact, but instead aim to find a "sweet spot" to balance the two metrics.

An alternative to this can be found in Perceptron-based Prefetch Filtering (PPF) [2], which intentionally permits a highly aggressive prefetcher by filtering out prefetches predicted to be inaccurate from an existing base prefetcher. This allows for an aggressively-tuned prefetcher to still have high accuracy, through learning which prefetches are useful or not.

PPF uses perceptron learning to predict which prefetches will be useful or not. Perceptrons are simple neural networks that connect several input features via weighted edges to an output unit. The perceptron allows for multiple features and their weights to be considered in a binary prediction mechanism, while remaining simple enough to use in hardware. PPF uses perceptrons as a predictive filter to be attached to existing hardware prefetchers. The filter is trained using prefetch result data to predict whether a prefetch is going to be useful or not, to determine whether a given prefetch should actually be fetched. This method increases both the coverage and accuracy of the prefetcher, as the filter handles increasing accuracy entirely on it's own without prefetcher involvement. This allows for the underlying prefetcher to focus on maximizing coverage, without needing to take accuracy into account

A variety of information obtained within the hardware can be used as fea to train a perceptron to predict whether a given prefetch will be valuable. However, little semantic information from the original code is preserved at this level for the prefetch filter to use. Semantic information gained from compile-time static analyses may be useful as features in perceptron prefetch decisions - one example is trip count, which estimates how many times a loop will iterate. Typical implementations of prefetching done at compile time with statically-obtained information usually focus on inserting software prefetch instructions (an example can be found in [3]). However, software prefetching is not ideal - it often suffers from timeliness issues and additional overhead compared to hardware prefetching.

Therefore, communicating static analysis information to the hardware's PPF may allow for semantic information to be used in prefetch decisions, without the tricky pitfalls of software prefetching. Existing schemes to communicate static analysis information often introduced new instructions to existing architectures, such as in [4] which introduces new load opcodes for the AArch64 architecture. However, modifications to the ISA are tricky to justify, and cause portability issues between different architectures. An alternate solution we propose is storing features obtained from static analysis information in available special-purpose registers, using existing instructions. This way, the PPF may retrieve statically-determined features stored inside of this register, allowing the use of hints provided at compile-time. This does not require modifying the ISA, and has the benefit of the information being preserved within the register until the value is updated. Additionally, it is much easier to justify an additional special-purpose register in a hardware design that uses this approach, compared to an approach requiring changes to the ISA.

C. LLVM Target-Specific Passes

Following the middle-end passes of the compiler, intermediate representations must be translated to machine code. This translation to machine code makes up the back-end of the compiler, which contains many target-specific optimizations. These optimizations rely on architecture-

specific features, and involve directly interacting with the instruction set; one example can be found in the AArch64 load store optimizer (found in `llvm/lib/Target/AArch64/AArch64LoadStoreOptimizer.cpp` in the LLVM source code), which performs AArch64-specific peephole optimizations, combining stores with their immediate loads. These target-specific passes often access and use the results of static analyses; for example, the AArch64 load/store optimizer uses static alias analysis results to detect whether other stores may alias to an address that a load instruction uses, and avoid combining these loads with their immediately paired stores.

Although these target-specific passes are often used to create target-specific optimizations, they also provide the opportunity of providing an interface for the compiler to interact with the hardware. Instructions may be directly modified at this level while using the results of static analysis passes, providing an opportunity for static analysis information to be propagated to the hardware via software-hardware co-design. If static analysis information is encoded and contained in an instruction, and proper mechanisms are provided in the hardware to decode this information, then there is the potential to leverage static analysis information within the hardware via hint-passing.

However, it may be difficult to directly pinpoint the potential uses of this static analysis information. Implementing an accurate prefetching scheme using the wide breadth of static analysis information available may take a significant amount of trial and error and research and development. Alternatively, we opt to instead provide this static analysis information to the perceptron prefetch filter; this allows for static analysis information to be included in prefetching decisions, where the importance of each feature is automatically learnt by the perceptron filter, instead of being manually determined.

III. Implementation

In our implementation, we choose to:

a) Implement the SPP scheme in gem5.

b) Extend SPP with the original PPF design, to enable prefetch filtering.

c) Create a configuration file for gem5 to simulate the system shown in Table 1 of [2].

d) Develop an LLVM transformation pass that collects relevant existing static analysis results, and inserts instructions to store these results to architecture-specific special purpose registers.

e) Modify gem5 to interpret the results of the LLVM transformation pass as saved to the registers, and provide this information as features to PPF.

A. Signature Path Prefetcher

We were happily surprised that gem5 v24 provided two existing implementations of SPP that we could build off of. We decided to copy an existing version of SPP (found in `gem5-24.0.0.1/src/mem/cache/prefetch/signature_path.cc`) and extend it with the Perceptron Prefetch Filter mechanism by inserting the filtering logic immediately before prefetches are queued.

B. Original Perceptron Prefetch Filter

We first extend our copied version of SPP in gem5 to include the original version of the PPF, modelled as closely as possible to the original work [2]. Figure 2 shows the PPF architecture as modelled in our gem5 simulation.

To follow the implementation of PPF, we add four new data structures alongside SPP's existing Signature Table and Pattern Table. These structures are the fully associative Prefetch Table and Reject table, a nested vector of saturating counter weights, and a struct to represent the features that are used as inputs to the perceptron. The Prefetch and Reject tables both allow for 1024 entries, and use the LRU replacement policy. As described in [2], a nested vector of saturating counter weights can be used as a hashed perceptron mechanism [5]. The hashed perceptron mechanism entirely replaces the typical costly multiplication of weights with features with cheaper indexing.

Next, we implement inference and training algorithms. The original SPP implementation will construct signatures based on previous memory access patterns, and will iteratively look ahead and add new addresses to the prefetch queue, as long as SPP is confident enough in the prefetch. We insert our inference algorithm between the time when SPP calculates its next address to prefetch and when that address is inserted into the prefetch queue. We collect the features relevant to the potential prefetch as described in PPF (page address XOR confidence, cache line, page address, physical address, confidence, hash of the previous three instruction PCs, SPP signature XOR delta, PC OR depth, and PC XOR delta), and use the features as indexes into our weights tables (following the hashed perceptron scheme [5]). To enable the feature using the previous three instruction PCs, we added a new FIFO structure into the gem5 o3 pipeline (found in `gem5-24.0.0.1/src/cpu/o3/pc_fifo.cpp`) to track recent PCs. If the outcome of the inference is above a given threshold, the filter accepts the prefetch candidate from SPP and adds it to the prefetch queue, otherwise the candidate is rejected (filtered out) and will not be added to the prefetch queue. If a candidate is accepted, it is added to the Prefetch

table alongside some metadata, indexed by its address. If a candidate is rejected, it is added to the Reject table in a similar way.

In order to train the weights for our perceptron filter, we followed the training mechanism as described in [2], using the hashed perceptron access method as described in [5]. In this, we modified the base cache's probing notification system to call our training algorithm on a cache demand access and a cache eviction. On a cache demand access, we search both the Prefetch Table and the Reject Table for the demanded target address. If there is a valid match for the target in the Prefetch table, then the filter's prediction was correct, and we increment all associated weights for the prefetch features by one. Otherwise, if there is a valid match in the reject table, we also increment the associated saturating counters in the prefetch table by one because the address should have been prefetched, not rejected. On a cache eviction, the Prefetch Table is searched - if there is a match, the weights associated with the features of the prefetch are decremented by one. This is because an entry being evicted while also being in the prefetch table indicates that the entry was prefetched when it shouldn't have been, leading to it not being used.

We make two small changes to SPP itself. The first is that the lookahead and prefetch confidence thresholds are reduced from 0.5 and 0.75 to 0.3 and 0.55 respectively. This allows SPP to prefetch more aggressively, permitting SPP to focus on prefetch coverage while our PPF focuses on prefetch accuracy. Secondly, the original SPP included an auxiliary next line prefetcher when SPP finds no potential candidates. In our implementation the auxiliary prefetcher is removed, however, our PPF can be extended in the future to handle the outputs of this auxiliary prefetcher.

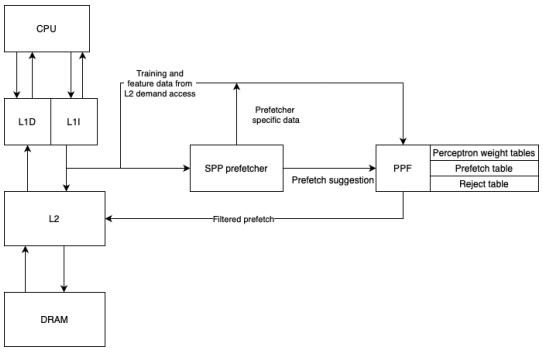

Figure 2: Our perceptron prefetch filtering architecture as modelled in gem5, aligning with the PPF architecture. Diagram adapted from Figure 4 in [2].

C. Configuration file

Our configuration file (found in `gem5-24.0.0.1/configs/perceptron prefetcher evaluation/arm`) is configured to use a similar setup to that in [2]. However, we opt to forgo the last-level cache, and only have L1 and L2 caches. This is because it was not feasible to share the same prefetch information between multiple different caches in gem5.

D. LLVM Target-Specific Pass

1) *Pass Structure:* We chose to develop a target-specific pass for the AArch64 architecture that passes static analysis information to the hardware via inserting instructions. While middle-end passes have the potential to easily be made into pass plugins for `opt`, target-specific passes do not benefit from this advantage; these passes cannot be made into `opt` pass plugins as `opt` only runs on IR. Instead, we have to make direct modifications to LLVM, to invoke our target-specific pass when compiling for a specific target [6].

We first investigated the avenue of inserting instructions before each memory access instruction, which save static analysis information to available special-purpose registers. In this line of investigation, we searched for special purpose-registers that could be modified without causing side effects or changing program semantics. We came across the `TPIDR_EL0` register, which are used for thread-specific storage in some ARM processors, are highly implementation-specific, and are modifiable in user-space using the `MRS` and `MSR` instructions. We also discovered that the AArch_64 architecture has available optional `TPIDR_ELx` registers. One example can be found in the `TPIDR2_EL0` register, which shares the same purpose as `TPIDR_EL0` and is only used when the `FEAT_SME` extension is implemented. We chose to repurpose this register as it is optional and has an implementation-defined use; however, we concede that this has the potential to lead to undefined behaviour in architectures that use this register in the `FEAT_SME` extension. We note that the careful repurposing of this register would not be needed in an actual hardware implementation; a hardware vendor could implement a new system register for this purpose instead of repurposing an existing one.

We decided to collect all of our statically-discovered feature values into a 64-bit value so it could fit in the 64-bit `TPIDR2_EL0` register (shown in Figure 3). We identify the instructions that access memory using `MachineInstruction::mayLoad()` and `MachineInstruction::mayStore()`. Then, we inserted instructions that load this 64-bit value into the `TPIDR2_EL0` register immediately before any instruction that accesses memory. The instructions inserted before the memory access instructions create a new temp register, load the 64-bit aggregated feature value into the temp register, and

then use the `MSR` instruction to copy the contents of the temp register into the `TPIDR2_EL0` register. We also avoided duplicate unnecessary storage of the `TPIDR2_EL0` features for memory instructions that follow each other and have the same feature values. This was done by checking whether relevant features are identical between the current memory access instruction and the last instruction with instructions inserted before it. We chose not to insert the instructions if this is the case, as the register values would not change between the two - leading to redundant instructions.

2) *Static Analysis Features Chosen:* We decided to consider a combination of binary and non-binary features, all encoded as portions of a 64-bit register. We also mostly focused on loop-related static analysis information, as this semantic information is highly relevant to prefetching decisions, and often lost at the hardware level. Additionally, we found that loop-related static analysis information was easiest to obtain for target-specific passes, as machine basic blocks can be directly associated to IR basic blocks via `MachineBasicBlock::getBasicBlock()`. The layout of our features in the `TPIDR2_EL0` register can be found in Figure 3.

Is instr in loop (1 bit): This feature is a boolean value obtained from `LoopInfo`, and indicates whether a memory access is inside of a loop or not. This is relevant for prefetching as memory accesses that are inside a loop may be likely to be accessed again.

Is instr in loop header (1 bit): This feature is a boolean value obtained from `LoopInfo`, and indicates whether a memory access is in a loop header. This is relevant for prefetching, as the loop header basic block must be executed every loop iteration. Therefore, it is likely that an instruction inside a loop header is executed more often than an instruction that within a loop (which may be guarded by conditionals, inside inner loops, and only executed on some iterations).

Does loop have abnormal exits (2 bits): This feature is a boolean value obtained from `ScalarEvolution`, and indicates whether an instruction is in a loop (first bit) and whether the loop has abnormal exits (second bit). An abnormal exit is defined as either throwing out of the function, or entering an infinite loop in a function callee. A well-defined loop must exit through an explicit edge in the CFG. This is relevant for prefetching, as a loop that has a high trip count but uses an abnormal exit may not access a memory instruction again.

Is loop finite (2 bits): This feature is a boolean value obtained from `ScalarEvolution`, and indicates whether an instruction is in a loop (first bit) and whether the loop is **assumed** to be finite (second bit). This is relevant to prefetching as a memory access instruction inside of an infinite loop will have a very high likelihood of being re-accessed multiple times.

Loop depth (13 bits): This feature is a unsigned integer value obtained from `LoopInfo`, and indicates whether an instruction is in a loop (first bit) and the loop nesting level of the block the instruction is in (remaining 12 bits). This is relevant to prefetching as memory accesses in highly-nested inner loops are very likely to be accessed recently and often.

Max trip count (13 bits): This feature is a unsigned integer obtained from `ScalarEvolution`, and indicates whether an instruction is in a loop (first bit) and the estimated upper bound on the loop trip count (remaining 12 bits). This is relevant as memory accesses in loops with very high trip counts are more likely to be accessed multiple times.

# of instructions in function (16 bits): This feature is a unsigned integer obtained directly from the `MachineFunction`. This indicates how many instructions are in a function, which is relevant for prefetching as functions with fewer instructions that are called inside of a loop are more likely to have repeated instances of the same memory access instruction.

Block size (16 bits): This feature is a boolean value obtained directly from the `MachineBasicBlock`. This indicates how many instructions are in a basic block, which is relevant for prefetching as basic blocks for similar reasons as the number of instructions in a function.

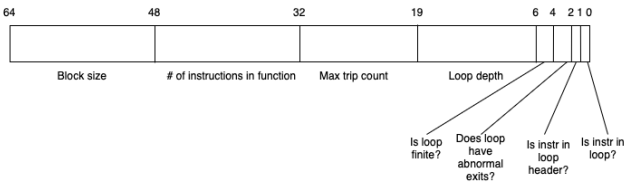

Figure 3: Breakdown of the features stored in the 64-bit `TPIDR2_EL0` register. Note that all loop-related features include an extra bit for their relevance to a memory-access instruction.

E. Extended Perceptron Prefetch Filter with Static Features

To extend our PPF implementation, we first duplicate our existing PPF prefetcher, and extend the struct of features to include room for the new LLVM features. We then modify the O3 pipeline to read from the `TPIDR2_EL0` register at the execute stage in the pipeline. If the currently executing instruction is a memory access instruction, then value of the `TPIDR2_EL0` register is read and inserted to a new table data structure named `FeatureCommunicationTable`, which is indexed by the PC of the executing memory instruction. In

our extended PPF implementation, we index this table on every triggered prefetch to extract the static features of the provided memory access triggering the prefetch. These eight static features are then included with the perceptron model alongside the existing nine PPF features.

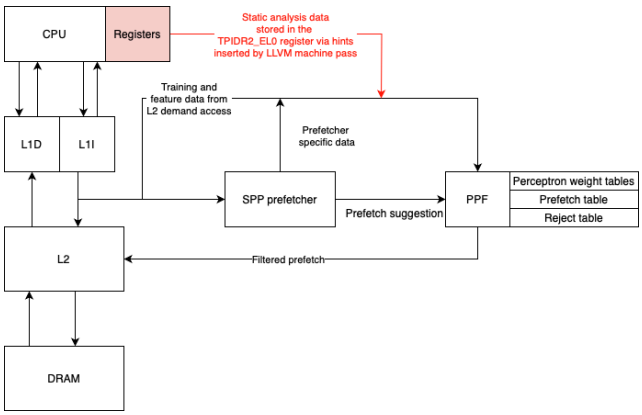

Figure 4: The modified perceptron prefetch filtering architecture, with static analysis feature hints. Changes required are shown in red. Diagram adapted from Figure 4 in [2].

IV. Limitations

While we were able to complete our implementation of a PPF with static features alongside a LLVM target-specific pass that propagates these features to the hardware, we were not able to fully evaluate it's effectiveness using the SPEC CPU 2017 benchmark suite and the gem5 architecture simulator. While we were able to cross-compile the SPEC CPU 2017 binaries to fit our gem5 configuration without a sysroot using `gcc`, `clang` required a sysroot with multiple header files and libraries that we found very difficult to obtain. This was a major roadblock, as the LLVM target-specific pass was only compatible with `clang`. Additionally, even with the `gcc`-compiled binaries, due to hardware constraints, our development environment was not able to properly simulate the benchmark suite using gem5 without running out of memory. To add on top of this issue, the provided undergraduate lab machines with greater memory resources did not have enough storage allocated per user to contain our modified LLVM and gem5 builds. These areas were a major roadblock for evaluation, preventing our ability to aggregate detailed statistics for our implementation.

Additionally, the specific usage of the `TPIDR2_EL0` register is questionable. Although this register is optional, there may be issues if a specific architecture chooses to use this register under the hood. Additionally, the `MSR` instruction required to set the register are privileged in some architectures, and can only be set in kernel mode - even though the register itself is said to be usable in user mode. This may prevent the usage of this register in specific architectures. However, as mentioned earlier, this is only an issue for situations where a new register cannot be added - a hardware vendor using this approach would be able to create a new special-purpose register instead of hijacking an existing one.

V. Next Steps

The immediate next steps for this project would be to obtain a working sysroot to cross-compile the SPEC CPU 2017 benchmark suite for the architecture specified in the gem5 config file. Additionally, we would need to complete the evaluation on a machine with more resources, to prevent resource-intensive benchmarks such as `mcf` from crashing. Once these things are resolved, we would be able to to collect data on how prefetcher accuracy and coverage is affected, and aggregate the results into graphs and present them.

Additionally, a further line of investigation would be to look at alternatives to using the `TPIDR2_EL0` register. While it is suitable for these purposes, a more effective approach would be to add a new register intended for this purpose.

VI. Concluding Remarks

- Many existing prefetchers opt to detect and predict complex memory access patterns.
- The state-of-the-art SPP prefetcher balances accuracy and coverage within the prefetcher by throttling lookahead depth - while the PFF extends SPP by delegate all accuracy measures to a prefetching filter to allow for aggressive coverage maximization within the prefetcher.
- Communicating static analysis information to the hardware allows for the prefetching schemes to be informed of important semantic information lost at lower levels, such as loop information.
- A target-specific LLVM backend pass can be used to insert instructions that store static analysis information for a memory access instruction to a register, which can then be accessed by the prefetching mechanism.
- While our evaluation was limited due to resource considerations, we were able to implement 1) SPP, 2) original PFF, 3) a LLVM target-specific pass that saves static analysis information to a register, and 4) a modification of PFF that uses the static analysis information stored in the register as a feature.
- Next steps are to acquire improved hardware resources and undergo evaluation of our system with the SPEC CPU 2017 benchmark suite.

References

[1] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12. doi: 10.1109/MICRO.2016.7783763.

[2] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-Based Prefetch Filtering," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2019, pp. 1–13. Accessed: Oct. 04, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/8980306

[3] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, in ASPLOS VII. New York, NY, USA: Association for Computing Machinery, Sep. 1996, pp. 222–233. doi: 10.1145/237090.237190.

[4] L. Panayi, R. Gandhi, J. Whittaker, V. Chouliaras, M. Berger, and P. Kelly, "Improving Memory Dependence Prediction with Static Analysis." Accessed: Sep. 26, 2024. [Online]. Available: http://arxiv.org/abs/2403.08056

[5] D. Tarjan and K. Skadron, "Merging path and gshare indexing in perceptron branch prediction," *ACM Transactions on Architecture and Code Optimization*, vol. 2, no. 3, pp. 280–300, Sep. 2005, doi: 10.1145/1089008.1089011.

[6] Radhika Ghosal, "Writing a MachineFunctionPass in LLVM." Accessed: Dec. 12, 2024. [Online]. Available: https://www.kharghoshal.xyz/blog/writing-machinefunctionpass