Informed Prefetching for Indirect Memory Accesses[1]

Mustafa Cavus†, Resit Sendag† and Joshua J. Yi††

† Dept. of Elect., Comp. and Biomed. Eng., Univ. of Rhode Island, Kingston, RI 02881, USA. (mcavus, sendag)@uri.edu
†† U.S. Courts, Western District of Texas. joshua.yi@gmail.com

ABSTRACT

Indirect memory accesses have irregular access patterns which limit the performance of conventional software and hardware-based prefetchers. To address this problem, we propose the Array Tracking Prefetcher (ATP) which tracks array-based indirect memory accesses using a novel combination of software and hardware. ATP is first configured by special metadata instructions, which are inserted by programmer or compiler to pass data structure traversal knowledge. It then calculates and issues prefetches based on this information. ATP also employs a novel mechanism for dynamically adjusting prefetching distance to reduce early or late prefetches. ATP yields average speedup of 2.17 as compared to a single-core without prefetching. By contrast, the speedup for conventional software and hardware-based prefetching is 1.84 and 1.32, respectively. For four-cores, the average speedup for ATP is 1.85, while the corresponding speedups for software and hardware-based prefetching are 1.60 and 1.25, respectively.

1 Introduction

Traversing sparse matrices and graphs frequently results in indirect memory accesses, which have irregular access patterns and thus poor cache spatial locality. These data structures are often implemented as nested arrays, e.g., A[B[i]], wherein the index of the outer array is the value stored in a memory location within the inner array. A hardware stream prefetcher can effectively prefetch entries of array B because its entries are accessed sequentially, thus having good spatial locality. By contrast, because there may be no pattern to the values stored in array B, there is likewise no pattern in the accesses to array A, which diminishes the efficacy of a hardware stream prefetcher.

Software prefetching can hide the memory latencies of indirect memory accesses, but requires executing additional instructions (*e.g.*, the prefetching instructions themselves, instructions for address calculation and border checking, *etc.*). *See, e.g.*, [1] (describing a compiler-based system to generate software prefetches for indirect memory accesses). Figure 1 shows the percentage increase in the number of instructions in a loop iteration due to software prefetching for various benchmarks. For some benchmarks, *e.g.*, *Integer Sort* (*is*), and *Conjugate Gradient* (*cg*), the overhead is very high (*e.g.*, approximately 100%) as those benchmarks have relatively few instructions in each iteration. *Graph500 (g500)* has a high instruction overhead due to border checking instructions in the frequently executed loop. While the benefit of prefetching may offset the overhead for some benchmarks (*e.g.*, *is and g500*), for others (*e.g.*, *cg*), the reverse is true.

In addition to instruction overhead, the benefit of software prefetching is further limited by (1) dependences related to prefetch address calculation, which may reduce the prefetch distance (*i.e.*, how far (in terms of number of iterations) in advance of the memory access the prefetch instruction is issued), and (2) the lack of run-time information, which is required to optimally place the prefetch instructions. Figure 2 depicts the effect of prefetch distance on speedup for two benchmarks, *Histogram* (*histo*) and *PageRank* (*pr*). For *histo*, the highest speedup occurs at the largest prefetch distance (128) while the speedup is lower for smaller distances, especially for prefetch distances of 1, 2, 4, and 8. The opposite is true for *pr*, namely, the highest speedups occur at the smallest prefetch distances. *histo* has an indirect memory access behavior of the form `A[B[i]]` and as shown by Ainsworth and Jones [1], larger distance values (e.g., 32 or 64) are more effective for simple two-level indirect accesses. For *pr*'s `A[B[i][j]]` structure, because the second dimension of array `B`, *i.e.*, `j`, is short (16 for the inputs we

[1] This paper is an extended version of the following conference paper: Mustafa Cavus, Resit Sendag, Joshua J. Yi, "Array Tracking Prefetcher for Indirect Accesses," IEEE International Conference on Computer Design (ICCD), Orlando, FL, USA, Oct 7-10, 2018. pp. 132-139.
This work was partly supported by NSF grant #1422516.

used), prefetch addresses are calculated based on the first dimension instead. This increases the number of instructions executed between two consecutive accesses to the first dimension of array `B` which suggests using shorter distances. For a set of memory-bound benchmarks with indirect memory accesses, the prefetch distance has a very significant effect on the speedup of software prefetching. More specifically, the average speedup across all benchmarks ranges from 1.19 (worst) to 1.84 (best). Finally, it is also important to remember that the optimal prefetch distance for a given application may change based on running the application on a different underlying architecture [1], which further underscores the necessity of run-time information to optimally place the prefetch instructions.

Hardware prefetchers, by contrast, do not require core pipeline executing additional instructions in order to compute and issue prefetches. But, in order to capture irregular access patterns, hardware prefetchers generally require very complex mechanisms. *See, e.g.*, Hashemi et al. [2] (describing continuous runahead execution). Yu et al. [3] proposed a pure hardware mechanism called Indirect Memory Prefetcher (IMP) which was designed to capture a few different indirect memory access patterns (*e.g.*, `A[B[i]]` and `A[B[C[i]]]`).

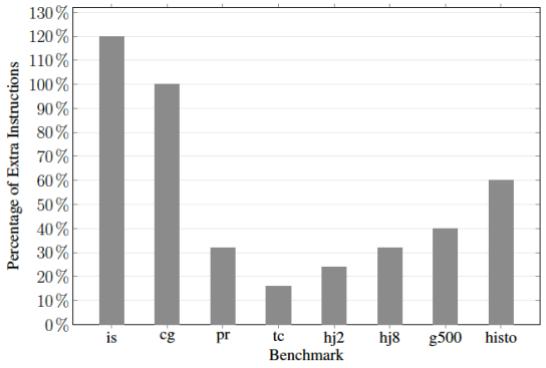


Figure 1: Instruction overhead of software prefetching as a percentage of the instructions in the main loop.

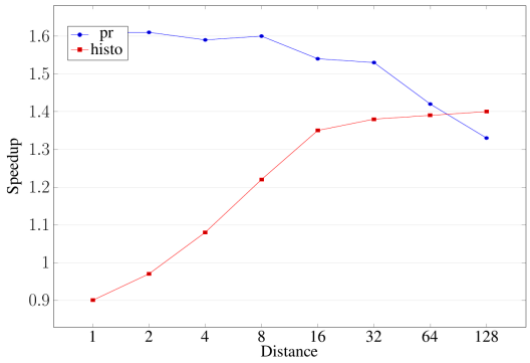


Figure 2: Effect of prefetch distance on software prefetching speedup.

Code Snippets 1.a-c below illustrate the limitations of hardware prefetching and concomitantly the advantages of software prefetching. We use IMP as an exemplary hardware prefetcher given its efficacy and relatively low complexity. Code Snippet 1.a depicts an indirect memory access where the index array, *i.e.* B, is a multidimensional array. This type of code appears in benchmarks such as *PageRank (PR)* and *Triangle Count* (*TC)*. A hardware prefetcher like IMP can capture this indirect memory access by prefetching `A[B[i][j+D]]` where `D` is the prefetch distance. Even when the maximum number of iterations for the inner loop is very small, IMP can still capture these indirect memory accesses, but it may not be able to fully hide the memory latency as the prefetch distance is too small. But, in this case, increasing the distance actually decreases performance because inner loop is too short. Software prefetchers solve this problem by prefetching memory accesses for the next iteration in the outer loop, *e.g.*, `A[B[i+4][j]]`; hardware prefetchers, however, have trouble detecting this behavior.

```
for (i = 0; i < N; i++)
   for (j = 0; j < M; j++)
     load A[B[i][j]]
```

(a)

```
for (i = 0; i < N; i++)
   load A[(B[i]&0x3f)>>2]
```

(b)

```
for (i = 0; i < N; i++)
   load A[B[C[i]]]
   load D[C[i]]
```

(c)

Code Snippets 1: (a) Two-dimensional array in a nested loop (b) Index requires arithmetic/logical computations. (c) Code requires simultaneous tracking of multiple indirect accesses of varying depth using the same index array.

Code Snippet 1.b depicts an indirect memory access that requires arithmetic/logical operations to compute the memory address. More specifically, the index to array `A` requires a logical AND and a right-shift. This type of code appears in benchmarks such as *HashJoin ph2* (*hj2)*. Most hardware prefetchers such as IMP cannot capture these memory accesses because they require more than one arithmetic/logical operations; expensive hardware prefetchers, *e.g.*, continuous runahead

execution [2], can successfully prefetch this type of indirect memory access but only when runahead is sufficiently far and dependence chain can be successfully detected.

Code Snippet 1.c depicts an example of when multiple indirect accesses of varying depth appear at the same time. In this situation, because tracking these memory accesses in hardware is very complicated, IMP does not capture the full behavior. More specifically, IMP was able to detect and prefetch `B[C[i]]` and `D[C[i]]`, but not `A[B[C[i]]]`. IMP can track `A[B[C[i]]]`, but only if it does not exist at the same time as `D[C[i]]`.

By contrast, software prefetching can accurately prefetch the memory accesses depicted in the above code snippets, but only with substantial programmer effort. Also, prefetching these memory accesses requires significant overhead because it requires performing the arithmetic/logical operations for every prefetch. Lastly, the best prefetching distance is hard to predict due to lack of run-time information.

Given that software and hardware prefetchers have different strengths and weaknesses, in this paper, we propose a prefetch mechanism that attempts to combine the strengths of each. More specifically, we propose the Array Tracking Prefetcher (ATP) which tracks array-based indirect memory accesses such as `A[B[i]]`, `A[B[C[i]]]`, `A[B[i][j]]`, and `A[func(B[i])]` where `func()` comprises some arithmetic and binary operations, and combinations of these array-based indirect memory access types. Rather than using software to insert prefetching instructions, ATP uses special metadata instructions to pass data structure traversal information to the hardware component. These metadata instructions execute outside of the loop, so they do not result in significant instruction overhead. ATP's hardware component uses the data structure traversal information to configure itself, based on the behavior of the software which can significantly reduce the training time. Furthermore, by passing this traversal information, ATP's hardware component does not need to detect the indirect memory access behavior itself, which simplifies the complexity of the hardware while still enabling it to prefetch a wide variety of indirect memory accesses.

Concurrent with our work, Ainsworth and Jones [51] proposed a more general system with programmable RISC cores to implement an event-triggered prefetcher (ETP). While this method would perform well for capturing complex indirect access patterns, it does this with higher hardware cost, more complex programming interface and relatively expensive operations per event. ATP is custom hardware that can be configured with sufficient programmability for a variety of access patterns It is configured once as opposed to each time a trigger event is observed. In terms of prefetching indirect access chains, ATP and ETP are quite different. While ETP uses an event-triggered methodology, ATP uses pipelined prefetch model. In our early evaluations of ATP, we abandoned the event-triggered model because pipelined prefetch model achieved better performance on our workloads.

Across a set of memory-bound benchmarks, for a single-core architecture, ATP achieved an average speedup of 2.17X (with a maximum of 5.57X) over the no prefetching baseline. By comparison, software prefetching (using manually-inserted and hand-tuned prefetching instruction) had an average speedup of 1.84X while a state-of-the-art indirect hardware prefetcher (IMP) had an average speedup of 1.32X only. For a 4-core architecture, ATP had an average speedup of 1.85X (up to 5.02X) while software prefetching and IMP had average speedups of 1.60X and 1.25X, respectively.

2 Array Tracking Prefetcher (ATP)

ATP is an integrated software/hardware approach to prefetching indirect data access patterns. The remainder of this section describes the software and hardware components in more depth.

2.1 ATP's Software Component

The software component of the ATP extracts information related to indirect memory accesses within a loop and passes this information to the hardware component. The programmer can manually mark this loop as shown in the Code Snippet 2.a or can use the compiler to automatically identify the loop using an approach that is similar to that described in [1]. The software component passes this extracted indirect-access information to the hardware component through special metadata instructions called *Array Tracking Instructions* (*ATIs*), as shown in Code Snippet 2.c.

2.1.1 Array Tracking Instructions (ATIs).

Each ATI is a single 6-byte long instruction (two bytes for the opcode, 2-bits to specify the type of the ATI instruction, and the remainder for the operands). When a core detects an ATI instruction, the core removes it from the pipeline and forwards it to the ATP hardware. Because ATI instructions appear only once for each main loop where indirect access traversals occur, the number of executed ATI instructions is insignificant.

There are four types of ATIs: `ATCL`, `ATAR`, `ATRL`, and `ATOP`. `ATCL` clears all ATP tables. It does not have any operands. `ATAR` inserts entries into the ATP's Array Table (AT). It has a single operand, a 16-bit offset (from `ATAR`'s PC), which is used to compute the PC of the load instruction that accesses the index or target array involved in indirect accesses.

```
                        #at_indacc_loop
                        for (i = 0; i < N; i++)
                             sum += A[B[i]];
```

(a)

```
4005a0:   movslq    (%rax), %rcx        ; load B[i]
4005a3:   add  $0x4, %rax
4005a7:   add  (%rsp, %rcx, 4), %edx    ; load A[B[i]]
4005aa:   cmp  %rsi, %rax
4005ad:   jne  4005a0
```

(b)

```
atcl                              ; clear AT tables
atar $0x4005a0                    ; Pass PC of the load for array B
atar $0x4005a7                    ; Pass PC of the load for array A
atrl $0x4005a0, $0x4005a7, 0      ; Pass the relationship between B and A
```

(c)

Code Snippets 2: Finding indirect accesses in software. (a) Marking the loop for indirect prefetching. Note that the compiler can automatically perform the marking and generate ATI instructions using a pass similar to approach in [1]. (b) Instructions inside the marked loop. (c) ATI instructions generated (these instructions are placed above the entry point of the loop).

The `ATRL` instruction inserts relationship information between two arrays into the Indirect Relation Table (IRT), which is described in more detail in Section 2. For example, for the `A[B[i]]` structure, the software mechanism uses an `ATRL` instruction to insert an entry into the IRT in order to account for the relationship between target array `A` and index array `B`. `ATRL` has three operands: the first two are the PCs (offsets) of the load instructions accessing the index and target arrays, while the third operand (1-bit) specifies the type of the relation: 0 is used for `A[B[i]]` type accesses while 1 is used for `A[B[i][j]]` type accesses.

The `ATOP` instruction is used to calculate the index of the target array, *e.g.*, `func(B[i])`, in complex data structures, *e.g.*, `A[func(B[i])]`. More specifically, `ATOP` inserts these operations to ATP's Operation Table (OT). `ATOP` has two operands. The first operand specifies the operation type (*e.g.*, NOT, ADD, etc.) while the second specifies the data for that operation. If there is more than one operation, the software mechanism uses multiple `ATOP` instructions, one for each operation. The first operand of the first `ATOP` instruction is implicitly specified as the index array value (*e.g.*, `B[i]`). The second `ATOP` uses the result of first `ATOP` as its first operand. The first `ATOP` instruction always follows an `ATRL` instruction. Table 1 shows example sequences of ATI Instructions for different types of indirect array traversals targeted in this paper. ATP can simultaneously keep track of multiple indirect array access structures with various levels of complexity.

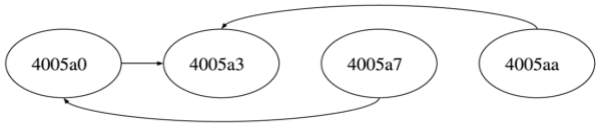


Figure 3. Dependency graph generated from the Code Snippets 2.b.

Table 1. Example sequences of ATI Instructions for different type of indirect array traversals.

Basic (1D index array)	Pointer (2D index)	Multi-level	Multi-way (more than one indirect structure)	Hashed index
A[B[i]]	A[B[i][j]]	A[B[C[i]]]	A[B[C[i]]] and A[D[j]]	A[(B[i]-1) & 0xf]

`atar pcA`
`atar pcB`
`atrl pcB, pcA, 0`
	`atar pcA`
	`atar pcB1`
	`atar pcB2`
	`atrl pcB2, pcA, 0`
	`atrl pcB1, pcB2, 1`
		`atar pcA`
		`atar pcB`
		`atar pcC`
		`atrl pcB,pcA, 0`
		`atrl pcC, pcB, 0`
			`atar pcA`
			`atar pcB`
			`atar pcC`
			`atar pcD`
			`atrl pcB, pcA, 0`
			`atrl pcC, pcB, 0`
			`atrl pcD, pcA, 0`
				`atar pcA`
				`atar pcB`
				`atrl pcB, pcA, 0`
				`atop fsub, 1`
				`atop fand, 0xf`

2.1.2 Generating ATIs.

Generating ATIs consists of two stages. First, the software component generates a dependency graph of the instructions inside a marked loop. Figure 3 depicts the dependency graph for the code loop shown in Code Snippet 2.b (PC 4005a0 refers to accesses to array B and PC 4005a7 refers to accesses to array A in Code Snippet 2.b. Second, the software component generates ATIs based on the type of node or its connections. More specifically, for nodes that correspond to a `load` instruction, the software component generates an `ATAR` instruction. The software component generates `ATRL` instructions (*e.g.*, when one load instruction is used to calculate the address of another load instruction) and `ATOP` instructions (if the index for the outer array requires computation) based on the connections of the nodes. The software component places these instructions above the entry point of the loop. The software component also places `ATCL` instructions before the beginning of the loop in order to clear the ATP tables before the loop begins to execute.

ATIs are often inserted on top of the outermost loop when there are multiple nested loops. If ATIs are inserted above the entry point of the inner loop, the outer loop will be ignored by ATP. In this case, ATP will be initialized each time the inner loop is entered. This is often not desirable and it is only beneficial if the inner loop iterates sufficiently long so that ATP can calculate prefetches for future iterations of the loop.

2.2 ATP's Hardware Mechanism

The hardware mechanism is the part of the ATP that actually prefetches the indirect accesses. More specifically, the hardware mechanism of the ATP uses the information provided by ATIs to initialize the ATP tables. During execution of the loop, the ATP calculates the prefetching stride based on the base address of the required arrays, *e.g.*, the address of `A[0]` for `A[B[i]]` indirect access, and the size of the array type. The hardware mechanism generates prefetch addresses for forthcoming indirect memory addresses based on the base address and predicted stride. The ATP also includes a mechanism to dynamically change the prefetch distance in order to adapt to specific run-time behavior to achieve better timeliness and performance.

2.2.1 High-level description of the operation of ATP's hardware mechanism.

Due to the potential complexity of the hardware mechanism, before diving into the detailed description, it may be useful to first explain the operation of the hardware mechanism from a 30,000 foot view.

The hardware mechanism initiates the prefetch process after the load instruction commits. More specifically, when a load commits, the hardware mechanism checks whether that load instruction corresponds to an index array (by checking if the PC of that load instruction was previously inserted into the array table (AT) via an `ATAR` instruction). If that load instruction does not correspond to an index array, then the hardware mechanism takes no action. But if the load instruction corresponds to an index array, then the hardware mechanism will (eventually) issue prefetches based on the type of indirect access and a (dynamically varying) distance (in terms of the number of iterations). For example, for an `A[B[i]]` type of indirect access, when the load instruction corresponding to `B[i]` commits, the hardware mechanism checks whether `B[i]` is an index array. Because it is in this case, the hardware mechanism will issue prefetches for the future accesses to `A[B[i]]` and `B[i]`. In particular, for an `A[B[i]]` type of indirect access, the hardware mechanism will issue prefetches for `B[i+2*distance]` and `A[B[i+1*distance]]`. Because there is a read-after-write dependence between array `A` and array `B` (i.e., the value at

`B[i]` is used as the index for array `A`), the prefetch for array `B` needs to be for an iteration further into the future than the iteration being prefetched for array `A`.

The hardware mechanism dynamically chooses the `distance` value by testing various distances (2, 4, 8, and 16 in our simulations) and then choosing the value with the best performance. (The hardware mechanism periodically reevaluates what the best distance is.) Assuming that the best distance is 16, then for an `A[B[i]]` type of indirect access, the hardware mechanism will issue prefetches for 32 iterations into the future for `B[i]` (*i.e.*, `B[i+2*16]`) and 16 iterations into the future for `A[i]` (*i.e.*, `A[B[i+1*16]]`).

If the indirect access is of the form `A[B[C[i]]]`, then when the load instruction corresponding to `C[i]` (which is the index array in this case) commits, the hardware mechanism will issue prefetches for `C[i+3*distance]`, `B[C[i+2*distance]]`, and `A[B[C[i+1*distance]]]`.

If the indirect access is of the form `A[B[i][j]]` (where `B[i]` is a pointer to `B[i][0]`), then the prefetch calculation is triggered by the committed load instructions corresponding to either `B[i]` or `B[i][j]`. When the load instruction corresponding to `B[i]` commits, the hardware mechanism will issue prefetches for `B[i+3*distance]`. When the load instruction corresponding to `B[i][j]` commits, the hardware mechanism will issue prefetches for `B[i+2*distance][j]` and `A[B[i+1*distance][j]]`. It is important to note that, we need to access `B[i]` to be able to calculate the address of `B[i][j]`. The prefetch for `B[i]` needs to be issued for an iteration further into the future than `B[i][j]` so the prefetched value of `B[i]` is available when it is needed.

Finally, if the indirect access is of the form `A[func(B[i])]`, then when the load instruction corresponding to `B[i]` (which is the index array in this case) commits, the hardware mechanism will issue prefetches for `B[i+2*distance]` and `A[func(B[i+1*distance])]`. For this indirect structure, the value of `B[i]` is often used as an input to one or multiple arithmetic/binary operations, and hardware mechanism should perform these calculations to be able to calculate the prefetch address of `A[func(B[i])]`.

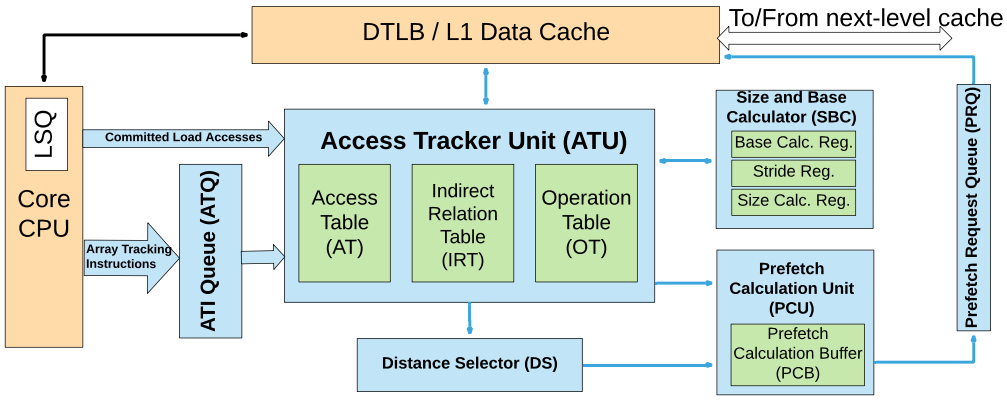


Figure 4: An overview of ATP

2.2.2 Detailed description of the components (and operation thereof) of the hardware mechanism.

Figure 4 depicts the block diagram of the hardware mechanism. It consists of an ATI queue (ATQ), an Access Tracker Unit (ATU), a Size and Base Calculator (SBC), a Prefetch Calculation Unit (PCU), and a Distance Selector (DS). After an ATI instruction has been identified in the processor pipeline, it is forwarded to the ATQ, which is a FIFO queue with head and tail pointers. A sub-opcode field identifies individual ATI instructions. The ATQ is simply the interface between the processor pipeline and the ATP. ATP's prefetch generation consists of four stages: A) First, ATIs in the ATQ are processed and used to initialize the ATU tables. B) Once all ATIs have been processed, SBC starts calculating the sizes and base addresses for index and target arrays, which is needed for prefetch calculation. C) Then, whenever the ATU observes a demand access to an index array, it notifies the PCU to begin the prefetch calculation process and issue prefetches (index array is also called trigger array since prefetch calculations are triggered by accesses to this array). D) Finally, the prefetch distance is dynamically adjusted

based on the feedback from the DS, which finds the best performing prefetch distance using a simple mechanism. The overall process flow is managed by a finite-state machine (FSM). Next, we explain the operation of ATP in these four stages.

A. Processing of ATI instructions and ATP initialization

Each valid ATI in the ATQ is processed in-order from ATQ's head to tail. ATIs are used to initialize/program the ATU tables. The ATU consists of three important tables, the Array Table (AT), the Indirect Relation Table (IRT) and the Operation Table (OT). An `ATCL` instruction resets all the ATU tables, namely valid bits are set to zero in the AT, IRT and OT. We explain how ATI instructions initialize or program ATU tables using the example in Figure 5, which shows the final status of the AT, IRT and OT after they are initialized for `A[(B[i]&0x7F)*14]` structure. The indirect access structure in this example generates two `ATAR` instructions, one for array `A` and one for array `B`. The `ATAR` instructions updates the AT. Each `ATAR` instruction reserves the next available entry in the AT. It specifies a load PC that is involved in reading an array element (and involved in indirect access). The fields in the AT are shown in Figure 5. Detailed description of each field in the ATU tables is given in Table 2. Initially, trigger-bit and depth field of the AT is 1. Trigger type, indirect map, and root fields are all initially 0s.

For the example in Figure 5, following the two `ATAR` instructions is an `ATRL` instruction specifying the relation between arrays `A` and `B`. Each of the PCs specified by `ATRL` have already been placed in the AT due to prior `ATAR` instructions. When an `ATRL` instruction is executed, it allocates an entry in the IRT, locates the index array's PC (`B`) in the AT and updates the indirect map field of the AT entry with the index of the IRT entry it allocated. Then, it locates the target array PC (`A`) in the AT and saves its index in the destination field in the IRT entry. Indirect map field of the AT is a bitmap (each bit refers to an index of IRT entry) specifying if an IRT entry in relation to the array in the current AT entry exists. 00 means no relation exists and thus array in that AT entry is not used as an index for any target array. In Figure 5, AT's entry 1 for array `B` has a non-zero indirect map, 10, suggesting the 0_{th} entry in the IRT table provides in its destination field a pointer to the target array (in the AT) for which array B is used as index for. **If all indirect maps are 0, ATP acts as a stream prefetcher.** This will happen when no `ATRL` instruction is observed for `ATAR` instructions.

Array Table (AT)

valid	load_pc	base_bit	base_addr	trigger_bit	trigger_type	size_bit	size	indirect_map	depth	root_bit	root_addr	root_size
1	PC_A	1	Base A	0		1	8	00	1	0		
1	PC_B	0		1	1	1	4	10	2	0		
0												
0												

Indirect Relation Table (IRT)

valid	destination	type	op_bit	op_idx
1	0	REG	1	0
0				

Operation Table (OT)

valid	op	data	next_bit	next_idx
1	AND	0X7f	1	1
1	MUL	14	0	

ATI Instructions

atar	PC_A
atar	PC_B
atrl	PC_A, PC_B, 0
atop	fAND, 0x7f
atop	fMUL, 0xe

Figure 5. The final state of the AT, IRT and OT when they are initialized for a `A[(B[i]&0x7f)*14]` structure. A is an array of 8-byte doubles and B is an array of 4-byte integers

Level (or depth) of an indirect access depends on the number of indirect accesses made in a chain starting with the access to the index array. AT has a field, depth, monitoring this level. `ATRL` updates the depth field of the base array (`B`) by checking if it is less than the depth of the target array, `A`. If so, it sets the depth of array `B` to be one more than its target array (in this example, 2). The index array has the highest depth and a target array which is not an index to another target array has the lowest depth, 1. Depth is used for prefetch address calculation as described in Section 2.C.

Finally, `ATRL` is followed by `ATOP` instructions since base array is not directly used as index for the target array. The first `ATOP` is an AND with data `0x7f` and the second `ATOP` is a MUL with `14` as its data. `ATOP` instructions can only follow an `ATRL` or another `ATOP` instruction. `ATOP` sets to 1 the op bit of the IRT entry it corresponds to denoting that base array must undergo an operation before used as index for target array. op_idx field specifies the index of the OT that corresponds to the

operation specified by ATOP. If ATOP is followed by another ATOP, the next bit field of the last ATOP is set to 1. Once the last ATOP in the ATQ have been processed, ATU has completed initialization and ATP is ready to move to the size and base calculation stage.

Table 2. Detailed Explanation of the fields in the AT, IRT and OT.

Field Name	Detailed Explanation
AT/IRT/OT.valid	Specifies if the entry is valid or not. Initial value is zero.
AT.pc	The PC of the load instruction that accesses an array element involved in indirect access. This could either be index or target array access PCs.
AT.base_bit	Set when (and if) base address calculation has been completed.
AT.base_addr	Start (base) address of an indirectly accessed array. For example, for A[B[i]] structure, base address is needed for array A (address of A[0]).
AT.trigger_bit	Set if array in this entry is a trigger array. A trigger array must be an index array where its value is used as index to a target array. For multi-level structures, the trigger array is the highest depth (inner most) array.
AT.trigger_type	Two types of trigger accesses are recognized by ATP: regular (type 0) and pointer (type 1). Regular refers to a direct access as in A[B[i]]. Pointer triggers refer to two-dimensional index array structures, such as A[B[i][j]].
AT.size_bit	Set when size calculation has been completed.
AT.size	Size (in bytes) of the array element. This information can be supplied by software. It is, however, easy to compute it at-run-time also.
AT.indirect_map	A bitmap, where each bit refers to a specific IRT entry holding the array's indirect relation to its target array. If multiple bits are set in the bitmap, this array is used as index for more than one target array.
AT.depth	Level of an indirect access. A[B[i]] is a two-level indirect access (depth of A and B are 1 and 2, respectively), whereas A[B[C[i]]] is a three-level indirect access (depth of A, B, C are 1, 2, and 3, respectively). Depth is used for indirect prefetch calculation.
AT.root_bit	Set when root address has been calculated.
AT.root_addr	Used only for trigger type 1 entries. Updated dynamically during prefetch calculation when necessary.
AT.root_size	Size (in bytes) of the root array element size.
AT.node_bit	Set when (and if) it is an access to the first node of a linked list.
AT.node_offset	Offset value from the current node address to the next node's pointer.
AT.num_nodes	Number of expected nodes in each linked list.
IRT.destination_idx	The index of the AT entry that holds the target array for this base (index) array. For A[B[i]], destination idx is the index of the AT that holds array A.
IRT.type	Specifies the indirect relation type. Two types are supported: direct and pointer. In direct relation, the value read from the base (index) array is used as an index or used in calculation of the index for the target array. Code Snippet 1 demonstrates a direct type. Pointer types are used to connect type 0 trigger entries of base arrays to type 1 trigger entries of destination arrays. In pointer type, value read from the base array is used as the root address of destination access. Root addresses are used to calculate prefetch addresses for incoming dimensions in a two-dimensional array.
IRT.op_bit	Set if operations need to be performed on index array for target array index calculation (See Figure 5 as example).
IRT.op_idx	The index of the OT entry that specifies the operation to perform.
OT.op	Operation to perform on index array values.
OT.data	Data needed for operation. The first operand is the index array value if previous entry's next_bit is zero. Otherwise, the first operand is the result of the previous operation.
OT.next_bit	Set if another operation (specified in the next OT entry) needs to be performed after the current operation

B. Size and Base Address Calculation

Before prefetching can start for indirect accesses, the stride of the trigger arrays and, the element sizes and base addresses of the target arrays must be known. ATP employs a single mechanism to compute sizes and base addresses.

Size Calculation: For each committed load instruction (to make sure that accesses are monitored in-order), if its PC is found in the AT table, and if the size bit is zero, ATP's Size and Base Calculator (SBC) starts the process for size computation. First, the stride between two accesses of the trigger array is computed. The SBC uses a stride register to keep track of trigger array accesses. A stride register consists of a valid bit, an index to the AT (holding the trigger array), last address, stride and

confidence fields. If the observed stride (difference between addresses of two consecutive accesses) repeats, confidence counter is incremented. If confidence reaches a certain threshold, the detected stride is saved in the AT entry corresponding to the trigger array and its size bit is set. The stride register is then released to be used by other trigger arrays.

The size calculation for a non-trigger array uses a different method. If the size bit is not set for a non-trigger array in the AT, SBC employs a size calculation register (SCR) for that AT entry, which monitors the values read by index arrays and the resulting addresses of the non-trigger array. SCR consists of a valid bit, two AT index fields, two address fields, two computed index (`idx`) fields, confidence and size fields. SCR holds the AT index or indexes that hold the index array/s for the target array (the non-trigger array). For each committed load, if the index arrays were accessed, the values read from these arrays are recorded in the computed index field (`idx`) of the SCR and if the OT has any entry for the relation related to that index array, the operation/s are performed on these values to compute the final index (`idx1`) for the target array and computed index field of SCR is updated. When an access to the non-trigger array is observed (after access to its index array), its address (`addr1`) is recorded in the SCR. Once SCR observes two addresses (`addr1` and `addr2`) and two indexes (`idx1` and `idx2`), the size of non-trigger array is computed using Equation 1. The size computation is repeated multiple times until a certain confidence threshold is reached. After `size` is computed, its recorded in the AT entry corresponding to the non-trigger array, and SCR is released.

$$\text{Size}(A) = \frac{\text{Addr}\left(\text{A[B[i+1]}\right) - \text{Addr}\left(\text{A[B[i]]}\right)}{\text{B[i+1]} - \text{B[i]}} \quad (1)$$

Base Address Calculation: Finally, before indirect prefetching can be performed for target arrays, their base addresses must be computed. SBC assigns a base address calculation register (BACR) to a non-trigger type entry of a destination array if the base address of it is not yet calculated but the size of it is already known. The value of the `idx` field in the BACR is set to the index of the AT entry which it is assigned for. Like the SCR, BACR also has a field for the indirect map which shows the entries of source (index) arrays in the AT.

After BACR is assigned to an entry in the AT, SBC monitors the accesses requested for any of the source arrays by checking its indirect map field. When SBC is notified by an access to a source array (e.g., `B[i]` in Equation 2), it stores the value of the accessed data in its BACR after performing the required operations in the OT if any operation exists between the source array and the destination array (`value`). Once an access to the destination array (`addr`) is seen by SBC, it calculates the base address based on Equation 2 (here `B[i]` is the `value` and `Addr(A[B[i]])` is the `addr`). If the same base address is computed multiple times to satisfy a confidence threshold, SBC sets the base address field of the corresponding entry in the AT and releases the BACR.

$$\text{Addr(A[B[i]])} = \text{BaseAddr(A)} + (\text{B[i]} \times \text{Size(A)}) \quad (2)$$

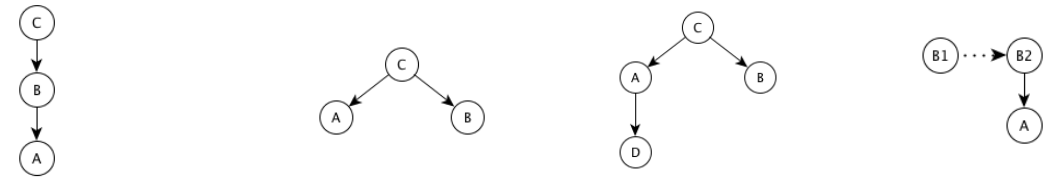
Although the element size and base address of the arrays can simply be passed with an instruction, ATP calculates them dynamically to be able to optimize prefetch calculation for different application behaviors. For example, for simple multiplication operation on the index values in an indirect access structure like in the example of `A[B[i]*3]`, ATP does not need to perform the multiplication. Instead, it can dynamically calculate the size of the array elements three times bigger (which is the stride of the corresponding array accesses). Similarly, an `A[B[i]+10]` can be represented as `A[B[i]]` with `base_addr(A)=addr(A[0])+10`. This will prevent ATP from using an OT entry to calculate `B[i]+10` for each prefetch trigger.

C. Prefetch Triggering and Calculation

As shown in Figure 4, ATP employs a Prefetch Calculation Unit (PCU) to calculate prefetch addresses when triggered. Prefetch calculation in PCU is performed in two steps: 1) prefetch initialization and 2) prefetch address calculation and issuing of prefetches. Upon an access to a trigger array the ATU signals the PCU to start a prefetching operation. The PCU begins the initialization phase by first allocating an entry for the trigger array in the Prefetch Calculation Buffer (PCB). The PCB is a temporary buffer that keeps detailed information about the entries of potential prefetches. Each entry of PCB has a 1-bit field

to show whether the corresponding entry is free for a new task or it is busy calculating a prefetch address. By referring to the ATU tables, the target array/s characteristics are also inserted into the PCB and they are linked to their sources in the PCB entries. Each PCB entry keeps the AT index of the corresponding array, an index pointer which points to the PCB entry of the source array (not used for trigger arrays), and an address field to keep the calculated prefetch addresses. The address field is initialized to the address of the current memory access for trigger arrays, and it is initialized to zero for non-trigger arrays. After the initialization is done, the PCU starts calculating the prefetch addresses for each entry. The address field is updated during prefetch calculation process and the final value of this field is the prefetch address to be issued.

ATP can efficiently calculate prefetch addresses for many different indirect access structures and their combinations. Figure 6 represents each indirect access behavior using a graph model where each entry in the AT is a node and each entry in the IRT is an edge. Since this graph model is in the form of a tree structure, the root of the tree represents the index (trigger) array. A prefetch calculation is performed for each node of this graph. Here, we explain how prefetching is performed for each indirect structure that is shown in Figure 6.



A[B[C[i]]] (a) **A[C[i]] and B[C[i]]** (b) **D[A[C[i]]] and B[C[i]]** (c) **A[B[i][j]]** (d)

Figure 6: Graph representation of different indirect access behaviors.

In a multi-level `A[B[C[i]]]` structure, the index array, `C`, represents the root node of the graph as shown in Figure 6.a. As ATU signals PCU for prefetching operation on an access to the trigger array `C`, an entry for `C` is allocated in the PCB. Since `C` is the trigger array, by following the indirect map fields in the AT, its target array `B` and then `A` are also inserted into the PCB and they are linked to their sources in the PCB entries. In our evaluations, the number of PCB entries is equal to the number of AT entries. Using more entries might be useful in case ATP cannot calculate all the prefetches on time and it can insert new entries into PCB before the previous entries are freed. Since we did not observe this in the benchmarks we use, we designed ATP to drop new prefetch tasks if there is no available entry in PCB. To be able to compute independent prefetch addresses concurrently, each PCB entry has its own ALU. After the initialization is done, the PCU starts calculating the prefetch addresses for each entry. To calculate the prefetch address for any non-trigger array, the PCU needs to read a value from the source array. Assuming that we want to calculate a prefetch address for `A[B[C[i+dist]]]` triggered by an access to `C[i]`, we need to read the value of `C[i+dist]` and then `B[C[i+dist]]` to be able to calculate the prefetch address for `A[B[C[i+dist]]]`. Naturally, to be able to find these (source) values in the cache when needed, they need to be prefetched ahead of time. To be able to prefetch the source values ahead of time, we assign a depth value to each array in the indirect access chain and actual prefetch distance of the corresponding array is calculated by multiplying the depth value with the global prefetch distance of ATP. So for an `A[B[C[i]]]`, a depth-3 indirect structure (the depth of arrays A, B, and C are 1, 2 and 3, respectively), PCU calculates prefetch addresses and performs prefetching for `C[i+3*dist]`, `B[C[i+2*dist]]` and `A[B[C[i+1*dist]]]`.

In general, an access to a trigger array initiates as many prefetches as there are levels in the indirect structure unless the prefetch address computation fails due to cache misses. Table 3 shows the prefetch calculation steps of an `A[B[C[i]]]` access structure. Events under the same step for different paths may occur in parallel. For instance, in step1: for path C, we can compute prefetch address for `C[i+3*dist]`; for path `C`→`B`, we can read `C[i+2*dist]`; and for path `C`→`B`→`A`, we can read `C[i+1*dist]`.

As Table 3 shows, PCU takes the following actions (triggered by an access to `C[i]`) to perform prefetching for an `A[B[C[i]]]` structure.

For node C:
Step 1. Compute the prefetch address for `C[i+3*distance]` using Equation 3 and issue the prefetch.
For node B:
Step 1. Compute the address for `C[i+2*distance]` using Equation 3 and read its value from the L1 cache.
Step 2. Compute the prefetch address for `B[C[i+2*distance]]` using Equation 2 and issue the prefetch.
For node A:
Step 1. Compute the address for `C[i+1*distance]` using Equation 3 and read its value from the L1 cache.
Step 2. Compute the address for `B[C[i+1*distance]]` using Equation 2 and read its value from the L1 cache.
Step 3. Compute the prefetch address for `A[B[C[i+1*distance]]]` using Equation 2 and issue the prefetch.

$$\text{PfAddr} = \text{CurrAddr} + (\text{Size} \times \text{Depth} \times \text{Distance}) \quad (3)$$

Table 3: Prefetch calculation steps of `A[B[C[i]]]` structure.

Path	Step 1	Step 2	Step 3
C	**Prefetch** C[i+3*dist]		
C → B	Read C[i+2*dist]	**Prefetch** B[C[i+2*dist]	
C → B → A	Read C[i+1*dist]	Read B[C[i+1*dist]	**Prefetch** A[B[C[i+1*dist]]]

Figure 6.b shows a graph generated from the multi-way indirect access structure of `A[C[i]]` and `B[C[i]]`. Since the depth of the tree is 2 in this example, prefetch calculation can be done in two steps as shown in Table 4. The indirect access structure of `D[A[C[i]]]` and `B[C[i]]` (both multi-level and multi-way) generates a graph with a depth of 3 as shown in Figure 6.c. The steps of prefetch calculation for this behavior is shown in Table 5. The PCU operation (both initialization and prefetch calculation) for the `A[func(B[i])]` structure is similar; thus, it is not presented.

Table 4: Prefetch calculation steps of `A[C[i]]` and `B[C[i]]` structure.

Path	Step 1	Step 2
C	**Prefetch** C[i+2*dist]	
C → A	Read C[i+1*dist]	**Prefetch** A[C[i+1*dist]
C → B	Read C[i+1*dist]	**Prefetch** B[C[i+1*dist]

Table 5: Prefetch calculation steps of `D[A[C[i]]]` and `B[C[i]]` structure.

Path	Step 1	Step 2	Step 3
C	**Prefetch** C[i+3*dist]		
C → A	Read C[i+2*dist]	**Prefetch** A[C[i+2*dist]	
C → B	Read C[i+2*dist]	**Prefetch** B[C[i+2*dist]	
C → A → D	Read C[i+1*dist]	Read A[C[i+1*dist]	**Prefetch** D[A[C[i+1*dist]]]

Finally, Figure 6.d shows the graph representation of the indirect access structure of `A[B[i][j]]`, which we classify as having a pointer-type relation. The index array in this structure is a two-dimensional array and it is represented by two separate nodes (i.e., separate entries in the AT). Node `B1` represents the first dimension of array `B` (i.e., the load instruction that reads `B[i]`) and node `B2` represents the second dimension (i.e., the load instruction that reads `B[i][j]`). Also, the edge from `B1` to `B2` is represented as a pointer type relation instead of a regular type as for the structures discussed above (for Figure 6.a-c).

In two dimensional arrays, prefetching can be triggered by two different load accesses (`B1` and `B2`). Table 6 shows the steps of prefetch calculation in an indirect access structure of `A[B[i][j]]`. Prefetch address calculation is somewhat more complicated for this structure. When ATU observes an access to `B1` entry, the prefetch address for `B1` is calculated as usual. However, prefetch addresses for `B2` and `A` are not calculated at this moment as the relation type between `B1` and `B2` is a pointer type. Instead, the root address field of `B1` entry in the AT table is updated with the value read by the current access (the value of `B[i]`, which is the address of `B[i][0]`). Prefetching for entries `B2` and `A` are triggered by the access to the `B2` entry. As it is shown in Table 6 when prefetching is triggered by an access to `B[i][j]`, ATP calculates the addresses of `B[i+2*dist][j]` and `A[B[i+1*dist][j]]`. To be able to calculate the prefetch address for `B[i+2*dist][j]`, we need to read the address of `B[i+2*dist][0]` (i.e., the value of `B[i+2*dist]`) which we call the prefetch root address. Using the root address (address of `B[i][0]`) which was saved previously by trigger `B1`, we calculate the address of `B[i+2*dist]`, in which the prefetch root address is stored, by using the first part of Equation 4. Then, we can calculate the prefetch address (address of `B[i+2*dist][j]`) using the second part of Equation 4. This calculation is based on the fact that the distance between `B[i][0]` and `B[i][j]` should be equal to the distance between `B[i+2*i][0]` and `B[i+2*i][j]`. Finally, prefetch address calculation for `A[B[i+1*dist][j]]` is done in a similar way: it calculates the address of `B[i+1*dist][j];` read its value and uses it to calculate the prefetch address for `A[B[i+1*dist][j]]` as shown in Table 6.

Three or more dimensional arrays can be represented in ATP, similarly. For example, an indirect access structure with a three-dimensional index array, `A[B[i][j][k]]`, should have 4 AT entries where 3 of them belong to the dimensions of array B and 3 IRT entries where 2 of them have pointer type relations. In this case, prefetches can be triggered by the accesses to `B[i]`, `B[i][j]`, and `B[i][j][k]` and both `B[i]` and `B[i][j]` should be set as root addresses.

Also, it is common to implement 2D arrays by allocating a 1D array and addressing it as a 2D array by calculating the indexes using a width offset (e.g., `A[i][j]` can be mapped as `A[i*width_offset+j])`. ATP recognizes these as 1D arrays and performs indirect memory prefetch calculations similar to when index array is a 1D array. The support for 2D arrays we mention in this paper is for actual 2D arrays, where each element of the first dimension of the array is a pointer to another array.

Table 6: Prefetch calculation steps of `A[B[i][j]]` structure.

Path (1st dimension)	Step 1	Step 2	Step 3
B1	**Prefetch** B[i+3*dist] Set root addr in the AT to B[i]		
Path (2nd dimension)	Step 1	Step 2	
B2	Read B[i+2*dist]	**Prefetch** B[i+2*dist][j]	
B2 → A	Read B[i+1*dist]	Read B[i+1*dist][j]	**Prefetch** A[B[i+1*dist][j]]

$$\text{PfRootAddr} = \text{RootAddr} + (\text{Size} \times \text{Depth} \times \text{Distance})$$
$$\text{PfAddr} = \text{Mem}[\text{PfRootAddr}] + (\text{AccessedAddress} - \text{Mem}[\text{RootAddr}]) \quad (4)$$

D. Adaptive Prefetching Distance Selection

Distance Selector (DS) enables ATP to adjust the prefetch distance (in terms of how many array elements ahead) dynamically to be timely accurate on different applications and configurations.

Each power of 2 prefetch distance from 2 to 16 is competed during a test period and at the end of this period, the distance that takes the smallest number of cycles to complete the same number of loop iterations is picked as the distance for the acting period that comes after the testing period. The acting period is a fixed 50 times larger than the testing period. After acting period another testing period follows. In testing period, each prefetch distance is run for a fixed number of loop iterations (64 in our experiments of which the first 32 are used for warm-up and next 32 are used for performance measurement) in a round robin fashion and the number of cycles is counted. DS employs two 32-bit cycle counters. `min_count` holds the smallest cycle count and `run_count` holds the cycle count for the currently tested distance. After each distance has completed its test,

if `run_count < min_count`, `min_count` is set to `run_count` and a 3-bit `best_dist` register is updated. After all distances are tested, `best_dist` indexes a table of 2-bit confidence counters and increments the count for that distance. DS repeats this process until any of the distance's confidence reaches to a threshold which is 2 in our implementation. Using a threshold less then 2 decreases the performance of some applications due to aggressive decisions. Using a threshold above 2 proved useless and increases the duration of the testing phase which also decreases the performance. Once the decision is made, the chosen distance is set to be used in the acting period and the testing period cycle counters are set to 0.
The ideal warm-up and measurement period may vary based on the previously evaluated distance and the current distance. We can use shorter warmup periods if the current and previous distances are shorter and longer warmup periods if either of them is longer. But in our evaluations, we observed that employing a mechanism to adjust the warmup time does not provide any performance benefit. Our experiments showed that 32 iterations of warmup and 32 iterations of testing period was sufficiently good to find the best distance for all benchmarks.

In multi-core architectures, distance selection is performed separately on each core. We observe that best distances vary for each core running a multi-threaded application due to the sharing of last-level cache and memory bandwidth.

2.3 Extending ATP for Prefetching Linked Data Structures

ATP is very successful in prefetching for indirect access structures as we show in the results section. However, for one of our workloads, HJ8, a significant performance opportunity was lost because indirect accesses were followed by linked list traversals and ATP was not able to capture this behavior. In HJ8, each element of the destination array is a linked list data structure. We extended the ATP to support linked lists. We add three additional fields in the AT: a node bit, node offset and number of nodes. An additional instruction, called `atnod` is also added to inform the hardware of this behavior and set the fields in the AT. This instruction always follows an ATAR instruction and have two operands: an offset that represent the next field in a linked-node and an immediate value that represents the number of nodes. For example, for a `A[B[i]]->next->next` structure, where each element of array `A` is a head node of a linked list where each list has 3 nodes (a head node and two additional nodes), generated ATI sequence is as follows:

```
atar PC_B
atar PC_A
atnod 0x18, 2
atrl PC_B, PC_A, 0
```

The `atnod` instruction will set the node bit, node offset and number of nodes fields of the AT table. In this example, `atnod` suggests that each element of array `A` points to the head node of a linked list. `atnod`'s first operand (offset) shows that `A[B[i]]+0x18` points to the next node of the linked list (`A[B[i]]->next`). The second operand (which is 2 in this example) shows the number of nodes except for the head node in linked list. Although ATP can stop prefetching nodes if it reaches to the end of the list (where the next node pointer is set to NULL), it requires this information to calculate the depth of prefetches in the chain. When the prefetch is triggered by an access to the index array, `B[i]`, ATP uses the same approach that it does for indirect memory accesses to prefetch future linked node accesses. The steps of prefetch address calculation in this case are shown in Table 7.

For an `A[B[i]]->next->next` structure, we prefetch the values `A[B[i+distance]]->next->next`, `A[B[i+distance*2]]->next`, `A[B[i+distance*3]]`, and `B[i+distance*4]` simultaneously. In this case, if `A[B[i+distance*2]]` is not in the cache (due to a late prefetch, etc.) ATP can drop the prefetch for array `A[B[i+distance*2]]->next` but still can issue the rest of the prefetches. Instead of aggressively prefetching for each level, it drops if the prefetches are not timely accurate and try to set the distance accordingly. Although using this method requires redundant calculations, we perform the address calculations of each level simultaneously by using a separate ALU for each PCB entry instead of having a separate mechanism to keep track of all incoming prefetched data and scheduling future

prefetches. This mechanism might be useful for an event-based system like ETP but in our evaluations, we observed that the pipelined prefetching method performs better with ATP on the benchmarks we used.

ATP can calculate prefetches for longer linked lists as but if the linked list is too long, it is difficult to find an optimal distance value which the prefetches for the index array will not be early and the prefetches for the last node of the linked list will not be too late. This is also an issue with ETP, however.

We observe significant performance improvement in HJ8 with an ATP that supports linked list traversals. Without this support, ATP achieves a speedup of 1.44 for this benchmark by successfully prefetching the indirect accesses. With the added linked list support, ATP boosts the speedup to 3.32 (2.3x speedup over ATP without linked-list support). By contrast, software prefetching, SWPF, that prefetch for both indirect accesses and the linked list in HJ8 could only achieve a speedup of 1.65, which is much lower than ATP's performance. It is important to note, however, that our implementation is limited to cases where number of nodes are known. HJ8 has 3 nodes for all the linked lists so prefetch depth is constant (e.g., depth is 4 in the example in Table 7) and ATP does not need to predict depth (distance, however, is dynamically evaluated as before). When the number of nodes is not known, ATP must make predictions for depth of the structure, which complicates the hardware mechanism. The situation is further complicated because unless searching of keys in hash-join buckets are pre-executed to determine if the next node has to be prefetched, many unnecessary prefetches are to be issued, which is also the case for ETP. This scenario is not evaluated and left as future work since our focus in this paper is indirect access structures.

Table 7: Prefetch calculation steps of `A[B[i]]` structure where each element of array A is a linked list.

Path	Step 1	Step 2	Step 3	Step 4
B	**Prefetch** B[i+4*dist]			
B → A	Read B[i+3*dist]	**Prefetch** A[B[i+3*dist]		
B → A	Read B[i+2*dist]	Read A[B[i+2*dist]	**Prefetch** A[B[i+2*dist]->next	
B → A	Read B[i+1*dist]	Read A[B[i+1*dist]	Read A[B[i+1*dist]->next	**Prefetch** A[B[i+1*dist]->next->next

3 Experimental Setup

We now discuss details of the simulation infrastructure, the workloads and the configurations that we used for our evaluation.

3.1 Simulation Environment

We implemented the ATP on the gem5 simulator [46] using System call Emulation (SE) mode (which does not simulate page-table walk) and generated the results using the x86 out-of-order CPU model. The SE mode only runs statically linked binaries because it emulates system calls without Operating System. TLB faults and misses are also handled by software (via emulated system calls), so all the detailed overhead related to the TLB are removed from the critical path of execution of the target machine and binary.

Table 8 shows the configuration of each core while Table 9 shows the ATP configuration. We inserted ATI instructions at the beginning of the loop and implemented ATP to prefetch for the L1 cache in order to provide for a direct comparison with IMP [3], which prefetches for the L1 cache. Each L1 is equipped with an 8-enty prefetch request queue (PRQ) in our evaluation. In all methods tested, computed prefetch addresses are placed into the PRQ before they are issued to the L1 cache.

As we described in Section 2.2.2.C, the prefetch address calculation depends on reading source array values from the L1 cache. In our evaluation, we assume that the ATP has dedicated ports to access the data TLB and the L1 data cache. If in any of the prefetch address calculation steps, the source values cannot be read due to a data TLB or an L1 data cache miss, the process of prefetch calculation and issuing is aborted until another access to the trigger array occurs and PCU is notified for initialization.

Table 8: Simulator Configurations

ISA	64-bit x86
Number of Cores	1-8
Architecture	4-Issue, Out-Order, 2GHz
LQ/SQ Entries	64/36
ROB Entries	168
Branch Pred.	Tournament BP
L1 Cache	Private, 8-way 32KB, MSHRs: 8, latency: 4 cycles
L2 Cache	Private, 8-way 256KB, MSHRs: 16, latency: 12 cycles
L3 Cache	Shared, 16-way, 1MB per core, MSHRs: 16, latency: 32 cycles
Memory	DDR3-1600, 800MHz, 1 channel, 2 ranks, 8 banks/rank, 512K rows/bank, 1kB row, baseline t_{RCD}/t_{RAS}/t_{WR}: 13.75/35/15 ns
Memory Controller	64-entry RD/WR request queue, FR-FCFS scheduling, closed-row policy

We faithfully implemented IMP (attached to each L1 cache) on our baseline architecture. Similar as in [3], our IMP implementation used a 16-entry Prefetch Table and a 4-entry Indirect Pattern Detector with 4-base address length and 4 shift values. Total hardware budget for our IMP implementation was 8032 bits (1004 bytes) per core, almost four times the size of the ATP (see Table 3). To evaluate the performance of software prefetching, we inserted software prefetching instructions inside the loops containing the indirect memory accesses. For both IMP and software prefetching, we measured the speedup for various prefetch distances but only report the results for the best performing distance.

For each benchmark, we fast-forward to the beginning of the loop containing indirect memory accesses and simulate 100M instructions after warming up for 10M instructions; for multi-core simulations, each core simulates at least 100M instructions. For the multi-core simulations, we run the multi-threaded version of the benchmarks where the number of threads is equal to the number of cores. In our simulations, we did not use warm-up, however, we expect the simulation results would not be significantly affected by it.

We use the number of cycles per loop-iteration as the performance metric as it eliminates additional overhead due to software prefetching. As such, it provides an apples-to-apples comparison between hardware and software prefetching.

Table 9: Hardware budget of ATP on single-core architecture

	ATI Queue	Array Table	Relation Table	Operation Table	Prefetch Table	Stream Register	Size Calc. Reg	Base Calc. Reg.	Distance Selector
# of Entries	4	4	4	2	4	1	1	1	-
Entry size (bits)	80	268	6	39	69	101	172	105	118
Total Size: 2266 bits (~284 bytes)									

3.2 Benchmarks

We used seven benchmarks to evaluate the performance of ATP. Each benchmark contains indirect memory accesses inside their performance critical loop. *Integer Sort* (*IS*) and *Conjugate Gradient* (*CG*) are from the NAS Parallel Benchmarks suite [4]. *IS* represents computational fluid dynamics programs and uses a bucket sort algorithm to sort integer values while *CG* represents unstructured grid computations and use eigenvalue estimation on sparse matrices. *IS* and *CG* have simple `A[B[i]]` access behavior.

Both *Pagerank* (*PR*) and *Triangle Counting* (*TC*) are from the CRONOSuite benchmark suite [6]. *PR* is a graph algorithm that ranks a website based on the rank of the websites that link to it [5] while *TC* counts the number of triangles in a graph and is used by graph algorithms such as clustering coefficients [7]. *PR* and *TC* have simple `A[B[i][j]]` access behavior.

Hash Join [8] hashes the keys stored in an array and uses the hashed values to access another array. Each bucket in the hash table consists of a linked list. We used two different variations of this benchmark: (1) *Hash Join 2EPB* (*HJ2*) has only one node per bucket and (2). *Hash Join 8EPB* (*HJ8*) has three nodes per bucket; as such it performs memory accesses for the additional nodes. *Hash Join* is a kernel representative of database applications.

Graph500 [9] (*g500*) runs a breadth first search (BFS) algorithm over a graph data structure. It performs indirect memory accesses while accessing neighbor vertices. *Histogram* (*Histo*) calculates the distribution of numerical data and is from the Parboil benchmark suite [10].

3.3 Evaluation Metrics

In our evaluations, we used throughputs of the experiments measured based on average cycles per iteration. All speedups are over the no-prefetching baseline. Although we could consider a baseline with a stride prefetcher, the performance results we observe with a stride prefetcher was similar to the results that we obtain with a no-prefetching baseline. The primary bottleneck of the workloads that we use in this work is indirect memory accesses.

4 Results

This section presents the performance of ATP, software prefetching (SWPF), and IMP, which is a pure hardware prefetching mechanism. We measure the performance of ATP, SWPF, and IMP for single and multi-core architectures. **It is important to note that the results are biased in favor of SWPF because** we presented the best speedup achieved by SWPF after carefully inserting prefetches and many profiling runs to obtain the best performing prefetch distances.

4.1 Single-Core Performance of ATP

Figure 7 shows the speedup of ATP, SWPF, and IMP over the no-prefetching baseline architecture. The average (geometric mean) speedup of ATP is 2.17 which outperforms both SWPF (1.84) and IMP (1.32). For *IS*, SWPF and ATP both outperform IMP, while ATP and IMP outperform SPWF for *CG*. As described in Section 1, the overhead due to SWPF was extremely high for both *IS* and *CG*. This overhead has little effect in *IS* because SWPF can hide this overhead by virtue for significantly reducing the latencies of the indirect memory accesses.

CG is one of the most sensitive benchmarks to the instruction overhead of software prefetching. Using software prefetching for *CG* decreases its performance by 33% while hardware prefetching mechanisms can achieve better performance (1.40 and 1.60 for ATP and IMP respectively). For *CG*, ATP has a lower speedup compared to IMP. Our evaluations show that for the same fixed distance value, ATP and IMP have a similar result for *CG*. We did not include CG's best performing distance (32) as a candidate distance for DS. DS can capture the best distance of CG as long as that distance value is a candidate of DS. However, increasing the number of candidate distances tend to decrease the overall performance due to longer distance evaluation periods. Therefore, we decided to exclude distance 32 as a candidate distance from ATP to achieve a better overall performance.

For *PR* and *TC*, SWPF and ATP outperform IMP. More specifically, for *PR*, SWPF and ATP have speedups of 1.53 and 1.20, respectively, while IMP has a 1.07 speedup. For *TC*, ATP and SWPF have a speedup of 1.76 and 1.65, respectively, while IMP does not have any speedup. *PR* and *TC*'s `A[B[i][j]]` type of indirect memory accesses is challenging for IMP as prefetching for the outer loop (*i.e.*, a prefetch distance of `i+distance`) yields more speedup than prefetching for the inner loop (*i.e.*, `j+distance`). By contrast, both SWPF and ATP are able to adjust the prefetch distance to maximize speedup although the overhead associated with SWPF degrades its achievable performance. However, SWPF still outperforms ATP for the *PR* benchmark. Since prefetching for inner loop iterations does not have much potential for this benchmark, ATP is programmed to target prefetching for future outer loop iterations. But even with the shortest distance, ATP loses opportunities and drops some of the prefetches due to early prefetches. SWPF does not suffer from this issue since it uses regular load instructions to read the index values. Even if these accesses are L1 cache misses, the index values can be read by accessing lower level caches or the memory. This is the only case where stalls due to intermediate memory accesses of software prefetching can be an advantage over ATP. However, further improvement is possible for ATP – L1 replacement policy can be modified for keeping ATP's prefetched blocks until used and evict them immediately afterwards. In this paper, we did not study the effects of changing cache replacement policy on ATP's performance, and left it as future work.

For *HJ2* and *HJ8*, ATP and SWPF again outperform IMP. More specifically, for *HJ2*, ATP and SWPF have speedups of 5.57 and 5.40, respectively, while IMP provides effectively no speedup. For *HJ8*, ATP and SWPF have speedups of 3.32 and 1.67, respectively, while IMP again provides effectively no speedup. IMP provides effectively no speedup over the no-prefetching baseline because IMP has great difficulty detecting the `A[func(B[i])]` type of indirect memory accesses present in *HJ2* and *HJ8*. The speedups of ATP and SWPF in *HJ2* are higher than those in *HJ8* because the latter contains linked list accesses which increases the depth of the access structure. ATP has a significant advantage over SWPF here since SWPF adds more instruction overhead while ATP can prefetch the linked list nodes without any extra instruction penalty.

Finally, as shown in Figure 7, all methods perform similarly for *g500*. ATP and SWPF has a slightly better speedup (1.22 and 1.23 respectively), while IMP has a 1.18 speedup for this benchmark.

4.2 Multi-Core Performance of ATP

Figure 8 and Figure 9 show speedups due to ATP, SWPF, and IMP on 4-core and 8-core architectures, respectively. Overall, for 4-core architectures, ATP has the highest average speedup (1.85) followed by SWPF (1.60) and IMP (1.25). For 8-cores, speedups for ATP, SWPF, and IMP are 1.41, 1.30, and 1.07, respectively.

For *g500*, benefit from prefetching increases on multi-core architectures. SWPF slightly outperforms ATP for this benchmark because ATP loses prefetch opportunities due to dropped prefetches (as we discuss in Section 4.6). Generally, the speedups due to prefetching in 4 and 8-core architectures are lower than on a single-core architecture due to increased resource utilization. By way of example, for *IS*, because the main loop is not long enough to hide memory latency, the speedup in *IS* decreases due to an increased number of memory accesses and the concomitant increase in latency for those accesses. Therefore, while resource contention has some effect on the efficacy of each prefetching method, ATP and SWPF still provide significant speedup.

4.3 Efficacy of Adaptive Distance on ATP Speedup

As described above, the Distance Selection Unit allows ATP to dynamically adjust the prefetch distance for different applications and configurations. Figure 10 compares the speedup of ATP when using adaptive prefetch distance versus ATP with various fixed distances (2, 4, 8, 16, and 32). The results in Figure 10 show that dynamically adjusting the prefetching distance has an average speedup of 2.17 while the highest performing fixed distance (distance = 8) yields an average speedup of 1.88. Therefore, even though periodically testing each distance for a certain number of iterations to choose the best distance for the next period may slightly degrade the speedup for some benchmarks, on average, dynamically adjusting the prefetch distance yields a higher average speedup.

Figure 10 shows that larger prefetch distances are more effective for benchmarks with simple two-level indirect accesses, such as *Histo*, *IS* and *CG*. For those benchmarks, optimal prefetch distance is similar (e.g., 32). However, *HJ8* and *G500* require more careful adjustments since they have more than two levels of accesses in the chain. For example, in *HJ8*, the values of the index array must be prefetched with the highest distance possible to allow the last node of the linked list to be prefetched with sufficient distance preventing a late prefetch.

IS and *HJ2* benefit from longer prefetch distances due to the small number of instructions in its main loop. As such, in order to be timely, prefetch instructions must be issued further away in order to be timely.

By contrast, *PR* and *TC* have higher performance when using shorter prefetch distances as Figure 10 shows. The indirect memory access behavior in these benchmarks is of the form `A[B[i][j]]`. Because the second dimension of array `B`, *i.e.*, j, is very short (16 for *PR* and 4 for *TC* for the inputs we used), ATP calculates prefetch addresses based on the first dimension instead. This increases the number of instructions executed between two consecutive accesses to the first dimension of array `B` which favors using shorter distances. As it is shown in our Figure 10, ATP can dynamically find the best performing distance which is near optimal for every level.

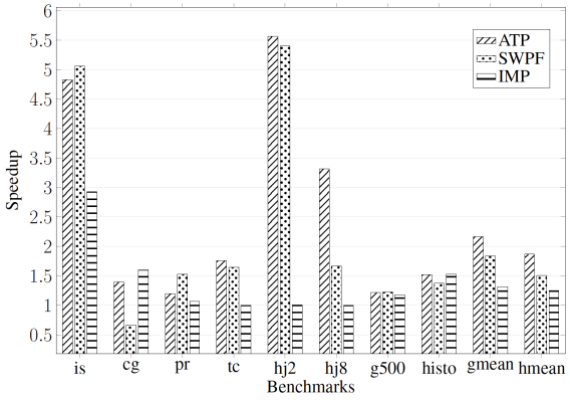


Figure 7: Performance comparison of SWPF, IMP, and ATP on single core architecture.

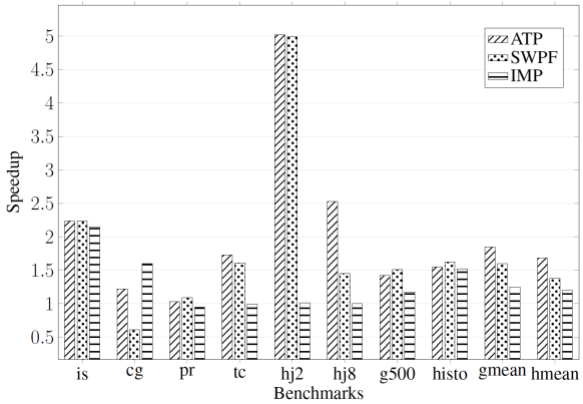


Figure 8: Performance comparison of SWPF, IMP, and ATP on 4-core architecture. Baseline is 4-core architecture with no prefetching.

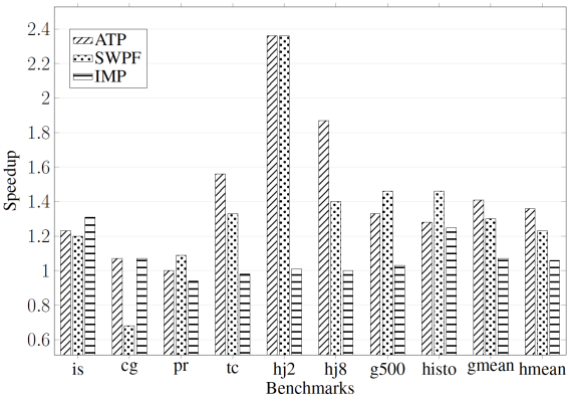


Figure 9: Performance comparison of SWPF, IMP, and ATP on 8-core architecture. Baseline is 8-core architecture with no prefetching.

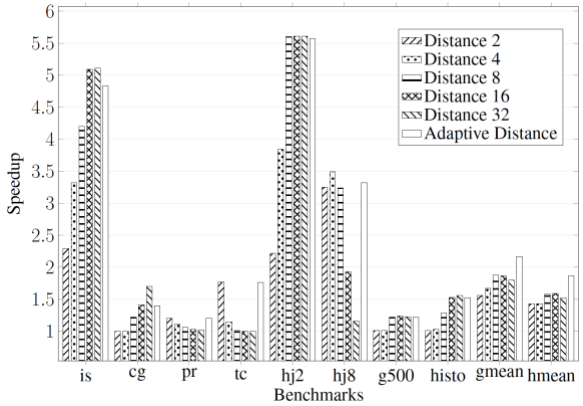


Figure 10: Performance comparison of ATP using different fixed distances and adaptive distance adjustment.

4.4 Prefetch Coverage and Accuracy

A prefetcher needs to be accurate or it will prefetch memory blocks that are never used, thus polluting its own cache. If a prefetcher is not timely, it will either not fully hide the memory latency of the cache miss or, even worse, the prefetched cache line will be evicted. Figure 11, Figure 12, and Figure 13 show the accuracy, timeliness and coverage, respectively, for different benchmarks using SWPF, IMP, and ATP. Accuracy is the percentage of prefetched cache lines which are demand-accessed later. Timeliness is the ratio of cache hits to previously prefetched cache lines to the total number of accurately prefetched cache lines. Coverage is computed as the ratio of demand accesses which are previously prefetched to the total number of demand accesses.

SWPF has 100% accuracy for all benchmarks because it handles border checks and mostly prefetches the correct cache lines, which are likely to be accessed in the near future. The prefetch accuracy of ATP and IMP are lower (99% and 80%, respectively). ATP is more accurate than IMP since the software mechanism specifies and limits the prefetches, thus reducing the number of useless prefetches.

We chose the best fixed distances for SWPF and IMP in our evaluations. The average timeliness for SWPF and IMP is 70% and 74%, respectively. By contrast, ATP dynamically adjusts the distance; the overall timeliness of ATP is significantly higher (88%). Although SWPF has the best coverage with 80%, ATP also has a high coverage with 73% since it calculates the prefetch addresses based on software hints. IMP fails to cover most of the potential prefetches as discussed in Section 1 (because it covers relatively small number of indirect access patterns) and therefore it has much lower coverage (19%).

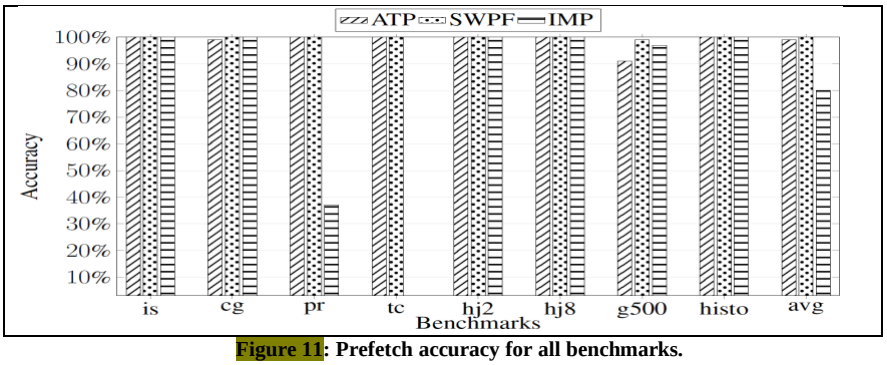


Figure 11: Prefetch accuracy for all benchmarks.

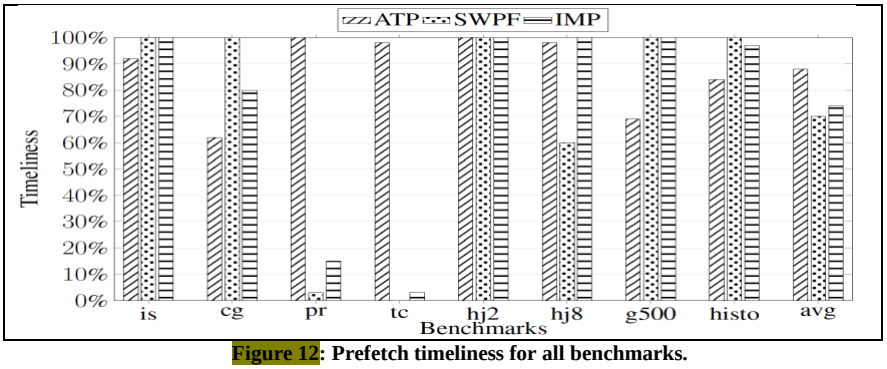


Figure 12: Prefetch timeliness for all benchmarks.

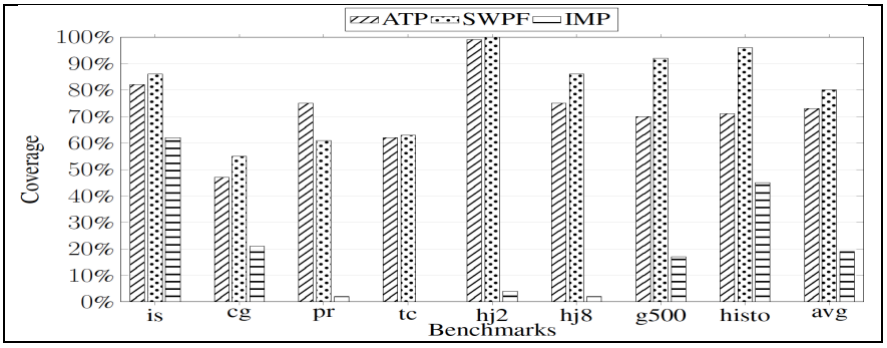


Figure 13: Prefetch coverage for all benchmarks.

4.5 Effects of Number of MSHRs, L1 Cache Size, and L1 Cache Access Latency

In this section, we evaluated the performance of ATP, SWPF, and IMP as we vary the number of MSHRs, L1 cache size, and L1 cache access latency. Figure 14 and Figure 15 show the effect of varying MSHRs on the performance of 1-core and 4-core architecture, respectively.

Figure 14 shows the overall results using 4, 8, and 16 MSHRs in L1 Cache in the single-core architecture. Increasing number of MSHRs from 4 to 8 and 16 improves the performance of no prefetching baseline by 13% and 16%, respectively (results not shown). Prefetching benefits from higher number of MSHRs more significantly than the no-prefetching baseline. As shown in Figure 14, the number of MSHRs have a big impact on all prefetching methods. The performance of IMP increases by 27% and 40% with 8 and 16 MSHRs, respectively, compared to its performance with 4 MSHRs. SWPF performs 32% better with 8 MSHRs and 57% better with 16 MSHRs. ATP benefits the most from higher number of MSHRs – 42% better with 8 MSHRs and %59 better with 16 MSHRs, compared to 4 MSHRs. For 4-core architecture, as shown in Figure 15, the number of MSHRs also has a big impact. Increasing the number of MSHRs from 4 to 8 shows improvements similar to the single-core case. However, 16 MSHRs does not have a significant benefit over 8 MSHRs since prefetching is already limited by higher utilization of shared resources.

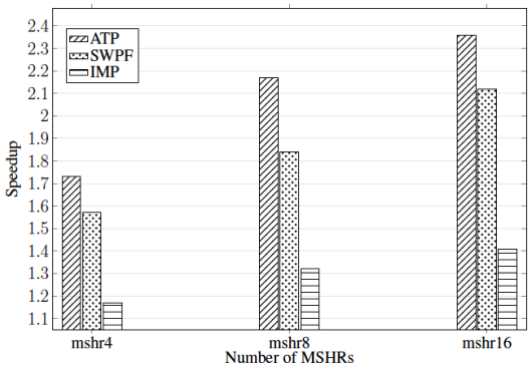


Figure 14: Effect of number of MSHRs in single-core architecture.

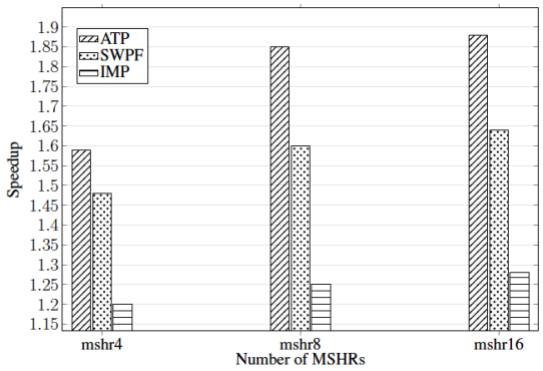


Figure 15: Effect of MSHRs in 4-core architecture.

As prefetching improves performance of demand execution, the frequency of observed index array accesses also increases. This in turn cause ATP to be triggered more frequently for prefetching and place more pressure on cache to service increasing number of simultaneous misses caused by prefetches. Therefore, for ATP and most prior hardware prefetchers, number of MSHRs has critical impact on performance. This impact is more severe for recently proposed ETP prefetcher since it is triggered not only by demand accesses but also on prefetched data. Although responding to prefetched data allows ETP to runahead of the demand execution significantly given sufficient number of MSHRs, this will cause many early prefetches to be present in cache and likely to be evicted before being used. For this reason, for ATP we decided not to react to prefetched data but rather use only demand access triggers. Not only does this reduce the pressure on the MSHRs but also keeps a fixed, yet adjustable, forward distance with demand execution in order to reduce the early prefetches.

Figure 16 and Figure 17 show the overall speedups of all prefetching methods using different L1 cache sizes on single- and multi-core architectures, respectively. The size of L1 cache has a very limited effect on the speedups either in no prefetching or prefetching methods. Also, both single-core and multi-core results show that, using prefetching on a smaller L1 cache size has much better performance compared to using a larger L1 cache size without prefetching. In all cache sizes that we evaluated, ATP outperforms SWPF and IMP.

Figure 18 and Figure 19 show the effect of L1 data cache access latency on the performance of the prefetching methods that we evaluated. Figure 18 shows the effect of three different access latencies on the single-core architecture and Figure 19 shows the same results for the multi-core architecture. Smaller cache latencies have a positive impact on all methods as

expected. However, the benefit of smaller latencies on prefetching methods reduce on multi-cores due to higher utilization of shared resources.

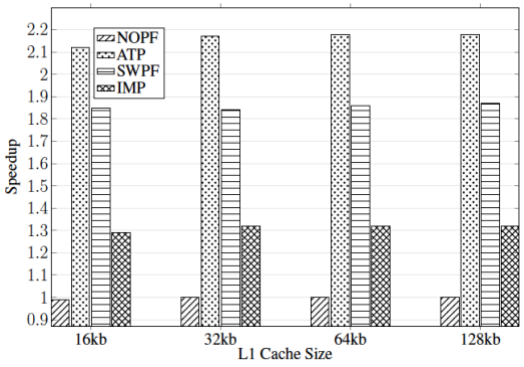


Figure 16: Effect of L1 Cache size in single-core architecture. Baseline is no prefetching with 32KB L1 cache.

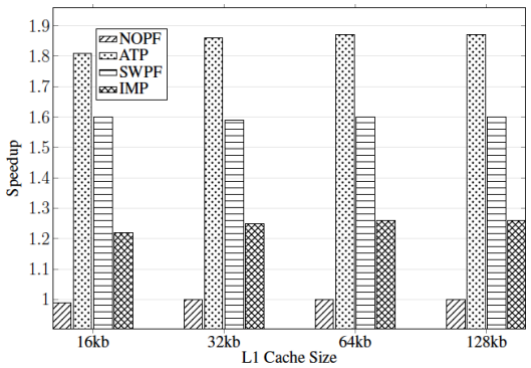


Figure 17: Effect of L1 Cache size in 4-core architecture. Baseline is no prefetching with 32KB L1 cache.

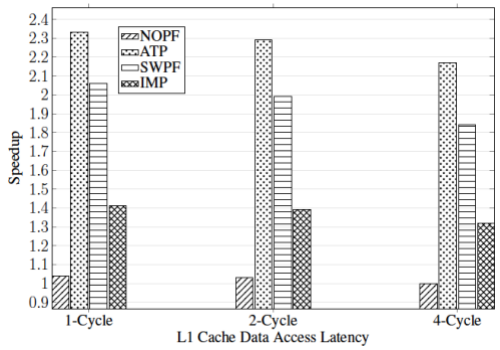


Figure 18: Effect of L1 Cache Data Access Latency in single-core architecture. Baseline for speedups is no prefetching with 4-cycle L1 cache latency.

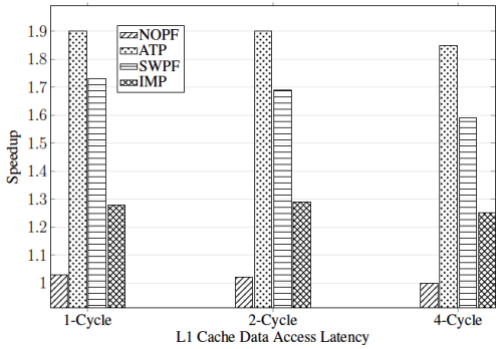


Figure 19: Effect of L1 Cache Data Access Latency in 4-core architecture. Baseline for speedups is no prefetching with 4-cycle L1 cache latency.

4.6 Prefetch Drops

Prefetch address calculation for indirect memory accesses requires the data of the source arrays to be present in cache at the time of calculation. Failing to read the required data will cause the indirect prefetch address calculation to be dropped. Prefetching the values of index arrays accurately and on-time is critical for indirect prefetching.

Figure 20 shows the percentage of the dropped indirect prefetches compared to the total number of prefetches that were expected to be calculated (does not include the prefetches for index arrays). We observe that most of the dropped prefetches in benchmarks *is*, *cg*, *g500*, and *histo* are due to late prefetching of index values. In *g500*, some of the prefetches are dropped because their dependent index values were not prefetched due to border conditions of the loop. Prefetch drops is most significant for *histo* followed by *g500*. Prefetch drops become more significant, especially as the number of cores in the system increases which impacts ATP's performance potential. As Figure 8 and Figure 9 show, for 4 and 8-core configurations, SWPF outperforms ATP. In *pr* and *tc*, early prefetching of the index values causes indirect prefetch address calculations to fail. These benchmarks are issuing prefetches for the outer loop iteration, which means that we may see many memory accesses during the execution of the inner loop. This can cause the prefetched index values to be evicted from the cache.

Prefetch drops can be eliminated completely if the prefetch triggering is done when source data is placed in the cache, similar to the operation of recently proposed ETP. However, this requires significant changes in the ATP and it may have other implications on performance as we discussed in Section 4.5. We left evaluating this method as future work.

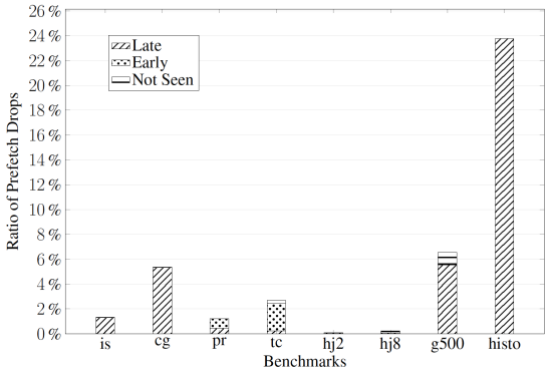


Figure 20: Ratio of dropped prefetches due to missing data required for indirect address calculation.

5 Related Work

Data prefetching [12] is a well-known technique to help alleviate the memory wall problem [21] by increasing Memory-Level Parallelism (MLP) [22, 23]. Many general-purpose microprocessors rely on data prefetching to improve performance for memory-intensive workloads. Most of the early prefetchers [11, 24, 25, 26] were based on sequential prefetchers, which prefetch sequential memory blocks relying on the fact that many applications exhibit spatial locality. Although sequential prefetchers work effectively in many cases, applications with non-sequential data access patterns do not benefit from sequential prefetching. That motivated the research on more complex prefetchers that try to capture the non-sequential nature of these applications [27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45].

Table 10 summarizes a variety of data access patterns and software and hardware prefetching methods targeting them. Prefetching techniques targeting pointer-based applications have been studied in [27, 28, 29, 41, 43, 45]. Indirect array references cannot be captured with those methods, however, since the desired addresses are computed, not contained in the memory as pointers. Guided Region Prefetching (GRP) [45] is a hardware prefetching scheme which uses compiler hints encoded in load instructions to regulate an aggressive hardware prefetching engine. GRP targets a broad range of behaviors from arrays and pointers to basic indirect arrays and recursive pointers. Indirect hints in GRP was limited to indirect array references of the form A[B[i]] and assumes fixed array element sizes, thus missing significant performance potential.

Table 10. Summary of Prefetching Methods for Basic Data Structures (+: full support; +/-: limited support; -: no support)

Basic Data Structure		Example Code	Pattern	Prefetching Method		
				Software	Hardware	ATP
Array	basic	A[i]	Stream	A[i+D]	stream [25]	+/-
	basic, pointer	A[i], A[i][constant]	Stride	A[i+D], A[i+D][constant]	stride [24], ghb [14], GRP [45]	+/-
	Indirect	A[B[i]], A[B[i][j]]	Irregular	A[B[i+D]], A[B[i][j+D]] Ainsworth and Jones (SWPF) [1]	GRP, CDP [42], Markov [13], IMP [3]	+
	Indirect+pointer	A[B[i]]->p	Irregular	A[B[i+D]]->p SWPF		+
Hash	basic	A[func(i)], A[func(B[i])]	Irregular	push-pull [29], spaid [19], precomputation [39]	precomputation [40], [44]	+
	basic+pointer	A[func(i)]->p	Irregular	push-pull, spaid, precomputation	Precomputation	+
Linked Data Structures		p = p->next	Irregular	p->next->next, Luk and Mowry [18]	GRP, CDP, Markov, Pointer cache [41], jump pointer [43]	+/-
Mixed		Complex	Irregular/ Regular	Helper threads [38], precomputation	precomputation, Memory streaming [32, 33, 34, 35, 37]	+/-

Helper Threads: Helper threads are separate threads which are used to prefetch future data accesses and they are useful for applications with irregular memory access patterns. Kim and Yeung [47] developed a compiler-based approach for creating helper threads which can capture irregular memory accesses without needing any extra hardware. Although it does not need any extra hardware, it is not power-efficient to run another thread on a high-performance core and it is difficult to keep them synchronized with the main thread. Lau et al. [48] proposed a small helper core and Ham et al. [49] provide a different scheme to create separate access and execute threads. Ganusov and Burtscher [50] proposed a lightweight architectural framework for efficient event-driven software emulation of complex hardware accelerators which can be applied to implement a variety of prefetching techniques. Although these methods can solve the synchronization issues, helper threads are unable to deal with stalling on intermediate loads created by the accesses of indirect memory access chains which makes it difficult for them to run ahead of the main thread.

Software Prefetching: Software prefetching [15, 16, 17, 18, 19, 20] provides a way for programmers to insert prefetching instructions into a program targeting various simple and complex patterns. Manual insertion is flexible but requires significant programmer effort. Automatic insertion requires compiler to recognize the access pattern. Ainsworth [1] developed an algorithm which automates the insertion of software prefetches for indirect memory accesses into programs. Although this approach eliminates the requirement for the programmer effort, it cannot guarantee to insert the instructions in an optimized way for the specific architecture. Furthermore, significant instruction overhead may offset its benefits. On the other hand, software prefetching can target more complex patterns than hardware counterparts, especially if hardware budget is limited. In contrast to prior work, we proposed a hybrid software-hardware approach using strengths of each for prefetching indirect memory accesses.

Finally, Lee et al. [20] studied the interaction between software and hardware prefetching and found that inserting software prefetching instructions in the presence of hardware prefetchers may hurt the overall prefetching performance due to the incorrect training of hardware prefetchers. ATP does not have this problem because prefetching is only initiated by hardware, not by software prefetch instructions; course-grain metadata instructions are used to guide the hardware prefetcher.

Recent Hardware Prefetchers for Irregular Accesses: More recently, Continuous Runahead Execution (CRE) [2] proposed a complex mechanism to dynamically identify address dependence chain of a load that is likely to create a cache miss. It can accurately prefetch data needed in near future. However, CRE cannot provide effective prefetching for indirect accesses because indirect accesses create load miss chains, which prevent CRE to run sufficiently ahead. Most relevant to our work is the study by Yu [3] which proposed a hardware mechanism targeting indirect accesses. Although it can successfully find many regular (`A[B[i]]`), multi-way (`A[B[i]]` and `C[B[i]]`), and multi-level (`A[B[C[i]]]`) structures, it struggles to detect more complex structures and it cannot perform software specific optimizations (e.g., prefetching for the future iterations of outer loop instead of inner loop).

Most sophisticated programmable prefetcher to date is the recently proposed method by Ainsworth and Jones [51]. They proposed an event-triggered programmable prefetcher (ETP), which can capture complex indirect access patterns but it relies on multiple RISC cores each of which should execute a subprogram to calculate a prefetch address for an event. This requires relatively expensive operations per event. ATP is custom hardware that can be configured with sufficient programmability for a variety of access patterns (initially ATP focuses on indirect accesses but in this paper, we also show how it can be extended for linked data structures). It is configured once as opposed to each time a trigger event is observed.

ATP is significantly different than Ainsworth and Jones' event-triggered model for prefetching indirect access chains. Their model needs to set the initial prefetch distance long enough that it would cover all the prefetches in the indirect access chain. For example, in an `A[B[C[i]]]` behavior, it first creates a prefetch for `C[i+distance]`, then when the data for `C[i+distance]` arrives at the cache, it issues a prefetch for `B[C[i+distance]]` and finally, when the data for `B[C[i+distance]]` arrives at the cache it issues a prefetch for `A[B[C[i+distance]]]`. In this case, it is critical to set the distance of C long enough that the prefetches for array A would not be late and prefetches for array C would not be early. In our method, we use a pipelined prefetch model where each array is prefetched individually. For an `A[B[C[i]]]` structure, we prefetch `A[B[C[i+distance]]]`, `B[C[i+distance*2]]`, and `C[i+distance*3]`. In this case, if `C[i+distance*2]` is not in the cache (due to a late prefetch, etc.) ATP can drop the prefetch for array B but still can issue the prefetches for array A and array C. Instead of aggressively prefetching for each level, it drops if the prefetches are not

timely accurate and try to set the distance accordingly. Initially, we designed ATP that used an event-triggered methodology but later abandoned it because pipelined prefetch model achieved better performance on our workloads.

6 Conclusion and Future Work

We propose and implement the Array Tracking Prefetcher to have the benefits of both software and hardware prefetching for indirect memory accesses. ATP inserts prefetch metadata instructions outside the loop and use them to pass information to the hardware mechanism. The hardware mechanism uses this information to determine which indirect memory accesses to prefetch and when to do so. To increase the prefetch timeliness (and performance), ATP dynamically adjusts the distance. By using software hints, ATP avoids using an expensive hardware budget.

Our results show that ATP yields an average speedup of 2.17X, 1.85X, and 1.41X for single-core, 4-core, and 8-core architectures, respectively. ATP also outperforms the state-of-the-art software-based (SWPF) and hardware-based (IMP) prefetching methods.

In future work, we plan to improve ATP's capacity by targeting diverse data structures. In this paper, we showed that ATP can be extended to target linked-list traversals. However, our extension was based on a specific case where the number of the nodes was known. In the future, we plan to improve our ATP software/hardware interface to enable prefetching for more complex data structure traversals. In addition, currently, ATP misses significant opportunity by dropping prefetches when source data for address calculation is not present in the cache at the time of calculation. In future work, we plan to modify ATP so that prefetch address calculations can be triggered by cache fills from source data due to prefetched index array.

References

[1] Ainsworth, Sam, and Timothy M. Jones. 2017. Software prefetching for indirect memory accesses. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 305-317.

[2] Hashemi, Milad, Onur Mutlu, and Yale N. Patt. 2016. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 61.

[3] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: indirect memory prefetcher. In Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48). ACM, New York, NY, USA, 178-190.

[4] Bailey, David H. 2011. NAS parallel benchmarks. *Encyclopedia of Parallel Computing*: 1254-1259.

[5] Page, Lawrence, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web*. Stanford InfoLab.

[6] Ahmad, Masab, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. 2015. Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, 44-55.

[7] Chiba, Norishige, and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM Journal on computing* 14, no. 1 (1985): 210-223.

[8] Balkesen, Cagri, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 362-373.

[9] Murphy, Richard C., Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19: 45-74.

[10] Stratton, John A., Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W. Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127.

[11] Chen, Tien-Fu, and Jean-Loup Baer. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers* 44, no. 5: 609-623.

[12] Vanderwiel, Steven P., and David J. Lilja. 2000. Data prefetch mechanisms. ACM Computing Surveys (CSUR) 32, no. 2: 174-199.

[13] J D. Joseph and D. Grunwald. 1997. Prefetching using Markov Predictors. In Proc. 24th Int'l Symp. Comp. Arch., ISCA, 252-263.

[14] K. J. Nesbit and J. E. Smith. 2004. Data cache prefetching using a global history buffer. In Proceedings of the 10th International Symposium on High Performance Computer Architecture. IEEE Computer Society, Los Alamitos, CA, 96-105.

[15] Callahan, David, Ken Kennedy, and Allan Porterfield. 1991. Software prefetching. In *ACM SIGARCH Computer Architecture News*, vol. 19, no. 2: 40-52.

[16] Mowry, Todd C. 1994. Tolerating latency through software-controlled data prefetching. PhD diss., to the Department of Electrical Engineering.Stanford University.

[17] Wu, Youfeng, Mauricio Serrano, Rakesh Krishnaiyer, Wei Li, and Jesse Fang. 2002. Value-profile guided stride prefetching for irregular code. In *International Conference on Compiler Construction*. Springer, Berlin, Heidelberg, 307-324.

[18] Luk, Chi-Keung, and Todd C. Mowry. 1996. Compiler-based prefetching for recursive data structures. In *ACM SIGOPS Operating Systems Review*, vol. 30, no. 5: 222-233.

[19] M. Lipasti, W. Schmidt, S. Kunkel and R Roediger. 1995. SPAID: Software prefetching in pointer- and call-intensive environments. In Proceedings of the 28th International Symposium on Microarchitecture. IEEE Computer Society Press, Los Alamitos, CA, 232-236.

[20] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. ACM Trans. Archit. Code Optim. 9, 1, Article 2 (March 2012).

[21] Wulf, Wm A., and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, no. 1: 20-24.

[22] Chou, Yuan. 2007. Low-cost epoch-based correlation prefetching for commercial applications. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 301-313.

[23] Wenisch, Thomas F., Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. 2005. Temporal streaming of shared memory. *ACM SIGARCH Computer Architecture News* 33, no. 2: 222-233.

[24] Baer, Jean-Loup, and Tien-Fu Chen. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing'91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. IEEE, 176-186.

[25] Jouppi, Norman P. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI: 364-373.

[26] Palacharla, Subbarao, and Richard E. Kessler. 1994. Evaluating stream buffers as a secondary cache replacement. In *ACM SIGARCH Computer Architecture News*, vol. 22, no. 2. IEEE Computer Society Press, 24-33.

[27] Ebrahimi, Eiman, Onur Mutlu, and Yale N. Patt. 2009. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 7-17.

[28] Roth, Amir, Andeas Moshovos, and Gurindar S. Sohi. 1998. Dependence based prefetching for linked data structures. *ACM SIGOPS Operating Systems Review* 32, no. 5: 115-126.

[29] Yang, Chia-Lin, and Alvin R. Lebeck. 2000. Push vs. pull: Data movement for linked data structures. In *Proceedings of the 14th international conference on Supercomputing*. ACM, 176-186.

[30] Emma, Philip G., Allan Hartstein, Thomas R. Puzak, and Viji Srinivasan. 2005. Exploring the limits of prefetching. *IBM Journal of Research and Development* 49, no. 1: 127-144.

[31] Srinivasan, Viji, Edward S. Davidson, and Gary S. Tyson. 2004. A prefetch taxonomy. *IEEE Transactions on Computers* 53, no. 2: 126-140.

[32] Somogyi, Stephen, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. 2009. Spatio-temporal memory streaming. In *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3: 69-80.

[33] Wenisch, Thomas F., Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2009. Practical off-chip meta-data for temporal memory streaming. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 79-90.

[34] Ferdman, Michael, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. "Temporal instruction fetch streaming." In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 1-10.

[35] Wenisch, Thomas F., Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal streams in commercial server applications. In *2008 IEEE International Symposium on Workload Characterization*. IEEE, 99-108.

[36] Wenisch, Thomas F., Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2007. Mechanisms for store-wait-free multiprocessors. *ACM SIGARCH Computer Architecture News* 35, no. 2: 266-277.

[37] Somogyi, Stephen, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2006. Spatial memory streaming. In *33rd International Symposium on Computer Architecture (ISCA'06)*. IEEE, 252-263.

[38] Solihin, Yan, Jaejin Lee, and Josep Torrellas. 2002. Using a user-level memory thread for correlation prefetching. In *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 171-182.

[39] Luk, Chi-Keung. 2001. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 40-51.

[40] J. Collins, D. Tullsen, H. Wand and J. Shen. 2001. Dynamic speculative precomputation. In Proceedings of the 34th International Symposium on Microarchitecture. IEEE Computer Society, Los Alamitos, CA, 306-317.

[41] J. Collins, S. Sair, B. Calder and D. Tullsen. 2002. Pointer cache assisted prefetching. In Proceedings of the 35th International Symposium on Microarchitecture. IEEE Computer Society Press, Los Alamitos, CA, 62-73.

[42] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. 2002. A stateless, content-directed data prefetching mechanism. In Proceedings of the 10th international conference on Architectural support for programming languages and operating systems (ASPLOS X). ACM, New York, NY, USA, 279-290.

[43] Amir Roth and Gurindar S. Sohi. 1999. Effective jump-pointer prefetching for linked data structures. In Proceedings of the 26th annual international symposium on Computer architecture (ISCA '99). IEEE Computer Society, Washington, DC, USA, 111-121.

[44] C. Zilles and G. Sohi. 2001. Execution-based prediction using speculative slices. In Proceedings of the 23rd International Symposium on Computer Architecture. ACM, New York, NY, 2-13.

[45] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. 2003. Guided region prefetching: a cooperative hardware/software approach. In Proceedings of the 30th annual international symposium on Computer architecture (ISCA '03). ACM, New York, NY, USA, 388-398

[46] Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S. and Sen, R., 2011. The gem5 simulator. ACM SIGARCH Computer Architecture News, 39(2), 1-7.

[47] Kim, Dongkeun, and Donald Yeung. 2002. Design and evaluation of compiler algorithms for pre-execution. In ACM SIGPLAN Notices, vol. 37, no. 10. ACM, 159-170.

[48] Lau, Eric, Jason E. Miller, Inseok Choi, Donald Yeung, Saman P. Amarasinghe, and Anant Agarwal. 2011. Multicore Performance Optimization Using Partner Cores. In *HotPar*. 2011.

[49] Ham, Tae Jun, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 191-203.

[50] Ganusov, Ilya, and Martin Burtscher. 2006. Efficient emulation of hardware prefetchers via event-driven helper threading. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, 144-153.

[51] Ainsworth, Sam, and Timothy M. Jones. 2018. An event-triggered programmable prefetcher for irregular workloads. In *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 578-592.