Array Tracking Prefetcher for Indirect Accesses

Mustafa Cavus
Department of Electrical, Computer, and Biomedical Engineering
University of Rhode Island
Kingston, RI, USA
mcavus@uri.edu

Resit Sendag
Department of Electrical, Computer, and Biomedical Engineering
University of Rhode Island
Kingston, RI, USA
sendag@uri.edu

Joshua J. Yi
Dechert LLP
Austin, TX, USA
joshua.yi@dechert.com

***Abstract*—Indirect memory accesses have irregular access patterns and concomitantly poor spatial locality. To address this problem, we propose the *Array Tracking Prefetcher* which tracks array-based indirect memory accesses using a novel combination of software and hardware. Our results show that ATP yields average speedup of 1.60 over the baseline single-core without prefetching. By contrast, the speedup for conventional software and hardware-based prefetching, is 1.49 and 1.16, respectively. For four-cores, the average speedups for ATP, software, and hardware are 1.49, 1.38, and 1.11, respectively.**

I. Background and Introduction

The execution of data structures such as sparse matrices and graphs frequently result 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]]`. A hardware stream prefetcher can easily prefetch entries of array B because its entries are sequentially accessed. By contrast, because there may be no pattern to the values stored in array B, accesses to array A are likely to be irregular, which obviates the efficacy of a stream prefetcher which depends on accesses having spatial locality.

Software prefetching can hide the memory latencies of indirect memory accesses. See*, e.g.,* [1] (describing a compiler-based system to generate software prefetches for indirect memory accesses). Software prefetching requires additional instructions, *e.g.*, the prefetching instructions themselves, instructions for address calculation and border checking, *etc.*

In addition to instruction overhead, the benefit of software prefetching is further limited by dependencies related to prefetch address calculation and lack of run-time information needed for optimal placing of prefetch instructions. For a set of memory-bound benchmarks (described in Section III) with indirect accesses, we observe that the effect of prefetching distance (*i.e.*, how far in advance of the memory access the prefetch instruction is issued) on software prefetching speedup is very significant. On average, resulting speedups vary from 1.14 (worst) to 1.49 (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 instruction.

Hardware prefetchers, by comparison, do not require executing additional instructions in order to compute and issue prefetches but, as described above, they can easily prefetch sequentially-accessed array entries. But, in order to capture irregular access patterns, hardware prefetchers generally require very complex mechanisms to be able to capture irregular access patterns. *See, e.g.*, Hashemi et al. [2] (describing continuous runahead execution). Yu et al. [3] proposed a relatively less complex 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]]]`).

Code snippets 1-3 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 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 distance. Even when `M` (the maximum number of iterations for the inner loop) is very small, IMP can still capture this indirect memory access and issue prefetches, but it may not be able to fully hide the memory latency of these indirect memory accesses because the distance between the prefetch instruction and the memory access is too small. But, in this case, increasing the distance actually results in a performance decrease 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]]
```

Code Snippet 1: Two-dimensional array in a nested loop.

Code snippet 2 depicts an indirect memory access that requires additional arithmetic/logical operations to compute the memory address. More specifically, the index to array A requires both a logical AND and a right-shift. This type of code appears in benchmarks such as *HashJoin ph2* (*hj2).*

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

Code Snippet 2: Index requires arithmetic/logical computations.

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

Code Snippet 3: Code requires simultaneous tracking of multiple indirect accesses of varying depth using the same index array.

Most hardware prefetchers such as IMP are unable to capture these memory accesses because they require more than one arithmetic/logical operations; only very 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 dependency chain can be successfully detected. But if this code snippet only required one operation, most hardware prefetchers could capture them by calculating a virtual base address or virtual element size.

Finally, when multiple indirect accesses of varying depth appear at the same time as in Code snippet 3, it complicates hardware tracking. We observe that IMP is not successful to capture the full behavior. IMP was able to detect and prefetch for `B[C[i]]` and `D[C[i]]`, but not for `A[B[C[i]]]`. However, IMP is able to track either `A[B[C[i]]]` or `D[C[i]]`, if they do not exist at the same time.

By contrast, software prefetching can accurately prefetch these memory accesses with some programmer effort. But this type of memory access has a significant overhead because requires performing the arithmetic/logical operations, which have a read-after-write dependence between them, for each prefetch. Furthermore, due to lack of run-time information, the best prefetching distance is hard to predict.

Because 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 individual indirect memory access types. To enable the compiler to extract indirect access information from a loop, ATP relies on the programmer to mark the corresponding loop. And rather than using software to insert prefetching instructions, ATP uses special instructions to pass hints the hardware mechanism. These special instructions only execute outside of the loop, so they do not result in significant instruction overhead. Providing hints to the hardware mechanism is better than a pure hardware mechanism because the hardware mechanism can configure itself based on the behavior of the software. This can significantly reduce the training time. Furthermore, it enables the hardware mechanism to effectively prefetch a wide variety of indirect access behaviors as it does not need to detect these behaviors itself.

During execution of the loop, ATP calculates size of the array type (*i.e.*, the prefetching stride) and base address of the required arrays, *e.g.*, Address of `A[0]` for a `A[B[i]]`. After calculating the strides and base addresses, ATP starts generating prefetch addresses for forthcoming indirect memory addresses based on the calculated base address and stride.

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. ATP selects the best distance after a period of cycles and uses the selected distance for the next period.

Across a set of memory-bound benchmarks, for a single-core architecture, ATP achieved an average speedup of 1.60X (with a maximum of 3.27X) over the no prefetching baseline. By comparison, (manually inserted) software prefetching had an average speedup of 1.49X while hardware prefetching (IMP) had an average speedup of 1.16X. For a 4-core architecture, ATP had an average speedup of 1.49X (up to 3.04X) while software prefetching and IMP had average speedups of 1.38X and 1.11X, respectively.

II. Array Tracking Prefetcher

ATP consists of a software and a hardware component. The software component is responsible to detect indirect access related information within a loop and pass it to hardware. The hardware component uses this information to initialize the ATP, which will in turn compute indirect prefetch addresses and issue prefetch requests at run-time.

A. Software Component

The software component of the ATP extracts information related to indirect memory accesses within a loop. This loop can either be marked by the programmer as shown in the Code snippet 4 or can be automatically identified using a compiler pass similar to the approach in [1]. The extracted indirect-access information is provided to the hardware through special instructions (e.g., see Code snippet 6), called *Array Tracking Instructions* (*ATIs*).

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

Code Snippet 4: Marking the loop for indirect prefetching

```
4005a0:       movslq   (%rax), %rcx
4005a3:       add  $0x4, %rax
4005a7:       add  (%rsp, %rcx, 4), %edx
4005aa:       cmp  %rsi, %rax
4005ad:       jne  4005a0
```

Code Snippet 5: Instructions inside the marked loop.

```
atar  $0x4005a0
atar  $0x4005a7
atrl  $0x4005a0, $0x4005a7, 0
```

Code Snippet 6: ATIs generated from graph in Figure 3

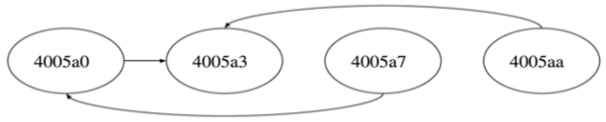

Figure 1: Dependency graph generated from the Code Snippet 5.

Array Tracking Instructions (ATIs): Each ATI is packed into a single *Array Tracking Execute* (*ATE*) instruction, which can be recognized in the processor pipeline. When a core identifies an ATE instruction, the core removes it from the pipeline and forwards it to the ATP. It is important to note that the number of executed ATE instructions is insignificant since they appear only once per the main loop where indirect access traversals occur. There are four types of ATIs: 1) `ATCL` (or `ATE x3`) clears all ATP tables. It does not have any operands. 2) `ATAR` (or `ATE x0`) inserts entries into ATP's Array Table (AT).

It has a single operand: the address[1] (PC) of the load instruction that accesses the base[2] or the target array involved in indirect accesses. 3) `ATRL` (or `ATE x1`) inserts relation (*e.g,* between target array `A` and base array `B` in an `A[B[i]]` structure) information to the ATP's Indirect Relation Table (IRT). It has three operands: the first two are the PCs (offsets) of the load instructions accessing the base and target arrays, respectively. The third operand (1-bit) specifies the type of the relation. 0 (zero) is used for regular `A[B[i]]` type accesses and 1 is used when the index array is a 2D array, such as `B` in `A[B[i][j]]`. Finally, 4) `ATOP` (or `ATE x2`) is used for complex structures when the indexes to the target array is computed by a function that uses the base array as an argument (e.g., `A[func(B[i])]`). `ATOP` inserts these operations to ATP's Operation Table (OT). `ATOP` has two operands. The first operand is the operation type (e.g, `NOT`, `ADD`, etc.). The second operand specifies the data for the operation. If there are more than one operation, more than one `ATOP` instruction are created. For example, `A[(B[i]-1)&0xF]`, requires two consecutive `ATOP` instructions. The first `ATOP` instruction always follows an `ATRL`. The first `ATOP` instruction's first operand is implicitly specified as the base array value (e.g., `B[i]`). The second `ATOP` uses the result of first `ATOP` as its first operand.

ATE instructions are 6-bytes long: Two bytes are reserved for the opcode, 2-bits specifies type of the ATI instruction, and the remainder are for the operands as discussed above.

Generating ATIs: Generation of ATIs consists of two stages. First, a dependency graph of the instructions inside the marked loop (in Code Snippet 5) is generated as shown in Figure 1 and then ATIs are generated based on this graph. Once the dependency graph is generated, a software pass visits all nodes to generate ATIs. For all nodes which belong to a load instruction, an `ATAR` instruction is generated. Then, `ATRL` instructions (and `ATOP` instructions if required) are generated based on the connections of the nodes. Code snippet 6 shows the ATIs generated from the graph in Figure 1. These instructions are placed above the entry point of the loop. Additionally, an

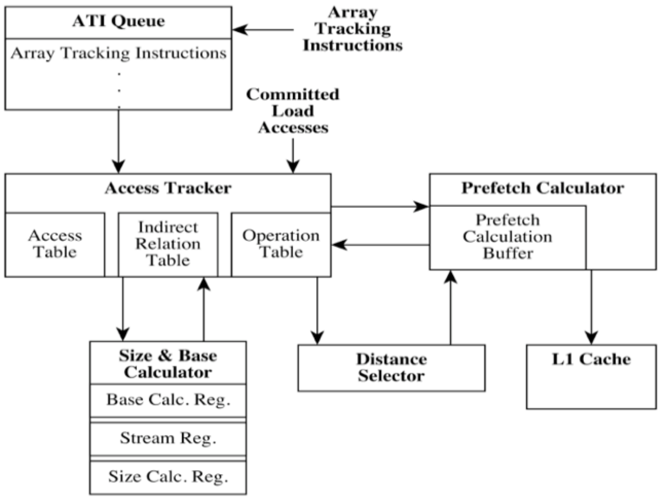

Figure 2: An overview of ATP

`ATCL` instruction will be inserted to the start of the loop (preceding other ATIs) to clear the ATP tables before the execution of the loop has completed.

B. Hardware Component

An overview of ATP hardware mechanism is shown in Figure 2. ATP consists of an ATI queue (ATQ), an Access Tracker Unit (ATU), a Base Calculator (BC), a Prefetch Calculator (PFC), and a Distance Selector (DS).

After an ATE instruction has been identified in the processor pipeline, it is forwarded to the ATP hardware. A sub-opcode field identifies individual ATI instructions. Each ATI instruction is inserted into the ATQ, which is a FIFO queue with head and tail pointers. ATQ is simply the interface between the processor pipeline and the ATP hardware. ATP processes ATI instructions in-order from the ATQ.

ATU and Processing of ATI instructions: Each valid ATI in the ATQ is processed in-order from ATQ's head to tail. In general, ATIs are used to initialize/program the ATU tables. 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 ATU tables, namely valid bits are set to zero in AT, IRT and OT tables. We explain how ATI instructions initialize or program ATU tables using the example in Figure 3, which shows the final status of AT, IRT and OT tables after they are initialized for `A[(B[i]&0x7F)*7]` 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 table. 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 is shown in Figure 3. 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 3, 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 3, AT's entry 1 for array `B` has a non-zero indirect map, 10, suggesting the 0th entry in the IRT table provides in its destination field a pointer to the target array (in the AT) for which array B 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.

Level 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

[1] 16-bit offset from ATAR's own address (PC) is sufficient.

[2] We use base and index array interchangeably to mean the same thing.

Array Table (AT)

valid	load pc	base bit	base addr	trigger bit	trigger type	size bit	size	indirect map	depth	root addr bit	root addr	root size
1	PC A	1	Base A	0		1	8	00	0	0		
1	PC B	0		1	1	1	4	10	1	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	7	0	

ATI instructions

```
atar    PC_A
atar    PC_B
atrl    PC_A, PC_B, 0
atop    fAND, 0x7f
atop    fMUL, 0x7
```

Figure 3: Array Table, Indirect Relation Table, and Operation Table are initialized for a A[(B[i]&0x7F)*7] structure where A is an array of 8-byte double values and B is an array of 4-byte integer values.

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 II.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 7 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 specify 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 have been processed, ATU has completed initialization and ATP is ready to move to the size and base calculation stage. It is important to note that with sequence of ATIs, ATU can keep track of multiple indirect array access structures with various levels of complexity, simultaneously, which is a significant improvement compared to the state-of-the-art indirect hardware prefetcher, IMP.

Size and Base Calculator Unit (SBCU): 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, if its PC is found in the AT table, and if the size bit is 0, SBCU starts the process for size computation. First, the stride between two accesses of the trigger array is computed. SBCU unit uses a stream register to keep track of trigger array accesses. A stream 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 stream 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, SBCU 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 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.

$$Size(A) = \frac{Addr(A[B[i+1]) - Addr(A[B[i]])}{B[i+1] - B[i]} \qquad (1)$$

Base Address Calculation: Finally, before indirect prefetching can be performed for target arrays, their base addresses must be computed. SBCU 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 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, SBCU monitors the accesses requested for any of the source arrays by checking its indirect map field. When SBCU is notified by an access of a source array, it stores the value of the accessed data in 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 SBCU, 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, SBCU sets the base address field of the corresponding entry in the AT and releases the BACR.

$$Addr(A[B[i]]) = \mathrm{B}aseAddr(A) + (B[i] \times Size(A)) \qquad (2)$$

C. *Prefetch Triggering and Calculation*

As shown in Figure 2, ATP employs a Prefetch Calculation Unit (PCU) to calculate prefetch addresses when triggered. Whenever ATU observes a trigger access, it notifies PCU with the index of corresponding AT entry to begin prefetch calculation process. Prefetch calculation in ATP is performed in two steps: 1) prefetch initialization and 2) prefetch address calculation and issuing of prefetches.

Due to space limitation, in this section, we only explain the PCU operation for a 3-level indirect access structure `A[B[C[i]]]`. As ATU signals PCU for prefetching operation on an access to the trigger array `C`, PCU triggers initialization phase and first allocates an entry for `C` in the Prefetch Calculation Buffer (PCB). PCB is a temporary buffer that keeps detailed information about the entries of potential prefetches and it is cleared after all computed prefetches are issued. 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.

After the initialization is done, PCU starts calculation of prefetch addresses for each entry. To calculate the prefetch address for any non-trigger array, PCU needs to read a value from the source array. In an `A[B[C[i]]]` structure, assuming we want to calculate a prefetch address for `A[B[C[i+distance]]]` triggered by an access to `C[i]`, we need to read the value of `C[i+distance]` and then `B[C[i+distance]]` to be able to calculate the prefetch address for `A[B[C[i+distance]]]`. To be able to find these values in the cache when needed, they need to be prefetched ahead (the source values). So for an `A[B[C[i]]]` structure, PCU follows the following steps for address calculation and to perform prefetching:

1. Compute the prefetch address for `C[i+3*distance]` (depth of `C` is 3 in this structure) using Equation 3 and **issue the prefetch**.
2. Calculate the prefetch address for `B[C[i+2*distance]]` (depth of `B` is 2 in this structure):
    a. **Compute** the address for `C[i+2*distance]` using Equation 3 and **read** its value from the L1 cache.
    b. **Compute** the prefetch address for `B[C[i+2*distance]]` using Equation 2 and **issue the prefetch**.
3. Calculate the prefetch address for `A[B[C[i+1*distance]]]` (depth of `A` is 1 in this structure):
    a. **Compute** the address for `C[i+1*distance]` using Equation 3 and **read** its value from the L1 cache.
    b. **Compute** the address for `B[C[i+1*distance]]` using Equation 2 and **read** its value from the L1 cache.
    c. **Compute** the prefetch address for `A[B[C[i+1*distance]]]` using Equation 2 and **issue the prefetch**.

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

Prefetch address calculation depends on reading source array values from the L1 cache. We assume ATP has dedicated ports to access the data TLB and the L1 cache. If in any of the steps above, the source values cannot be read due to a data TLB or an L1 cache miss, the process of prefetch calculation and issuing is aborted until another access to the trigger array occur and PCU is notified for initialization. The distance value is read from the Distance Selection Unit (DSU) as explained in Section II.D. 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. The computed prefetch addresses are placed into an 8-entry prefetch request queue (PRQ) before they are issued to the L1 cache.

PCU operation (both initialization and prefetch calculation) is similar to the above for `A[func(B[i])]` structure but different for a `A[B[i][j]]` structure, where trigger is a 2D array. Initialization for `A[B[i][j]]` is involved with the root address fields of the target arrays in the AT and prefetch addresses are computed using Equation 4. Due to space limitations, the details are not presented.

$$PfRootAddr = RootAddr + (Size \times Depth \times Distance)$$

$$PfAddr = Mem[PfRootAddr] + (AccessedAddress - Mem[RootAddr]) \quad (4)$$

D. Distance Selection Unit

Distance Selection Unit (DSU) 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 1 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. DSU 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. DSU 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.

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.

III. Methodology

A. Simulation Environment

We implemented ATP on the **gem5** using System Emulation mode and generated the results using the x86 out-of-order CPU model. Table 1 shows the configuration of each core while Table 2 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 in our evaluation.

Table 1: Simulator Configurations

ISA	64-bit x86
Architecture	4-Issue, out-order, 2GHz
LQ/SQ Entries	64/36
ROB Entries	168
Br. Pred.	Tournament BP
L1 Cache	Private, 8-way 32KB, mshrs: 4, latency: 4
L2 Cache	Private, 8-way 256KB, mshrs: 16, latency: 12
L3 Cache	Shared, 16-way, 1MB per core, mshrs: 16, latency: 32
Memory	8GB DRAM

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 the IMP implementation was 8032 bits (1004 bytes) per core, almost four times the size of our proposed ATP (see Table 2). 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 the simulate 100M instructions; for multi-core simulations, each core simulates at least 100M instructions.

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 2: Hardware budget of ATP on single-core architecture.

	Instr. Queue	Array Table	Relation Table	Operation Table	Prefetch Table	Stream Reg.	Size Calc. Reg.	Base Adr. Reg.	Dist. Sel. Unit
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)									

B. 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 (BSF) 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].

IV. 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.

A. Single-Core Performance of ATP

Figure 4 shows the speedup of SWPF, IMP, and ATP over the no-prefetching baseline architecture. The average (geometric mean) speedup of ATP is 1.60 which outperforms both SWPF (1.49) and IMP (1.16). For *IS*, SPWF and ATP both outperform IMP, while ATP and IMP outperform SPWF for *CG*. As described in § 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.

The speedup for each prefetching method is very modest for *CG* (in fact, negative for SPWF) because *CG* is cache friendly thus limiting the opportunity for prefetching. IMP has almost same performance in this benchmark while ATP has a slight improvement. Because *CG* is so cache friendly, the increased overhead actually decreases performance of SWPF by -19%.

For *PR* and *TC*, SWPF and ATP outperform IMP. More specifically, for *PR*, SWPF and ATP have speedups of 1.07 and 1.06, respectively, while IMP has a slightly negative speedup. For *TC*, ATP and SWPF have a speedup of 1.75 and 1.62, respectively, while IMP has a speedup of 1.23. *PR* and *TC*'s `A[B[i][j]]` type of indirect memory accesses which 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 performance.

For *HJ2* and *HJ8*, ATP and SWPF again outperform IMP. More specifically, for *HJ2*, ATP and SWPF have speedups of 3.27 and 3.26, respectively, while IMP provides effectively no speedup. For *HJ8*, SWPF and ATP have speedups of 1.46 and 1.45, 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 reduce the potential speedup due to prefetching.

For *g500*, the speedup due to ATP (1%), SWPF (0%), and IMP (0%) is effectively zero because the marked loop does not execute continuously for long periods of time. Even though its speedup is very low, ATP still performs slightly better than SWPF and IMP in both single-core and multi-core architectures. For *histo*, the speedups due to ATP and SWPF are 1.52 and 1.35, respectively while IMP provides effectively no speedup. The overhead associated with SWPF degrades its performance as compared with ATP. *histo* speedup by IMP increases significantly for larger prefetch distances, however, larger distances negatively affect other benchmarks. We have used the best overall performing distance for IMP, which is 4.

B. Multi-Core Performance of ATP

Figures 5 and 6 show speedup 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 (1.49) followed by SWPF (1.38) and IMP (1.11). For 8-cores, speedups for ATP, SWPF, and IMP are 1.31, 1.24, and 1.03, respectively. 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. On the other hand, for *HJ2*, although the speedup for each prefetching method decreases as the number of increases, the speedup for 8-cores for ATP and SWPF is still 2.36 and 2.34, respectively. Therefore, while resource contention has some effect on the efficacy of each prefetching method, ATP and SWPF still provide significant speedup.

C. 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 7 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 7 show that dynamically adjusting the prefetching distance has an average speedup of 1.59 while the highest performing fixed distance (distance = 4) yields an average speedup of 1.46. 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.

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 7 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 total instruction executed between two consecutive accesses to the first dimension of array B which favors using shorter distances.

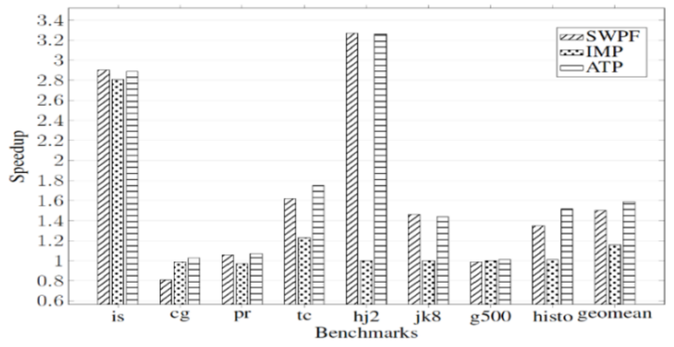

Figure 4: Performance comparison of SWPF, IMP and ATP on single core

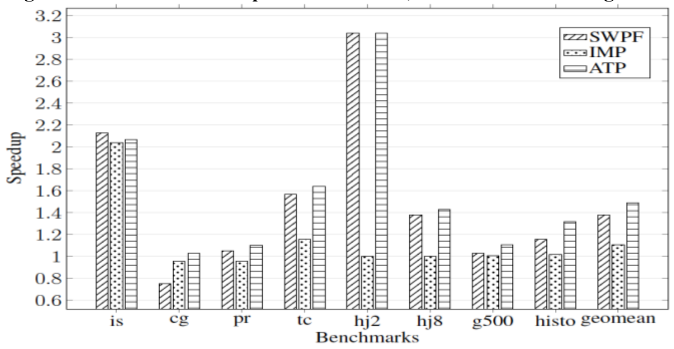

Figure 5: Performance comparison of SWPF, IMP and AT on 4-cores. Baseline is 4-cores with no prefetching.

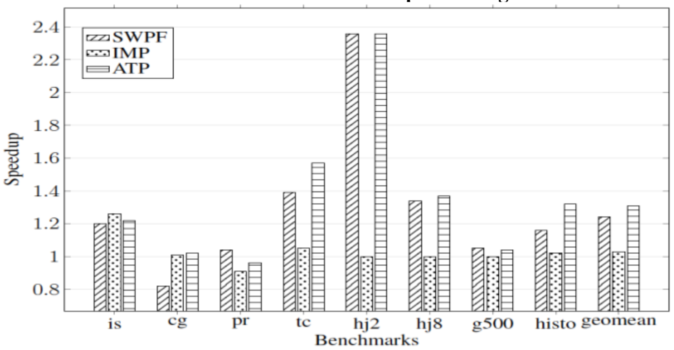

Figure 6: Performance comparison of SWPF, IMP and AT on 8-cores. Baseline is 8-cores with no prefetching.

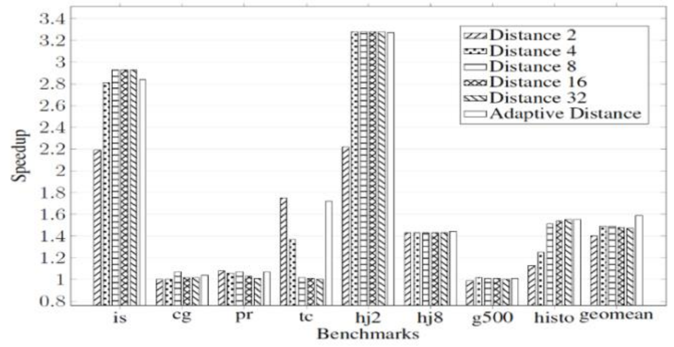

Figure 7: Performance comparison of ATP using different fixed distances and adaptive distance adjustment.

D. 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. Table 4 shows the accuracy and timeliness for different benchmarks using SWPF, IMP, and ATP. Accuracy is the percentage of prefetched cache lines which are accessed later. Timeliness is the percentage of cache hits to previously prefetched cache lines to the total number of accurately prefetched cache lines.

Table 4. Prefetch accuracy and timeliness

Benchmark.	SWPF		IMP		ATP	
	Acc.	Tim.	Acc.	Tim.	Acc.	Tim.
is	100%	100%	100%	76%	100%	94%
cg	100%	94%	79%	39%	95%	83%
pr	100%	8%	88%	86%	100%	98%
tc	100%	6%	100%	100%	100%	98%
hj2	100%	91%	99%	56%	100%	100%
hj8	100%	100%	100%	100%	100%	100%
g500	100%	20%	73%	42%	86%	78%
histo	100%	54%	100%	32%	100%	90%
avg	100%	59%	92%	66%	98%	93%

SWPF has 100% accuracy for all benchmarks because it handles border checks and guarantees the prefetched cache line will be accessed. The prefetch accuracy of ATP and IMP are lower (98% and 92%, respectively). ATP is more accurate than IMP since the software mechanism specifies and limits the prefetches, thus reducing the number of useless prefetches.

Even though we chose the best overall distances for SWPF and IMP, this distance is not best for all the benchmarks well. As such, the average timeliness for SWPF and IMP is 59% and 66%, respectively. By contrast, because ATP dynamically adjusts the distance, the overall timeliness of ATP is significantly higher (93%).

V. Related Work

Hardware prefetching is a well-known technique that rely on past memory access patterns to predict future misses. Many prior hardware prefetching methods exist from stream/stride prefetchers [11, 12] targeting simple patterns to correlation prefetchers [13, 14] maintaining large tables for detecting more complicated access patterns, none of which have been effective for indirect access patterns. 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 a recent 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).

Software prefetching [15-20] provides a way for programmers to insert prefetching instructions into a program targeting various simple and complex patterns. Insertion of instructions can be manual, which requires significant programmer effort, or automatic, which 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.

VI. Conclusion

We propose and implement the Array Tracking Prefetcher to have the benefits of both software and hardware prefetching for indirect memory accesses. ATP inserts 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 1.60X, 1.49X, and 1.31X for single-core, 4-core, and 8-core architectures, respectively. ATP also outperforms software-based and hardware-based (IMP) prefetching methods.

Acknowledgment

This work was supported by NSF grant 1422516.

References

[1] Ainsworth and Jones, “Software prefetching for indirectmemory accesses,” in CGO 2017.
[2] Hashemi et al., “Continuous runahead: transparent hardware acceleration for memory intensive workloads,” in MICRO 2016.
[3] Yu et al., “Imp: Indirect memory prefetcher,” in MICRO 2015.
[4] D. H. Bailey, “Nas parallel benchmarks,” in Encyclopedia of Parallel Computing. Springer, 2011, pp. 1254–1259.
[5] Page et al., “The pagerank citation ranking: Bringing order to the web.” Stanford InfoLab, Tech.Rep., 1999.
[6] Ahmad et al., “Crono: A benchmarksuite for multithreaded graph algorithms executing on futuristic multicores,” in IISWC 2015.
[7] Chiba and Nishizeki, “Arboricity and subgraph listing algorithms,”SIAM Journal on Computing, vol. 14, no. 1, pp. 210–223,1985.
[8] Balkesen et al., “Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware,” in ICDE 2013, pp. 362–373.
[9] Murphy et al., “Introducing the graph 500,” Cray Users Group (CUG), vol. 19, pp. 45–74,2010.
[10] I. R. Group et al., “Parboil benchmark suite,” 2007.
[11] Chen and Baer, “Effective hardware-based data prefetching for high-perf. processors,” in IEEE Trans. Computers, vol. 44, no. 5, pp. 609–623, 1995.
[12] Vanderwiel and Lilja, “Data prefetch mechanisms,” ACM Computing Surveys (CSUR), vol. 32, no. 2, pp. 174–199, 2000.
[13] Joseph and Grunwald, “Prefetching using markov predictors,” in ACM SIGARCH Comp. Arch. News, vol. 25, no. 2. ACM, 1997, pp. 252–263.
[14] K. J. Nesbit and J. E. Smith, “Data cache prefetching using a global history buffer,” Micro, IEEE, vol. 25, no. 1, pp. 90–97, 2005.
[15] Callahan et al., “Software prefetching,”in ACM SIGARCH Computer Architecture News, vol. 19, no. 2.ACM, 1991, pp. 40–52.
[16] Mowry, “Tolerating latency through software-controlled dataprefetching,” Ph.D. dissertation, Stanford University, 1994.
[17] Serrano et al., “Value profile guided stride prefetching for irregular code,” in Compiler Construction. Springer, 2002, pp. 307–324.
[18] Luk and Mowry, “Compiler-based prefetching for recursive data structures,” in ACM SIGOPS Review, vol. 30, no. 5. 1996, pp. 222–233.
[19] Lipasti et al.,“Spaid: Software prefetching in pointer-and call-intensive environments,”in MICRO 1995.
[20] Lee et al., “When prefetching works, when it does not, and why,” ACM Trans. Archit. Code Optim.,vol. 9, no. 1, pp. 2:1–2:29, Mar. 2012.