Address-Value Delta (AVD) Prediction: A Hardware Technique for Efficiently Parallelizing Dependent Cache Misses

Onur Mutlu, *Member*, *IEEE*, Hyesoon Kim, *Student Member*, *IEEE*, and Yale N. Patt, *Fellow*, *IEEE*

Abstract—While runahead execution is effective at parallelizing *independent* long-latency cache misses, it is unable to parallelize *dependent* long-latency cache misses. To overcome this limitation, this paper proposes a novel hardware technique, *address-value delta (AVD) prediction*. An AVD predictor keeps track of the address (pointer) load instructions for which the *arithmetic difference* (i.e., *delta*) between the effective address and the data value is stable. If such a load instruction incurs a long-latency cache miss during runahead execution, its data value is predicted by subtracting the stable delta from its effective address. This prediction enables the preexecution of dependent instructions, including load instructions that incur long-latency cache misses. We analyze why and for what kind of loads AVD prediction works and describe the design of an implementable AVD predictor. We also describe simple hardware and software optimizations that can significantly improve the benefits of AVD prediction and analyze the interaction of AVD prediction with runahead efficiency techniques and stream-based data prefetching. Our analysis shows that AVD prediction is complementary to these techniques. Our results show that augmenting a runahead processor with a simple, 16-entry AVD predictor improves the average execution time of a set of pointer-intensive applications by 14.3 percent (7.5 percent excluding benchmark health).

Index Terms—Single data stream architectures, runahead execution, value prediction, memory-level parallelism.

1 Introduction

Main memory latency is a major performance limiter in current high-performance microprocessors. As the improvement in DRAM memory speed has not kept up with the improvement in processor speed, aggressive high-performance processors are currently facing DRAM latencies of hundreds of processor cycles [37], [34]. The gap between the processor and DRAM memory speed and the resulting negative impact of memory latency on processor performance are expected to continue to increase [39], [37]. Therefore, innovative techniques to tolerate long-latency main memory accesses are needed to improve the performance of memory-intensive application programs. As energy/power consumption has already become a limiting constraint in the design of high-performance processors [12], simple power- and area-efficient memory latency tolerance techniques are especially desirable.

Runahead execution [9], [25] is a promising technique that was recently proposed to tolerate long main memory latencies. This technique speculatively preexecutes the application program while a long-latency data cache miss is being serviced, instead of stalling the processor for the duration of the long-latency miss. In runahead execution [25], if a long-latency (L2 cache miss) load instruction becomes the oldest instruction in the instruction window, it triggers the processor to checkpoint its architectural state and switch to a purely speculative processing mode called *runahead mode*. The processor stays in runahead mode until the cache miss that initiated runahead mode is serviced. During runahead mode, instructions *independent of the pending long-latency cache misses* are speculatively preexecuted. Some of these preexecuted instructions cause long-latency cache misses, which are serviced in parallel with each other and the runahead-causing cache miss. Hence, runahead execution improves latency tolerance and performance by allowing the parallelization of *independent* long-latency cache misses that would otherwise not have been generated because the processor would have stalled. The parallelization of independent long-latency cache misses has been shown to be the major performance benefit of runahead execution [26], [6].

Unfortunately, a runahead execution processor cannot parallelize *dependent* long-latency cache misses. A runahead processor cannot preexecute instructions that are *dependent on the pending long-latency cache misses* during runahead mode since the data values they are dependent on are not available. These instructions are designated as bogus (INV) and they mark their destination registers as INV so that the registers they produce are not used by instructions dependent on them. Hence, runahead execution is not able to parallelize two long-latency cache misses if the load instruction generating the second miss is dependent on the load instruction that generated the first miss.[1] These two misses need to be serviced serially. Therefore, the full-latency of each miss is exposed and the latency tolerance of the processor cannot be improved by runahead execution.

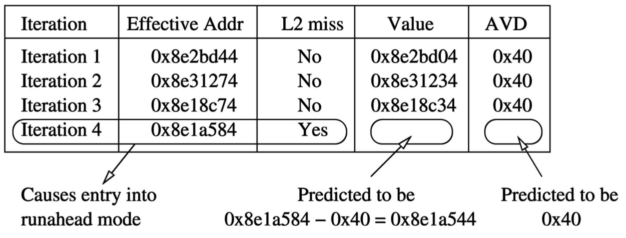
• *O. Mutlu is with Microsoft Research, One Microsoft Way, Redmond, WA 98052. E-mail: onur@microsoft.com.*
• *H. Kim and Y.N. Patt are with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712. E-mail: {hyesoon, patt}@ece.utexas.edu.*

Manuscript received 3 Feb. 2006; revised 23 June 2006; accepted 7 July 2006; published online 20 Oct. 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0037-0206.

1. Two dependent load misses cannot be serviced in parallel in a conventional out-of-order processor either.

```
while (list != NULL) {
    // ...
    p = list–>patient;  // Load 1 – causes 67% of all runahead entries
    // ...
    t = p–>time;  // Load 2 – dependent on load 1, frequently causes L2 misses
    // ...
    list = list–>forward;  // Load 3
}
```

(a)



Iteration	Effective Addr	L2 miss	Value	AVD
Iteration 1	0x8e2bd44	No	0x8e2bd04	0x40
Iteration 2	0x8e31274	No	0x8e31234	0x40
Iteration 3	0x8e18c74	No	0x8e18c34	0x40
Iteration 4	0x8e1a584	Yes		

Causes entry into runahead mode
Predicted to be 0x8e1a584 – 0x40 = 0x8e1a544
Predicted to be 0x40

(b)

Fig. 1. Source code example showing a load instruction with a stable AVD (Load 1). (a) Code example. (b) Execution history of Load 1.

Applications and program segments that heavily utilize linked data structures (where many load instructions are dependent on previous loads) therefore cannot significantly benefit from runahead execution. In fact, for some pointer-chasing applications, runahead execution reduces performance due to its overheads and significantly increases energy consumption due to the increased activity caused by the preprocessing of useless instructions. In a recent paper [23], we showed that the performance benefit of runahead execution would be almost doubled if runahead execution were able to parallelize all dependent L2 cache misses.

In order to overcome the serialization of dependent long-latency cache misses, techniques to parallelize dependent load instructions are needed. These techniques need to focus on predicting the values loaded by *address (pointer) loads*, i.e., load instructions that load an address that is later dereferenced. Several microarchitectural techniques have been proposed to predict the values of address loads [20], [31], [2], [7] or to prefetch the addresses generated by them [29], [30], [7]. Unfortunately, to be effective, these techniques require a large amount of storage and complex hardware control. As energy/power consumption becomes more pressing with each processor generation, simple techniques that require small storage cost become more desirable and necessary. Our goal in this paper is to devise a technique that reduces the serialization of dependent long-latency misses *without significantly increasing the hardware cost and complexity*.

We propose a simple, implementable, novel mechanism, *address-value delta (AVD) prediction*, that allows the parallelization of dependent long-latency cache misses. The proposed technique learns the *arithmetic difference (delta)* between the effective address and the data value of an *address load* instruction based on the previous executions of that load instruction. Stable *address-value deltas* are stored in a prediction buffer. When a load instruction incurs a long-latency cache miss, if it has a stable *address-value delta* in the prediction buffer, its data value is predicted by subtracting the stored delta from its effective address. This predicted value enables the preexecution of dependent instructions, including load instructions that incur long-latency cache misses. We provide source-code examples showing the common code structures that cause stable *address-value deltas*, describe the implementation of a simple *address-value delta* predictor, and evaluate its performance benefits on a runahead execution processor. We show that augmenting a runahead processor with a simple, 16-entry (102-byte) AVD predictor improves the execution time of a set of pointer-intensive applications by 12.1 percent.

We also propose hardware and software optimizations that increase the benefits of AVD prediction. One hardware optimization, called NULL-value optimization, increases the performance improvement of a 16-entry AVD predictor to 14.3 percent. Furthermore, we examine the interaction of AVD prediction with runahead efficiency techniques and stream-based data prefetching and show that AVD prediction is complementary to these previously proposed techniques.

2 AVD Prediction: The Basic Idea

We have observed that some load instructions exhibit stable relationships between their effective addresses and the data values they load. We call this stable relationship the *address-value deltas (AVDs)*. We define the *address-value delta* of a dynamic instance of a load instruction L as:

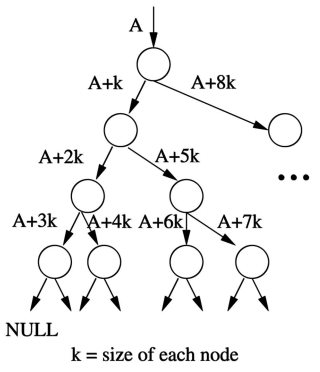
$$AVD(L) = Effective\ Address\ of\ L - Data\ Value\ of\ L.$$

Fig. 1 shows an example load instruction that has a stable AVD and how we can utilize AVD prediction to predict the value of that load in order to enable the execution of a dependent load instruction. The code example in this figure is taken from the `health` benchmark. Load 1 frequently misses in the L2 cache and causes the processor to enter runahead mode. When Load 1 initiates entry into runahead mode in a conventional runahead processor, it marks its destination register as INV (bogus). Load 2, which is dependent on Load 1, therefore cannot be executed during runahead mode. Unfortunately, Load 2 is also an important load that frequently misses in the L2 cache. If it were possible to correctly predict the value of Load 1, Load 2 could be executed and the L2 miss it causes would be serviced in parallel with the L2 miss caused by Load 1, which initiated entry into runahead mode.

Fig. 1b shows how the value of Load 1 can be accurately predicted using an AVD predictor. In the first three executions of Load 1, the processor calculates the AVD of the instruction. The AVD of Load 1 turns out to be stable and it is recorded in the AVD predictor. In the fourth execution, Load 1 misses in the L2 cache and causes entry into runahead mode. Instead of marking the destination register of Load 1 as INV, the processor accesses the AVD predictor with the program counter of Load 1. The predictor returns the stable AVD corresponding to Load 1. The value

```
treeNode* TreeAlloc(level) {
    // ...
    if (level == 0)
        return NULL;
    else {
        new = (treeNode *) malloc (...);
        left = TreeAlloc(...);
        right = TreeAlloc(...);
        new->left = left;
        new->right = right;
        return new;
    }
}
```

(a)



(b)

```
int TreeAdd(treeNode *t) {
    if (t == 0)
        return 0;
    else {
        tleft = t->left;  // Load 1
        leftval = TreeAdd(tleft);
        tright = t->right;  // Load 2
        rightval = TreeAdd(tright);
        // ...
    }
}
```

(c)

Eff. Addr	Value	AVD
A	A+k	−k
A+k	A+2k	−k
A+2k	A+3k	−k

(d)

Fig. 2. An example from `treeadd` showing how stable AVDs can occur for traversal address loads. (a) Source code of the recursive function that allocates the binary tree. (b) Layout of the binary tree in memory (A is the address of the root node). (c) Source code of the recursive function that traverses the tree. (d) Execution history of Load 1.

of Load 1 is predicted by subtracting the AVD returned by the predictor from the effective address of Load 1 such that:

$$Predicted\ Value = Effective\ Address - Predicted\ AVD.$$

The predicted value is written into the destination register of Load 1. The dependent instruction, Load 2, reads this value and is able to calculate its address. Load 2 accesses the cache hierarchy with its calculated address and it may generate an L2 cache miss, which would be serviced in parallel with the L2 cache miss generated by Load 1.

Note that Load 1 in Fig. 1 is an *address (pointer) load*. We distinguish between *address loads* and *data loads*. An *address load* is a load instruction that loads an address into its destination register that is later used to calculate the effective address of itself or another load instruction (Load 3 is also an address load). A *data load* is a load whose destination register is not used to calculate the effective address of another load instruction (Load 2 is a data load). We are interested in predicting the values of only *address loads*, not *data loads*, since address loads—by definition—are the only load instructions that can lead to the generation of dependent long-latency cache misses. In order to distinguish address loads from data loads in hardware, we bound the values AVD can take. We only consider predicting the values of load instructions that have—in the past—satisfied the equation:

$$-MaxAVD \leq AVD(L) \leq MaxAVD,$$

where $MaxAVD$ is a constant set at the design time of the AVD predictor. In other words, in order to be identified as an address load, the data value of a load instruction needs to be *close enough* to its effective address. If the AVD is too large, it is likely that the value that is being loaded by the load instruction is not an address.[2] Note that this mechanism is similar to the mechanism proposed by Cooksey et al. [8] to identify address loads in hardware. Their mechanism identifies a load as an address load if the upper N bits of the effective address of the load match the upper N bits of the value being loaded.

2. An alternative mechanism is to have the compiler designate the address loads with a single bit augmented in the load instruction format of the ISA. We do not explore this option since our goal is to design a simple purely hardware mechanism that requires no software or ISA support.

3 Why Do Stable AVDs Occur?

Stable AVDs occur due to the regularity in the way data structures are allocated in memory by the program, which is sometimes accompanied by regularity in the input data to the program. We examine the common code constructs in application programs that give rise to regular memory allocation patterns that result in stable AVDs for some address loads. For our analysis, we distinguish between what we call *traversal address loads* and *leaf address loads*. A traversal address load is a static load instruction that produces an address that is later consumed by itself or another address load, such as in a linked list or tree traversal (e.g., Load 3 in Fig. 1 is a traversal address load). A leaf address load produces an address that is later consumed by a data load (e.g., Load 1 in Fig. 1 is a leaf address load).

3.1 Stable AVDs in Traversal Address Loads

A traversal address load may have a stable AVD if there is a pattern to the allocation and linking of the nodes of a linked data structure. If the allocation of the nodes is performed in a regular fashion, the nodes will have a constant distance in memory from one another. If a traversal load instruction later traverses the linked data structure nodes that have the same distance from one another, the traversal load can have a stable AVD.

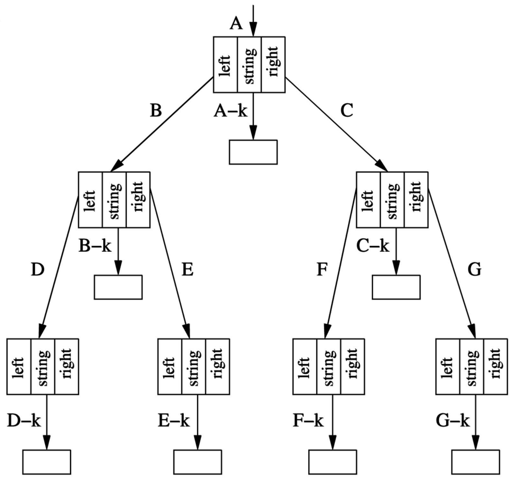
Fig. 2 shows an example from `treeadd`, a benchmark whose main data structure is a binary tree. In this benchmark, a binary tree is allocated in a regular fashion using a recursive function where a node is allocated first and its left child is allocated next (Fig. 2a). Each node of the tree is of the same size. The layout of an example resulting binary tree is shown in Fig. 2b. Due to the regularity in the allocation of the nodes, the distance in memory of each node and its left child is constant. The binary tree is later traversed using another recursive function (Fig. 2c). Load 1 in the traversal function traverses the nodes by loading the pointer to the left child of each node. This load instruction

```
struct Dict_node {
    char *string;
    Dict_node *left, *right;
    // ...
}

char *get_a_word(...) {
    // read a word from file
    s = (char *) xalloc(strlen(word) + 1);
    strcpy(s, word);
    return s;
}

Dict_node *read_word_file(...) {
    // ...
    char *s; Dict_node *dn;
    while ((s = get_a_word(...)) != NULL) {
        dn = (Dict_node *) xalloc(sizeof(Dict_node));
        dn->string = s;
        // ...
    }
    return dn;
}
```

(a)



(b)

```
int dict_match (char *s, char *t) {
    while((*s != '\0') && (*s == *t))
     {s++; t++;}
    if ((*s == '*') || (*t == '*')) return 0;
    // ...
}

rabridged_lookup(Dict_node *dn, char *s) {
    // ...
    if (dn == NULL) return;
    t = dn->string; // Load 1
    m = dict_match(s, t);
    if (m>=0) rabridged_lookup(dn->right, s);
    if (m<=0) rabridged_lookup(dn->left, s);
}
```

Eff. Addr.	Value	AVD
A	A–k	k
C	C–k	k
F	F–k	k

(c)

Fig. 3. An example from `parser` showing how stable AVDs can occur for leaf address loads. (a) Source code that allocates the nodes of the dictionary (binary tree) and the strings. (b) Layout of the dictionary in memory (the value on an arc denotes the memory address of the structure that is pointed to). (c) Source code of the recursive function that performs the dictionary lookup and the execution history of Load 1.

has a stable AVD, as can be seen from its example execution history (Fig. 2d). Load 1 has a stable AVD because the distance in memory of a node and its left child is constant. We found that this load causes 64 percent of all entries into runahead mode and predicting its value correctly enables the generation of dependent L2 misses (generated by the same instruction) during runahead mode.

As evident from this example, the stability of AVDs in traversal address loads is also dependent on the behavior of the memory allocator. If the memory allocator allocates memory chunks in a regular fashion (e.g., allocating fixed-size chunks from a contiguous section of memory), the likelihood of the occurrence of stable AVDs increases. On the other hand, if the behavior of the memory allocator is irregular, the distance in memory of a node and the node(s) it is linked to may be totally unpredictable; hence, the resulting AVDs would not be stable.

We also note that stable AVDs occurring due to regularity in the allocation and linking of the nodes can disappear if the linked data structure is significantly reorganized during runtime unless the reorganization of the data structure is performed in a regular fashion. Therefore, AVD prediction may not work for traversal address loads in applications that require extensive modifications to the linkages in linked data structures.

3.2 Stable AVDs in Leaf Address Loads

A leaf address load may have a stable AVD if the allocation of a data structure node and the allocation of a field that is linked to the node via a pointer are performed in a regular fashion.

Fig. 3 shows an example from `parser`, a benchmark that parses an input file and looks up the parsed words in a dictionary. The dictionary is constructed at the startup of the program. It is stored as a sorted binary tree. Each node of the tree is a `Dict_node` structure that contains a pointer to the `string` corresponding to it as one of its fields. Both `Dict_node` and `string` are allocated dynamically as shown in Fig. 3a. First, memory space for `string` is allocated. Then, memory space for `Dict_node` is allocated and it is linked to the memory space of `string` via a pointer. The layout of an example dictionary is shown in Fig. 3b. In contrast to the binary tree example from `treeadd`, the distance between the nodes of the dictionary in `parser` is not constant because the allocation of the dictionary nodes is performed in a somewhat irregular fashion (not shown in Fig. 3) and because the dictionary is kept sorted. However, the distance in memory between each node and its associated `string` is constant. This is due to the behavior of the `xalloc` function that is used to allocate the `strings` in combination with regularity in input data. We found that `xalloc` allocates a fixed-size block of memory for the `string` if the length of the string is within a certain range. As the length of most `strings` falls into that range (i.e., the input data has regular behavior), the memory spaces allocated for them are of the same size.[3]

Words are later looked up in the dictionary using the `rabridged_lookup` function (Fig. 3c). This function recursively searches the binary tree and checks whether the `string` of each node is the same as the input word `s`. The `string` in each node is loaded by Load 1 (`dn->string`), which is a leaf address load that loads an address that is later dereferenced by data loads in the `dict_match` function. This load has a stable AVD, as shown in its example execution history, since the distance between a

3. The code shown in Fig. 3a can be rewritten such that memory space for a `Dict_node` is allocated first and the memory space for its associated `string` is allocated next. In this case, even though the input data may not be regular, the distance in memory of each node and its associated string would be constant. We did not perform this optimization in our baseline evaluations. However, the effect of this optimization is evaluated separately in Section 7.2.

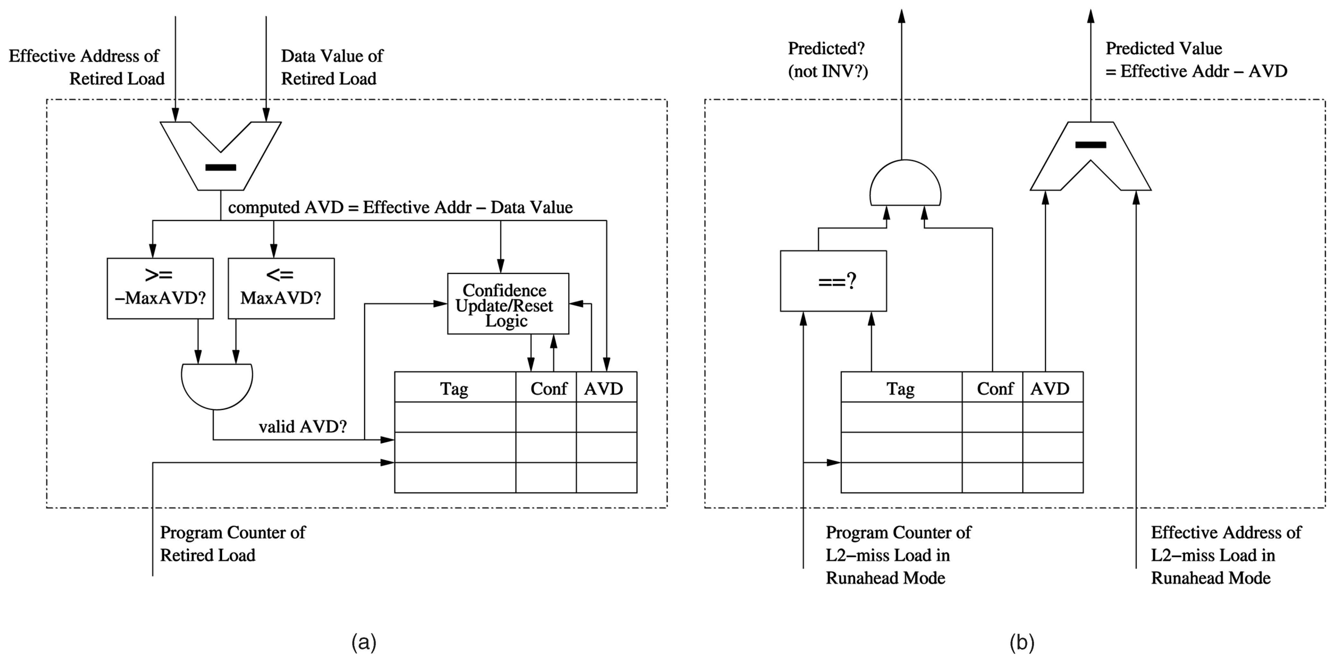

Effective Address of Retired Load
Data Value of Retired Load
computed AVD = Effective Addr – Data Value
>= –MaxAVD?
<= MaxAVD?
Confidence Update/Reset Logic
Tag
Conf
AVD
valid AVD?
Program Counter of Retired Load
(a)
Predicted? (not INV?)
Predicted Value = Effective Addr – AVD
==?
Tag
Conf
AVD
Program Counter of L2–miss Load in Runahead Mode
Effective Address of L2–miss Load in Runahead Mode
(b)

Fig. 4. Organization of the AVD predictor. (a) Update logic for the AVD predictor. (b) Prediction logic for the AVD predictor.

node and its associated string is constant. The values generated by Load 1 are hard to predict using a traditional stride- or context-based value predictor because they do not follow a pattern. In contrast, the AVDs of Load 1 are quite easy to predict. We found that this load causes 36 percent of the entries into runahead mode and correctly predicting its value enables the execution of the dependent load instructions (and the dependent conditional branch instructions) in the `dict_match` function.

Stable AVDs occurring in leaf address loads continue to be stable even if the linked data structure is significantly reorganized at runtime. This is because such AVDs are caused by the regularity in the links between nodes and their fields rather than the regularity in the links between nodes and other nodes. The reorganization of the linked data structure changes the links between nodes and other nodes, but leaves intact the links between nodes and their fields.

4 Design and Operation of a Recovery-Free AVD Predictor

An AVD predictor records the AVDs and information about the stability of the AVDs for address load instructions. The predictor is updated when an address load is retired. The predictor is accessed when a load misses in the L2 cache during runahead mode. If a stable AVD associated with the load is found in the predictor, the predicted value for the load is calculated using its effective address and the stable AVD. The predicted value is then returned to the processor to be written into the register file.

Fig. 4 shows the organization of the AVD predictor along with the hardware support needed to update/train it (Fig. 4a) and the hardware support needed to make a prediction (Fig. 4b). Each entry of the predictor consists of three fields: *Tag*, the upper bits of the program counter of the load that allocated the entry, *AVD*, the address-value delta that was recorded for the last retired load associated with the entry, and *Confidence (Conf)*, a saturating counter that records the confidence of the recorded AVD (i.e., how many times the recorded AVD was seen consecutively). The confidence field is used to eliminate incorrect predictions for loads with unstable AVDs.

4.1 Operation

At initialization, the confidence counters in all the predictor entries are reset to zero. There are two major operations performed on the AVD predictor: update and prediction.

The predictor is updated when a load instruction is retired during normal mode. The predictor is accessed with the program counter of the retired load. If an entry does not already exist for the load in the predictor and if the load has a valid AVD, a new entry is allocated. To determine if the load has a valid AVD, the AVD of the instruction is computed and compared to the minimum and maximum allowed AVD. If the computed AVD is within bounds [`-MaxAVD`, `MaxAVD`], the AVD is considered valid. On the allocation of a new entry, the computed AVD is written into the predictor and the confidence counter is set to one. If an entry already exists for the retired load, the computed AVD is compared with the AVD that is stored in the existing entry. If the two match, the confidence counter is incremented. If the AVDs do not match and the computed AVD is valid, the computed AVD is stored in the predictor entry and the confidence counter is set to one. If the computed AVD is not valid and the load instruction has an associated entry in the predictor, the confidence counter is reset to zero, but the stored AVD is not updated.[4]

4. As an optimization, it is possible to *not update* the AVD predictor state, including the confidence counters, if the data value of the retired load is zero. A data value of zero has a special meaning for address loads, i.e., NULL pointer. This optimization reduces the training time or eliminates the need to retrain the predictor and thus helps benchmarks where loads that perform short traversals are common. The effect of this optimization on AVD predictor performance is evaluated in Section 7.1.

TABLE 1
Baseline Processor Configuration

Pipeline	24-stage pipeline, 20-cycle minimum branch misprediction penalty
Front End	64KB, 4-way instruction cache with 2-cycle latency; 8-wide decoder with 1-cycle latency; 8-wide renamer with 4-cycle latency
Branch Predictors	64K-entry gshare/per-address hybrid with 64K-entry selector; 4K-entry, 4-way branch target buffer; 64-entry return address stack; 4K-entry target cache for indirect branches; wrong-path execution faithfully modeled (including misprediction recoveries on the wrong path)
Instruction Window	128-entry reorder buffer; 128-entry INT, 128-entry FP physical register files with 4-cycle latency; 128-entry load/store buffer, store misses do not block the instruction window unless store buffer is full
Execution Core	8 general-purpose functional units, fully-pipelined except for FP divide; full bypass network
On-chip Caches	64KB, 4-way L1 data cache with 8 banks and 2-cycle latency, allows 4 load accesses per cycle; 1-cycle AGEN latency; 1MB, 32-way, unified L2 cache with 8 banks and 10-cycle latency, maximum 128 outstanding L2 misses, 1 L2 read port, 1 L2 write port; all caches use LRU replacement and have 64B line size
Buses and Memory	500-cycle minimum main memory latency; 32 DRAM banks; 32B-wide, split-transaction core-to-memory bus at 4:1 frequency ratio; maximum 128 outstanding misses to main memory; bank conflicts, bandwidth, and queueing delays are faithfully modeled at all levels in the memory hierarchy
Runahead Support	128-byte runahead cache [25] for store-load data forwarding during runahead mode

The predictor is accessed when a load instruction misses in the L2 cache during runahead mode. The predictor is accessed with the program counter of an L2-miss load. If an entry exists for the load and if the confidence counter is saturated (i.e., above a certain confidence threshold), the value of the load is predicted. The predicted value is computed by subtracting the AVD stored in the predictor entry from the effective virtual address of the L2-miss load. If an entry does not exist for the load in the predictor, the value of the load is not predicted. Two outputs are generated by the AVD predictor: a *predicted* bit which informs the processor whether or not a prediction is generated for the load and the *predicted value*. If the *predicted* bit is set, the *predicted value* is written into the destination register of the load so that its dependent instructions read it and are executed. If the *predicted* bit is not set, the processor discards the *predicted value* and marks the destination register of the load as INV in the register file (as in conventional runahead execution [25]) so that dependent instructions are marked as INV and their results are not used.

The AVD predictor does not require any hardware for state recovery on AVD or branch mispredictions. Branch mispredictions do not affect the state of the AVD predictor since the predictor is updated only by retired load instructions (i.e., there are no wrong-path updates). The correctness of the AVD prediction cannot be determined until the L2 miss that triggered the prediction returns back from main memory. We found that it is not worth updating the state of the predictor on an AVD misprediction detected when the L2 cache miss returns back from main memory since the predictor will be updated anyway when the load is reexecuted and retired in normal execution mode after the processor exits from runahead mode.

An AVD misprediction can occur only in runahead mode. When it occurs, instructions that are dependent on the predicted L2-miss load can produce incorrect results. This may result in the generation of incorrect prefetches or the overturning of correct branch predictions. However, since runahead mode is purely speculative,[5] there is no need to recover the processor state on an AVD misprediction. We found that an incorrect AVD prediction is not necessarily harmful for performance. If the predicted AVD is close enough to the actual AVD of the load, dependent instructions sometimes still generate useful L2 cache misses that are later needed by the processor in normal mode. Hence, we do not initiate state recovery on AVD mispredictions that are resolved during runahead mode.

4.2 Hardware Cost and Complexity

Our goal in the design of the AVD predictor is to avoid high hardware complexity and large storage requirements, but to still improve performance by focusing on predicting the addresses of an important subset of address loads. Since the AVD predictor filters out the loads for which the absolute value of the AVD is too large (using the *MaxAVD* threshold), the number of entries required in the predictor does not need to be large. In fact, Section 6 shows that a 4-entry AVD predictor is sufficient to get most of the performance benefit of the described mechanism. The storage cost required for a 4-entry predictor is very small (212 bits[6]). The logic required to implement the AVD predictor is also relatively simple, as shown in Fig. 4. Furthermore, neither the update nor the access of the AVD predictor is on the critical path of the processor. The update is performed after retirement, which is not on the critical path. The access (prediction) is performed only for load instructions that miss in the L2 cache and it does not affect the critical L1 or L2 cache access times. Therefore, the complexity of the processor or the memory system is not significantly increased with the addition of an AVD predictor.

5 Performance Evaluation Methodology

We evaluate the performance impact of AVD prediction on an execution-driven Alpha ISA simulator that models an aggressive superscalar, out-of-order execution processor. The baseline processor employs runahead execution as described by Mutlu et al. [25] in order to tolerate long L2 cache miss latencies. The parameters of the processor we model are shown in Table 1.

We evaluate AVD prediction on 11 pointer-intensive and memory-intensive benchmarks from the Olden [28] and SPEC INT 2000 benchmark suites. We examine seven

5. That is, runahead mode makes no changes to the architectural state of the processor.

6. Assuming a 4-entry, 4-way AVD predictor with 53 bits per entry: 32 bits for the tag, 17 bits for the AVD (i.e., $\texttt{MaxAVD} = 65,535$), 2 bits for confidence, and 2 bits to support a True LRU (Least Recently Used) replacement policy.

TABLE 2
Relevant Information about the Studied Benchmarks

	bisort	health	mst	perimeter	treeadd	tsp	voronoi	mcf	parser	twolf	vpr
Input Data Set	250,000 integers	5 levels 500 iters	512 nodes	4K x 4K image	1024K nodes	100,000 cities	20,000 points	smred.in	test.in	ref	ref
Simulated instruction count	468M	197M	88M	46M	191M	1050M	139M	110M	412M	250M	250M
Baseline IPC	1.07	0.05	1.67	0.92	0.90	1.45	1.31	0.97	1.33	0.73	0.88
Exec. time reduction due to runahead	19.3%	-3.7%	62.4%	30.5%	22.5%	0.9%	22.1%	37.8%	2.4%	22.3%	18.9%
L2 data misses per 1K instructions	1.03	41.59	5.60	4.27	4.33	0.67	2.41	29.60	1.05	2.37	1.69
% L2 misses due to address loads	72.1%	73.5%	33.8%	62.9%	57.6%	46.2%	78.6%	50.3%	30.1%	26.3%	2.1%

IPC and L2 miss rates are shown for the baseline runahead processor.

memory-intensive benchmarks from the Olden suite, which gain at least 10 percent performance improvement with a perfect L2 cache, and the four relatively pointer-intensive benchmarks (`mcf`, `parser`, `twolf`, `vpr`) from the SPEC INT 2000 suite. All benchmarks were compiled for the Alpha EV6 ISA with the `-O3` optimization level. `Twolf` and `vpr` benchmarks are simulated for 250 million instructions after skipping the program initialization code using a SimPoint-like tool [32]. To reduce simulation time, `mcf` is simulated using the MinneSPEC reduced input set [17]. `Parser` is simulated using the test input set. Olden benchmarks are executed to completion. We used the simple, general-purpose memory allocator (malloc) provided by the standard C library on an Alpha OSF1 V5.1 system. We did not consider a specialized memory allocator that would further benefit AVD prediction and we leave the design of such memory allocators for future work.

Table 2 shows information relevant to our studies about the simulated benchmarks. Unless otherwise noted, performance improvements are reported in terms of execution time normalized to the baseline processor throughout this paper. IPCs of the evaluated processors, if needed, can be computed using the baseline IPC (retired Instructions Per Cycle) performance numbers provided in Table 2 and the normalized execution times. Table 2 also shows the execution time reduction due to runahead execution and we can see that the baseline runahead mechanism provides significant performance improvements except for three benchmarks, *health*, *tsp*, and *parser*. In addition, the fraction of L2 misses that are due to address loads is shown for each benchmark since our mechanism aims to predict the addresses loaded by address loads. We note that, in all benchmarks except *vpr*, at least 25 percent of the L2 cache data misses are caused by address loads. Benchmarks from the Olden suite are more address-load intensive than the set of pointer-intensive benchmarks in the SPEC INT 2000 suite. Hence, we expect our mechanism to perform better on Olden applications.

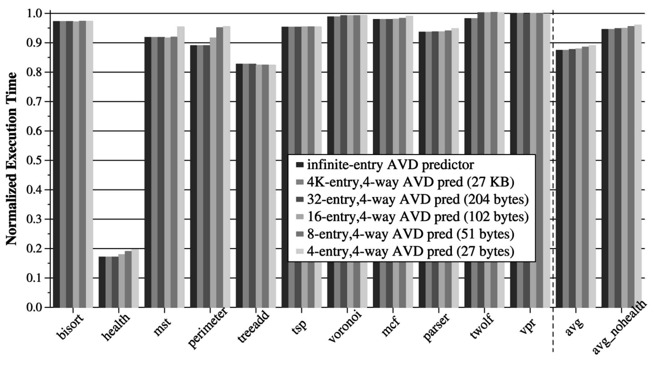


Fig. 5. AVD prediction performance on a runahead processor.

6 Performance of the Baseline AVD Prediction Mechanism

Fig. 5 shows the performance improvement obtained when the baseline runahead execution processor is augmented with the AVD prediction mechanism. We model an AVD predictor with a `MaxAVD` of 64K. A prediction is made if the confidence counter has a value of 2 (i.e., if the same AVD was seen consecutively in the last two executions of the load). On average, the execution time is improved by 12.6 percent (5.5 percent when `health` is excluded[7]) with the use of an infinite-entry AVD predictor. No performance degradation is observed on any benchmark. Benchmarks that have a very high L2 cache miss rate, most of which is caused by address loads (`health`, `perimeter`, and `treeadd`, as seen in Table 2), see the largest improvements in performance. Benchmarks with few L2 misses caused by address loads (e.g., *vpr*) do not benefit from AVD prediction.

A 32-entry, 4-way AVD predictor improves the execution time as much as an infinite-entry predictor for all benchmarks except `twolf`. In general, as the predictor size decreases, the performance improvement provided by the predictor also decreases. However, even a 4-entry AVD predictor improves the average execution time by 11.0 percent (4.0 percent without `health`). Because AVD prediction aims to predict the values produced by a regular subset of address loads, it does not need to keep track of data loads or address loads with very large AVDs. Thus, the number of load instructions competing for entries in the AVD predictor is fairly small and a small predictor is good at capturing them.

Table 3 provides insight into the performance improvement of AVD prediction by showing the increase in memory-level parallelism [11], [6] achieved with and without a 16-entry AVD predictor. We define the memory-level parallelism in a runahead period as the number of useful L2 cache misses generated in a runahead period.[8]

7. The performance of `health` can be improved by orders of magnitude by rewriting the program, as shown by Zilles [42]. Therefore, we show average performance results both including and excluding this program throughout the rest of the paper.

8. A useful L2 cache miss is an L2 cache miss generated during runahead mode that is later needed by a correct-path instruction in normal mode. Only L2 line (block) misses that cannot already be generated by the processor's fixed-size instruction window are counted.

TABLE 3
Average Number of Useful L2 Cache Misses Generated during a Runahead Period with a 16-Entry AVD Predictor

	bisort	health	mst	perimeter	treeadd	tsp	voronoi	mcf	parser	twolf	vpr	avg
L2 misses - baseline runahead	2.01	0.03	7.93	1.45	1.02	0.19	0.81	11.51	0.12	0.84	0.94	2.44
L2 misses - 16-entry AVD pred	2.40	6.36	8.51	1.67	1.53	0.25	0.90	12.05	0.50	0.87	0.94	3.27
% reduction in execution time	2.9%	82.1%	8.4%	8.4%	17.6%	4.5%	0.8%	2.1%	6.3%	0.0%	0.0%	12.1%

With AVD prediction, the average number of L2 cache misses parallelized in a runahead period increases from 2.44 to 3.27. Benchmarks that show large increases in the average number of useful L2 misses with an AVD predictor also show large increases in performance.

We have analyzed the performance improvement of AVD prediction and evaluated the design options, such as the confidence threshold and the value of *MaxAVD*, in an implementable AVD predictor in a previous paper [23]. In this paper, after briefly analyzing the performance of AVD prediction with respect to stride value prediction (in Section 6.1), we focus our attention on new hardware and software optimizations that can improve the performance benefits of AVD prediction (Section 7) and examine the interaction of AVD prediction with previously proposed runahead efficiency techniques and stream-based data prefetching (Section 8).

6.1 AVD Prediction versus Stride Value Prediction

We compare the proposed AVD predictor to stride value prediction [31]. When an L2-miss is encountered during runahead mode, the stride value predictor (SVP) is accessed for a prediction. If the SVP generates a confident prediction, the value of the L2-miss load is predicted. Otherwise, the L2-miss load marks its destination register as INV. Fig. 6 shows the normalized execution times obtained with an AVD predictor, a stride value predictor, and a hybrid AVD-stride value predictor.[9] Stride value prediction is more effective when the predictor is larger, but it provides only 4.5 percent (4.7 percent without `health`) improvement in average execution time even with a 4K-entry predictor versus the 12.6 percent (5.5 percent without `health`) improvement provided by the 4K-entry AVD predictor. With a small, 16-entry predictor, stride value prediction improves the average execution time by 2.6 percent (2.7 percent without `health`), whereas AVD prediction results in 12.1 percent (5.1 percent without `health`) performance improvement. The filtering mechanism (i.e., the `MaxAVD` threshold) used in the AVD predictor to identify and predict only address loads enables the predictor to be small and still provide significant performance improvements.

The benefits of stride and AVD predictors overlap for traversal address loads. Both predictors can capture the values of traversal address loads if the memory allocation pattern is regular. Many L2 misses in `treeadd` are due to traversal address loads, which is why both SVP and AVD predictors perform very well and similarly for this benchmark.

Most leaf address loads cannot be captured by SVP, whereas an AVD predictor can capture those with constant AVD patterns. The benchmark `health` has many AVD-predictable leaf address loads, an example of which is described in detail in [23]. The traversal address loads in `health` are irregular and, therefore, cannot be captured by either SVP or AVD. Hence, AVD prediction provides significant performance improvement in `health`, whereas SVP does not. We found that benchmarks `mst`, `perimeter`, and `tsp` also have many leaf address loads that can be captured with an AVD predictor but not with SVP.

In contrast to an AVD predictor, an SVP is able to capture data loads with constant strides. For this reason, SVP significantly improves the performance of `parser`. In this benchmark, correctly value-predicted L2-miss data loads lead to the execution and correct resolution of dependent branches which were mispredicted by the branch predictor. SVP improves the performance of `parser` by keeping the processor on the correct path during runahead mode rather than by allowing the parallelization of dependent cache misses.

Fig. 6 also shows that combining stride value prediction and AVD prediction results in a larger performance improvement than that provided by either of the prediction mechanisms alone. For example, a 16-entry hybrid AVD-SVP predictor results in 13.4 percent (6.5 percent without health) improvement in average execution time. As shown in code examples in Section 3, address-value delta predictability is different in nature from stride value predictability. A load instruction can have a predictable AVD but not a predictable stride and vice versa. Therefore, an AVD predictor and a stride value predictor sometimes generate predictions for loads with different behavior, resulting in increased performance improvement when they are combined. This effect is especially salient in *parser*, where we

9. In our experiments, the hybrid AVD-SVP predictor does not require extra storage for the selection mechanism. Instead, the prediction made by the SVP is given higher priority than the prediction made by the AVD predictor. If the SVP generates a confident prediction for an L2-miss load, its prediction is used. Otherwise, the prediction made by the AVD predictor is used, if confident.

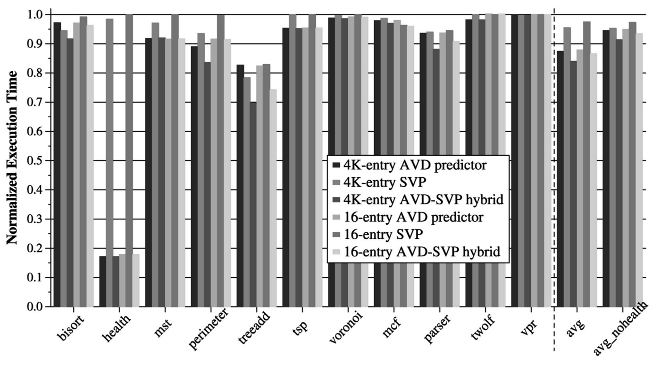


Fig. 6. AVD prediction versus stride value prediction.

found that the AVD predictor is good at capturing leaf address loads and the SVP is good at capturing zero-stride data loads.

In summary, AVD prediction advances the state-of-the-art in the following ways as compared to stride value prediction:

1. AVD prediction can capture stable patterns in leaf address loads, whereas SVP cannot.
2. AVD prediction detects stable relationships between the input (address) and output (data value) of a load instruction, whereas SVP detects stable relationships in the outputs (data values) *across* successive dynamic instances of a load instruction. This enables two benefits:
   - First, an AVD predictor can capture stable AVDs that occur for a traversal load instruction that does not have a stable stride because of the way the data structure is traversed. For example, in the traversal of a binary tree, there may be a stable AVD for `node = node->left` (because a node and its left child may be allocated consecutively), but the successive instances of the same instruction may not have a stable stride because the tree may be traversed in an irregular fashion, visiting the left children of some nodes and the right children of others.
   - Second, the hardware for an AVD predictor can be simple because the detection and maintenance of the stable pattern does not require the examination of (the outputs of) different instances of a load instruction. Maintenance of a stride across different instances of an instruction in a pipelined processor is relatively complex, as described in [2].
3. An AVD predictor does not require a large number of entries to provide large performance improvements because it filters out the data load instructions from the predictor table. In contrast, SVP does require a large number of predictor entries because it tries to predict the data values of all load instructions. Since the address load instruction working set of a program is usually much larger than its data load instruction working set, an AVD predictor can be very small.

7 Hardware and Software Optimizations for AVD Prediction

The results presented in the previous section were based on the baseline AVD predictor implementation described in Section 4. This section explores one hardware optimization and one software optimization that increases the benefits of AVD prediction by taking advantage of the data structure traversal and memory allocation characteristics in application programs.

7.1 Null-Value Optimization

In the AVD predictor we have evaluated, the confidence counter of an entry is reset if the computed AVD of the retired address load associated with the entry is not valid (i.e., not within bounds [`-MaxAVD`, `MaxAVD`]). The AVD of a load instruction with a data value of zero is almost always invalid because the effective addresses computed by load instructions tend to be very large in magnitude. As a result, the confidence counter of an entry is reset if the associated load is retired with a data value of 0 (zero). For address loads, a zero data value has a special meaning: A NULL pointer is being loaded. This indicates the end of a linked data structure traversal. If a NULL pointer is encountered for an address load, it may be better *not to reset* the confidence counter for the corresponding AVD, because the AVD of the load may otherwise be stable except for the intermittent instabilities caused by NULL pointer loads. This section examines the performance impact of not updating the AVD predictor if the value loaded by a retired address load is zero. We call this optimization the *NULL-value optimization*.

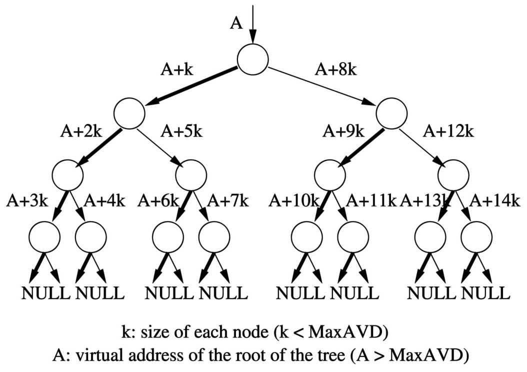

A
A+k
A+8k
A+2k
A+5k
A+9k
A+12k
A+3k
A+4k
A+6k
A+7k
A+10k
A+11k
A+13k
A+14k
NULL NULL NULL NULL NULL NULL NULL NULL
k: size of each node (k < MaxAVD)
A: virtual address of the root of the tree (A > MaxAVD)

Fig. 7. An example binary tree traversed by the `treeadd` program. Links traversed by Load 1 in Fig. 2 are shown in bold.

If the AVD of a load is stable except when a NULL pointer is loaded, resetting the confidence counter upon encountering a NULL pointer may result in a reduction in the prediction coverage of an AVD predictor. We show why this can happen with an example. Fig. 7 shows an example binary tree that is traversed by the `treeadd` program. The tree is traversed with the source code shown in Fig. 2. The execution history of the load that accesses the left child of each node (Load 1 in Fig. 2) is shown in Table 4. This table also shows the predictions for Load 1 that would be made by two different AVD predictors: one that resets the confidence counters on a NULL value and one that does not change the confidence counters on a NULL value. Both predictors have a confidence threshold of 2. To simplify the explanation of the example, we assume that the predictor is updated before the next dynamic instance of Load 1 is executed.[10]

The execution history of Load 1 shows that not updating the AVD predictor on a NULL value is a valuable optimization. If the confidence counter for Load 1 in the AVD predictor is reset on a NULL data value, the AVD predictor generates a prediction for only three instances of Load 1 out of a total of 15 dynamic instances (i.e., $\text{coverage} = 20\%$). Only one of these predictions is correct

10. This may not be the case in an out-of-order processor. Our simulations faithfully model the update of the predictor based on information available to the hardware.

TABLE 4
Execution History of Load 1 in the `treeadd` Program (See Fig. 2) for the Binary Tree Shown in Fig. 7

Dynamic instance	Effective Address	Data Value	Correct AVD	AVD valid?	Predicted AVD and (value) **reset on NULL**	Predicted AVD and (value) **no reset on NULL**
1	A	A+k	-k	valid	no prediction	no prediction
2	A+k	A+2k	-k	valid	no prediction	no prediction
3	A+2k	A+3k	-k	valid	-k (A+3k)	-k (A+3k)
4	A+3k	0 (NULL)	A+3k	not valid	-k (A+4k)	-k (A+4k)
5	A+4k	0 (NULL)	A+4k	not valid	no prediction	-k (A+5k)
6	A+5k	A+6k	-k	valid	no prediction	-k (A+6k)
7	A+6k	0 (NULL)	A+6k	not valid	no prediction	-k (A+7k)
8	A+7k	0 (NULL)	A+7k	not valid	no prediction	-k (A+8k)
9	A+8k	A+9k	-k	valid	no prediction	-k (A+9k)
10	A+9k	A+10k	-k	valid	no prediction	-k (A+10k)
11	A+10k	0 (NULL)	A+10k	not valid	-k (A+11k)	-k (A+11k)
12	A+11k	0 (NULL)	A+11k	not valid	no prediction	-k (A+12k)
13	A+12k	A+13k	-k	valid	no prediction	-k (A+13k)
14	A+13k	0 (NULL)	A+13k	not valid	no prediction	-k (A+14k)
15	A+14k	0 (NULL)	A+14k	not valid	no prediction	-k (A+15k)

(i.e., $\text{accuracy} = 33\%$). In contrast, if the AVD predictor is not updated on a NULL data value, it would generate a prediction for 13 dynamic instances ($\text{coverage} = 87\%$), five of which are correct ($\text{accuracy} = 38\%$).[11] Hence, not updating the AVD predictor on NULL data values significantly increases the coverage without degrading the accuracy of the predictor since the AVD for Load 1 is stable except when it loads a NULL pointer.

For benchmarks similar to `treeadd` where short regular traversals frequently terminated by NULL pointer loads are common, not updating the AVD predictor on a NULL data value would be useful. NULL-value optimization requires that a NULL data value be detected by the predictor. Thus, the update logic of the AVD predictor needs to be augmented with a simple comparator to zero (zero checker).

Fig. 8 shows the impact of using NULL-value optimization on the execution time of the evaluated benchmarks. NULL-value optimization significantly improves the execution time of `treeadd` (by 41.8 percent versus the 17.6 percent improvement when confidence is reset on NULL values) and does not significantly impact the performance of other benchmarks. On average, it increases the execution time improvement of a 16-entry AVD predictor from 12.1 percent to 14.3 percent (from 5.1 percent to 7.5 percent excluding health), mainly due to the improvement in `treeadd`.

To provide insight into the performance improvement in `treeadd`, Figs. 9 and 10 show the coverage and accuracy of AVD predictions for L2-miss address loads. Not updating the AVD predictor on NULL values increases the coverage of the predictor from 50 percent to 95 percent in `treeadd` while also slightly increasing its accuracy. For most other benchmarks, the AVD prediction coverage also increases with the NULL-value optimization; however, the AVD prediction accuracy decreases. Therefore, the proposed NULL-value optimization does not provide a significant performance benefit in most benchmarks.[12]

11. Even though many of the predicted AVDs are incorrect in the latter case, the predicted values are later used as addresses by the same load instruction. Thus, AVD prediction can provide prefetching benefits even if the predicted AVDs may not be correct.

12. In some benchmarks, encountering a NULL pointer actually coincides with the end of a stable AVD pattern. Not updating the AVD predictor on NULL values in such cases increases coverage but reduces accuracy.

7.2 Optimizing the Source Code to Take Advantage of AVD Prediction

As evident from the code examples shown in Section 3, the existence of stable AVDs highly depends on the existence of regular memory allocation patterns arising from the way programs are written. We demonstrate how increasing the regularity in the allocation patterns of linked data structures —by modifying the application source code—increases the effectiveness of AVD prediction on a runahead processor. To do so, we use the source code example from the `parser` benchmark that was explained in Section 3.2 and Fig. 3.

In the `parser` benchmark, stable AVDs for Load 1 in Fig. 3 occur because the distance in memory of a `string` and its associated `Dict_node` is constant for many nodes in the dictionary. As explained in Section 3.2, the distance in memory between a `string` and its associated `Dict_node` depends on the size of the `string` because the `parser` benchmark allocates memory space for `string` first and `Dict_node` next. If the allocation order for these two structures is reversed (i.e., if space for `Dict_node` is allocated first and `string` next), the distance in memory between `string` and `Dict_node` would no longer be dependent on the size of the `string`, but it would be dependent on the size of `Dict_node`. Since the size of the data structure `Dict_node` is constant, the distance between `string` and `Dict_node` would always be constant. Such an optimization in the allocation order would therefore increase the stability of the AVDs of Load 1. Fig. 11b shows

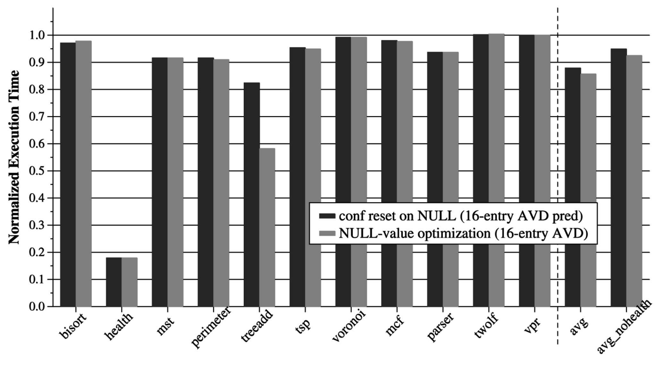


Fig. 8. AVD performance with and without NULL-value optimization.

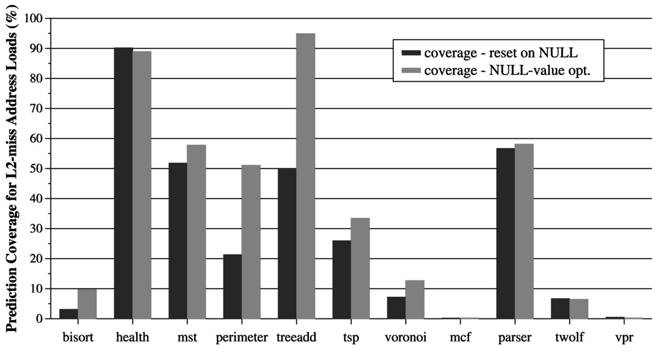


Fig. 9. Effect of NULL-value optimization on AVD prediction coverage.

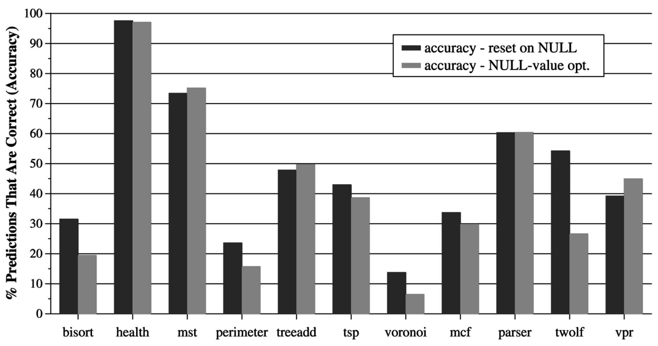


Fig. 10. Effect of NULL-value optimization on AVD prediction accuracy.

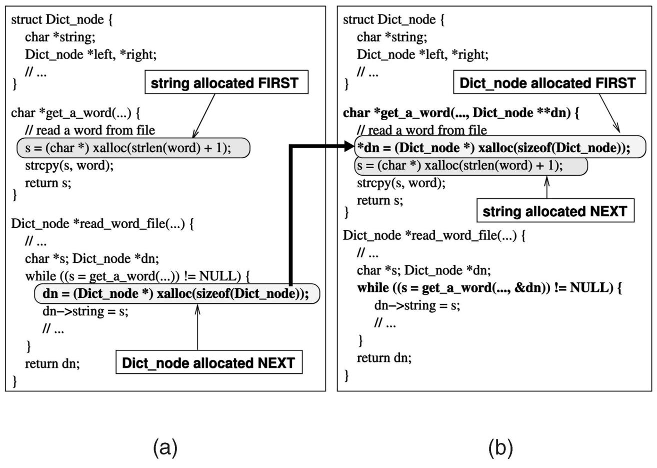
the modified source code that allocates memory space for `Dict_node` first and `string` next. Note that this optimization requires only three lines to be modified in the original source code of the `parser` benchmark.

Fig. 12 shows the execution time of the baseline `parser` binary and the modified `parser` binary on a runahead processor with and without AVD prediction support. The performances of the baseline and modified binaries are the same on the runahead processor that does not implement AVD prediction, indicating that the code modifications shown in Fig. 11 do not significantly change the performance of `parser` on the baseline runahead processor. However, when run on a runahead processor with AVD prediction, the modified binary outperforms the base binary by 4.4 percent. Hence, this very simple source code optimization significantly increases the effectiveness of AVD prediction by taking advantage of the way AVD prediction works.

Fig. 13 shows the AVD prediction coverage and accuracy for L2-miss address loads on the baseline binary and the modified binary. The described source code optimization increases the accuracy of AVD prediction from 58 percent to 83 percent. Since the modified binary has more regularity in its memory allocation patterns, the resulting AVDs for Load 1 are more stable than in the baseline binary. Hence the increase in AVD prediction accuracy and performance.

8 Interaction of AVD Prediction with Other Techniques

A runahead processor will likely incorporate other techniques that interact with AVD prediction, such as techniques for efficient runahead processing and stream-based hardware data prefetching. Some of the benefits provided



```
struct Dict_node {
   char *string;
   Dict_node *left, *right;
   // ...
}

char *get_a_word(...) {
   // read a word from file
   s = (char *) xalloc(strlen(word) + 1);
   strcpy(s, word);
   return s;
}

Dict_node *read_word_file(...) {
   // ...
   char *s; Dict_node *dn;
   while ((s = get_a_word(...)) != NULL) {
      dn = (Dict_node *) xalloc(sizeof(Dict_node));
      dn->string = s;
      // ...
   }
   return dn;
}
```

```
struct Dict_node {
   char *string;
   Dict_node *left, *right;
   // ...
}

char *get_a_word(..., Dict_node **dn) {
   // read a word from file
   *dn = (Dict_node *) xalloc(sizeof(Dict_node));
   s = (char *) xalloc(strlen(word) + 1);
   strcpy(s, word);
   return s;
}
Dict_node *read_word_file(...) {
   // ...
   char *s; Dict_node *dn;
   while ((s = get_a_word(..., &dn)) != NULL) {
      dn->string = s;
      // ...
   }
   return dn;
}
```

Fig. 11. Source code optimization performed in `parser`. (a) Base source code that allocates the nodes of the dictionary (binary tree) and the strings. (b) Modified source code (modified lines are in bold).

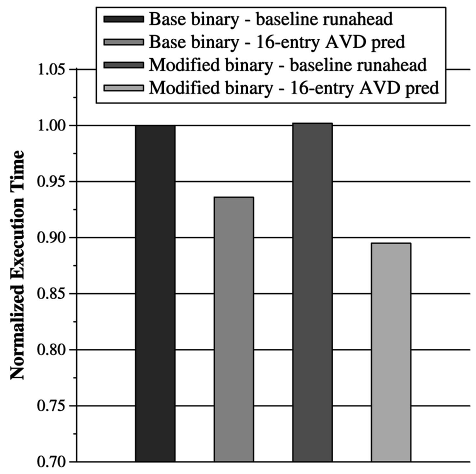


Fig. 12. Effect of source code optimization on AVD prediction performance in `parser`.

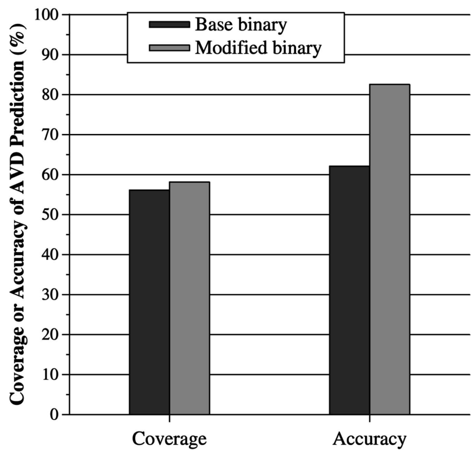


Fig. 13. Effect of source code optimization on AVD prediction coverage and accuracy in `parser`.

by these mechanisms can be orthogonal to the benefits provided by AVD prediction, some not. We examine two such mechanisms that were previously proposed in the literature and analyze their interactions with AVD prediction in a runahead processor.

TABLE 5
Evaluated Runahead Efficiency Techniques

Short runahead period elimination	Processor does not initiate runahead on an L2 miss that has been in flight for more than T=400 cycles.
Overlapping runahead period elimination	Not implemented. We found overlapping periods to be useful for performance in the benchmark set examined.
Useless runahead period elimination	1. 64-entry, 4-way RCST to eliminate useless periods based on the usefulness history of a load instruction
	2. Exit runahead mode if 75% of the load instructions executed is INV after 50 cycles in runahead mode
	3. Sampling: If the last N=100 runahead periods caused less than T=5 L2 cache misses, do not initiate runahead for the next M=1000 L2 cache misses

8.1 Interaction of AVD Prediction with Efficiency Techniques for Runahead Execution

Several techniques have been proposed to increase the efficiency of a runahead processor [24]. The efficiency of a runahead processor is defined as:

$$Efficiency = \frac{\textit{Percent Increase In Performance Due To Runahead}}{\textit{Percent Increase In Executed Instructions Due To Runahead}}.$$

Previously proposed efficiency techniques improve runahead efficiency by eliminating *short*, *overlapping*, and otherwise *useless* runahead periods without significantly reducing the performance improvement provided by runahead execution. In essence, these techniques predict whether or not a runahead period is going to be useful (i.e., will generate a useful L2 cache miss). If the runahead period is predicted to be useless, entry into runahead mode is disabled.

In contrast, AVD prediction improves the efficiency of a runahead processor by increasing the usefulness of runahead periods (either by turning a useless runahead period into a useful one or by increasing the usefulness of an already useful runahead period). Since AVD prediction and runahead efficiency techniques improve runahead efficiency in different ways, we would like to combine these two approaches and achieve even further improvements in runahead efficiency.

We have evaluated the runahead efficiency techniques proposed in [24] alone and in conjunction with AVD prediction. Table 5 lists the implemented techniques and the threshold values used in the implementation. For a thorough description of each technique, we refer the reader to [24].

Figs. 14 and 15 show, respectively, the normalized execution time and the normalized number of executed instructions when AVD prediction and efficiency techniques are utilized individually and together. We assume that NULL-value optimization is employed in the AVD predictor. In general, efficiency techniques are very effective at reducing the number of executed instructions. However, they also result in a slight performance loss. On average, using the efficiency techniques results in a 30 percent reduction in executed instructions accompanied by a 2.5 percent increase in execution time on the baseline runahead processor.

Compared to the efficiency techniques, AVD prediction is less effective in reducing the number of executed instructions. However, AVD prediction *increases* the baseline runahead performance while also reducing the executed instructions. On average, using a 16-entry AVD predictor results in a 15.5 percent reduction in executed instructions accompanied by a 14.3 percent reduction in execution time.

Using AVD prediction in conjunction with the previously proposed efficiency techniques further improves efficiency by *both* reducing the number of instructions *and*, at the same time, increasing performance. When AVD prediction and efficiency techniques are used together in the baseline runahead processor, a 35.3 percent reduction in executed instructions is achieved accompanied by a 10.1 percent decrease in execution time. Hence, AVD prediction and the previously proposed efficiency techniques are complementary to each other and they interact positively.

Fig. 16 shows the normalized number of runahead periods using AVD prediction and efficiency techniques. Efficiency techniques are more effective in eliminating useless runahead periods than AVD prediction. Efficiency techniques alone reduce the number of runahead periods by 53 percent on average. Combining AVD prediction and efficiency techniques eliminates 57 percent of all runahead periods and the usefulness of already useful runahead periods also increases.

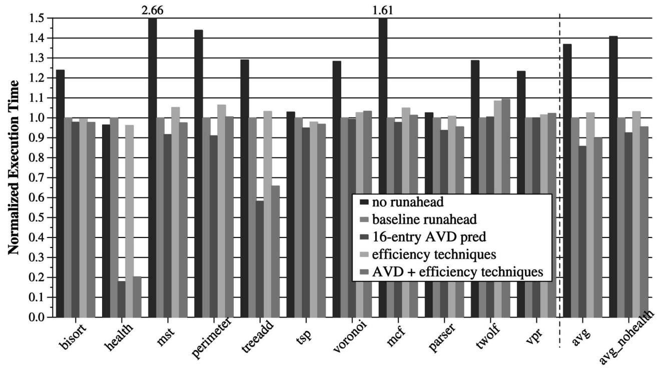


Fig. 14. Performance with AVD prediction and runahead efficiency techniques.

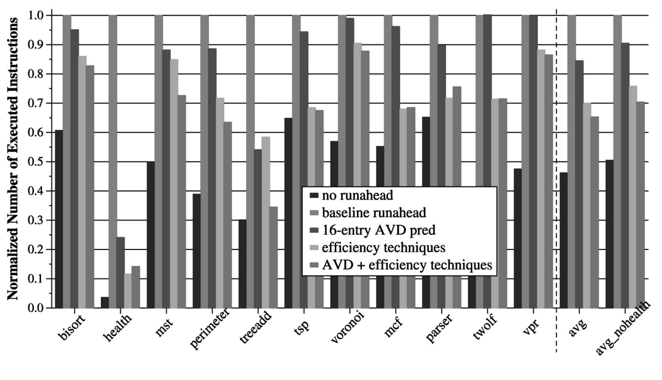


Fig. 15. Executed instructions with AVD prediction and runahead efficiency techniques.

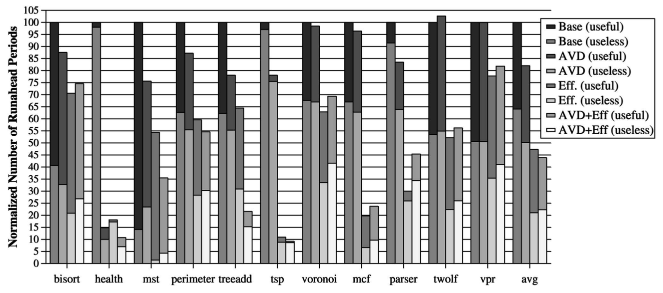


Fig. 16. Useful/useless runahead periods with AVD prediction and runahead efficiency techniques.

We conclude that using both AVD prediction and efficiency techniques together provides a better efficiency-performance trade-off than using either of the mechanisms alone. Therefore, an efficient runahead processor should probably incorporate both AVD prediction and runahead efficiency techniques.

8.2 Interaction of AVD Prediction with Stream-Based Prefetching

Stream-based prefetching [14] is a technique that identifies regular streaming patterns in the memory requests generated by a program. Once a streaming pattern is identified, the stream prefetcher generates speculative memory requests for later addresses in the identified stream. We compare the performance benefits and bandwidth requirements of an AVD predictor and an aggressive state-of-the-art stream-based prefetcher along with a combination of both techniques. The experiments in this section assume that the AVD predictor implements the NULL-value optimization described in Section 7.1.

The stream prefetcher we model is similar to the IBM Power 4 prefetcher described by Tendler et al. [35]. We evaluate two configurations of the same prefetcher: an aggressive one with a prefetch distance of 32 (i.e., a prefetcher that can stay 32 cache lines ahead of the processor's access stream) and a relatively conservative one with a prefetch distance of 8. Both configurations have 32 stream buffers. We found that increasing the number of stream buffers beyond 32 provides negligible benefits. A stream buffer is allocated on an L2 cache miss. The stream buffers are trained with L2 cache accesses. A generated prefetch request first queries the L2 cache. If it misses, it generates a memory request. Prefetched cache lines are inserted into the L2 cache.

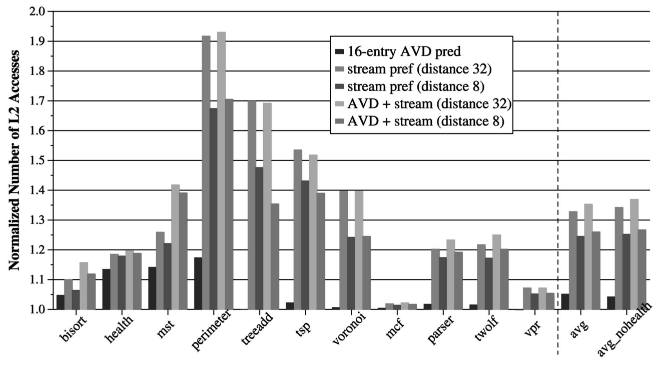


Fig. 18. Increase in L2 accesses due to AVD prediction and stream prefetching.

Fig. 17 shows the execution time improvement when AVD prediction and stream prefetching are employed individually and together on the baseline runahead processor. Figs. 18 and 19, respectively, show the increase in the number of L2 accesses and main memory accesses when AVD prediction and stream prefetching are employed individually and together. On average, the stream prefetcher with a prefetch distance of 32 improves the average execution time of the evaluated benchmarks by 16.5 percent (18.1 percent when `health` is excluded) while increasing the number of L2 accesses by 33.1 percent and main memory accesses by 14.9 percent. A prefetch distance of 8 provides an average performance improvement of 13.4 percent (14.8 percent excluding `health`) and results in a 25 percent increase in L2 accesses and a 12.2 percent increase in memory accesses. In contrast, a 16-entry AVD predictor improves the average execution time of the evaluated benchmarks by 14.3 percent (7.5 percent excluding `health`) while increasing the number of L2 accesses by only 5.1 percent and main memory accesses by only 3.2 percent. Hence, AVD prediction is much less bandwidth-intensive than stream prefetching, but it does not provide as much performance improvement.

Using AVD prediction and stream prefetching together on a runahead processor improves the execution time by more than either of the two techniques does alone. This shows that the two techniques are in part complementary.

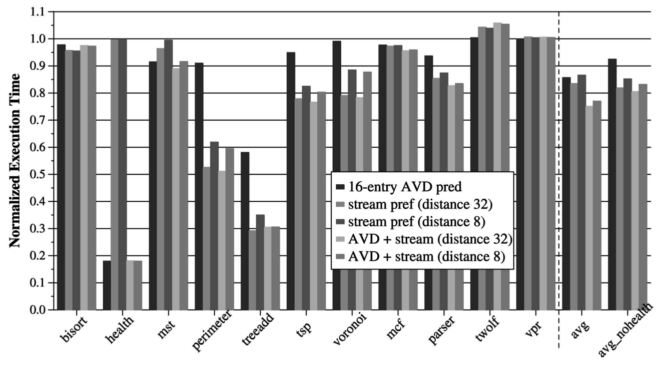


Fig. 17. AVD prediction versus stream prefetching.

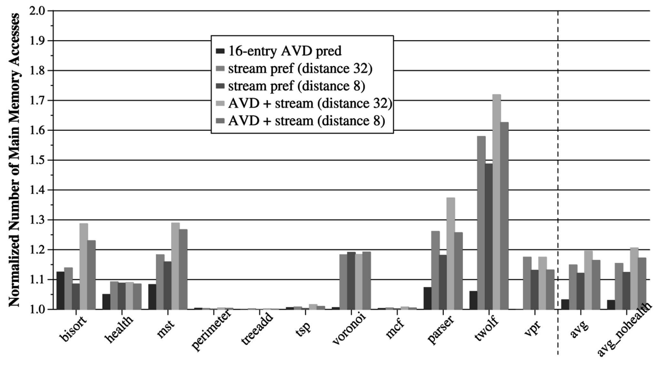


Fig. 19. Increase in memory accesses due to AVD prediction and stream prefetching.

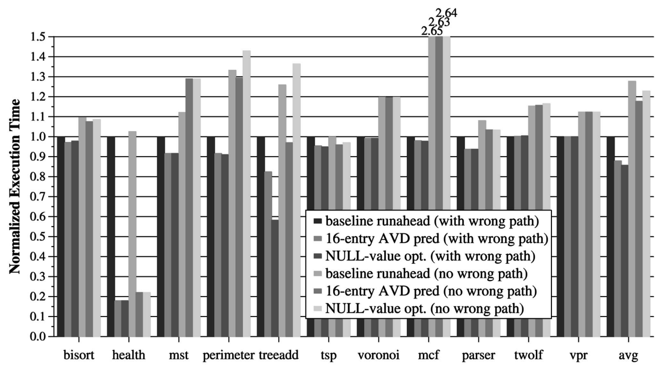


Fig. 20. Impact of wrong-path modeling on AVD prediction and NULL-value optimization performance.

Using a 16-entry AVD predictor and a stream prefetcher with distance 32 improves the average execution time by 24.9 percent (19.5 percent excluding `health`) while increasing the L2 accesses by 35.3 percent and main memory accesses by 19.5 percent.

In general, AVD prediction is limited to prefetching the addresses of dependent load instructions, whereas a stream prefetcher can prefetch addresses generated by both dependent and independent load instructions. Therefore, a stream prefetcher can capture a broader range of address patterns that are of a streaming nature. A traversal address load with a stable AVD (in this case, also a regular stride) results in a streaming memory access pattern. Hence, similarly to an AVD predictor, a stream prefetcher can prefetch the addresses generated by a traversal address load with a constant AVD.

In contrast to an AVD predictor, a stream prefetcher can capture the addresses generated by a leaf address load with a stable AVD and its dependent instructions *only if* those addresses form a streaming access pattern or are part of a streaming access pattern. An AVD predictor is therefore more effective in predicting the addresses dependent on leaf address loads with stable AVDs. For this very reason, AVD prediction significantly improves the performance of two benchmarks, `health` and `mst`, for which the stream prefetcher is ineffective.

8.3 Importance of Correctly Modeling Wrong Path on Performance Estimates for AVD Prediction

In a recent paper [22], we have shown that modeling wrong-path memory references is crucial to getting an accurate estimate of the performance improvement provided by runahead execution. We hereby examine the impact of modeling wrong-path references on the estimates of the performance improvements provided by AVD prediction. We show that the modeling of wrong-path (i.e., execution-driven simulation as opposed to trace-driven simulation) is necessary to get accurate performance estimates of AVD prediction and—perhaps more importantly—to accurately determine the effect of some AVD predictor optimizations.

Fig. 20 shows the normalized execution time of three processors when wrong-path execution is modeled versus not modeled: the baseline runahead processor, baseline processor with a 16-entry AVD predictor, and the baseline processor with a 16-entry AVD predictor with the NULL-value optimization. If wrong-path execution is not modeled in the simulator, the performance improvement of AVD prediction (without NULL-value optimization) is significantly underestimated as 8.1 percent (versus 12.1 percent when wrong-path execution is modeled). This is because runahead execution on the wrong path provides prefetching benefits for the normal mode execution on the correct path, as described in [22]. Furthermore, when wrong-path execution is not modeled, NULL-value optimization seems to degrade performance, whereas it increases performance with accurate modeling of wrong-path execution. Hence, not modeling wrong-path execution can cause the processor designer to make a wrong choice in the design of an AVD predictor.

TABLE 6
Less Aggressive Baseline Processor Configuration

Pipe depth	17 stages, 14-cycle min. branch mispred. penalty
Pipe width	4 instructions per cycle fetch/issue/exec/retire
Branch Predictors	16K-entry gshare/per-address hybrid, 4K-entry selector; 2K-entry, 4-way BTB; 1K-entry indirect target cache;
Instruction Window	64-entry reorder buffer; 64-entry INT physical reg. file 64-entry FP physical register file; 48-entry ld/st buffer
Memory	maximum 16 outstanding misses to main memory

8.4 Effect of AVD Prediction on a Less Aggressive Processor

We also evaluate AVD prediction on a less aggressive baseline processor. Those parameters of the less aggressive baseline processor that are different from our aggressive baseline processor are shown in Table 6.

Fig. 21 shows the performance improvement provided by a 16-entry AVD predictor (with NULL-value optimization) in comparison to and in conjunction with a 16-entry SVP or a stream prefetcher with 16 stream buffers and a prefetch distance of 16. The AVD predictor improves the average execution time by 12.9 percent (6.3 percent excluding `health`), whereas SVP improves the average execution time by 2.0 percent (2.2 percent excluding health). Furthermore, combining SVP or stream-based prefetching with AVD prediction improves the performance more than either of the two mechanisms alone. Thus, AVD prediction remains effective on a less aggressive processor (because dependent cache misses limit the performance of a less aggressive processor as well as a more aggressive one) and

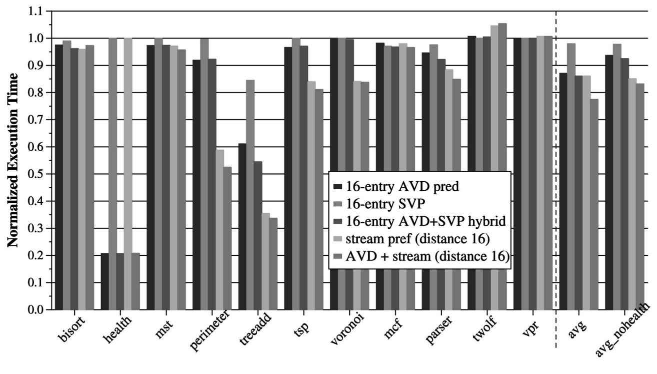


Fig. 21. AVD prediction performance on the less aggressive processor.

its performance benefits complement the benefits of SVP and stream-based prefetching.

9 Related Work

Several previous papers focused on predicting the values/addresses generated by pointer loads for value prediction or prefetching purposes. Most of the proposed mechanisms we are aware of require significant storage cost and hardware complexity. The major contribution of our study is a simple and efficient novel mechanism that allows the prediction of the values loaded by a subset of pointer loads by exploiting stable address-value relationships.

Section 9.1 briefly discusses the related research in value and address prediction for load instructions. Section 9.2 describes the related work in prefetching for pointer loads. And, Section 9.3 gives a brief overview of the related work in runahead execution.

9.1 Related Research in Load Value/Address Prediction

The work most relevant to our research is in the area of predicting the destination register values of load instructions. Load value prediction [20], [31] was proposed to predict the destination register values of loads. Many types of load value predictors were examined, including last value [20], stride [10], [31], FCM (finite context method) [31], and hybrid [36] predictors. While a value predictor recognizes stable/predictable values, an AVD predictor recognizes stable address-value deltas. As shown in code examples in Section 3, the address-value delta for an address load instruction can be stable and predictable even though the value of the load instruction is not predictable. Furthermore, small value predictors do not significantly improve performance, as shown in Section 6.1.

Load address predictors [10], [2] predict the effective address of a load instruction early in the pipeline. The value at the predicted address can be loaded to the destination register of the load before the load is ready to be executed. Memory latency can be partially hidden for the load and its dependent instructions.

Complex (e.g., stride or context-based) value/address predictors need significant hardware storage to generate predictions and significant hardware complexity for state recovery. Moreover, the update latency (i.e., the latency between making the prediction and determining whether or not the prediction was correct) associated with stride and context-based value/address predictors significantly detracts from the performance benefits of these predictors over simple last value prediction [27], [18]. Good discussions of the hardware complexity required for complex address/value prediction can be found in [2] and [27].

The pointer cache [7] was proposed to predict the values of pointer loads. A pointer cache caches the values stored in memory locations accessed by pointer load instructions. It is accessed with a load's effective address in parallel with the data cache. A pointer cache hit provides the predicted value for the load instruction. To improve performance, a pointer cache requires significant hardware storage (at least 32K entries where each entry is 36 bits [7]) because the pointer data sets of the programs are usually large. In contrast to the pointer cache, an AVD predictor stores AVDs based on pointer load instructions. Since the pointer load instruction working set of a program is usually much smaller than the pointer data working set, the AVD predictor requires much less hardware cost. Also, an AVD predictor does not affect the complexity in critical portions of the processor because it is small and does not need to be accessed in parallel with the data cache.

Zhou and Conte [41] proposed the use of value prediction only for prefetching purposes in an out-of-order processor such that no recovery is performed in the processor on a value misprediction. They evaluated their proposal using a 4K-entry stride value predictor which predicts the values produced by all load instructions. Similarly to their work, we employ the AVD prediction mechanism only for prefetching purposes, which eliminates the need for processor state recovery.

9.2 Related Research in Pointer Load Prefetching

In recent years, substantial research has been performed in prefetching the addresses generated by pointer load instructions. AVD prediction differs from pointer load prefetching in that it is *more than just a prefetching mechanism*. As shown in [23], AVD prediction can be used for simple prefetching. However, AVD prediction is more beneficial when it is used as a targeted value prediction technique for pointer loads that enables the preexecution of dependent load instructions, which may generate prefetches.

Hardware-based pointer prefetchers [4], [13], [29], [30], [7] try to dynamically capture the prefetch addresses generated by traversal loads. These approaches usually require significant hardware cost to store a history of pointers. For example, hardware-based jump pointer prefetching requires jump pointer storage that has more than 16K entries (64KB) [30]. A low-overhead content-based hardware pointer prefetcher was recently proposed by Cooksey et al. [8]. It can be combined with AVD prediction to further reduce the negative performance impact of dependent L2 cache misses.

Software and combined software/hardware methods have also been proposed for prefetching loads that access linked data structures [19], [21], [30], [40], [15], [33], [5], [38], [1]. Most relevant to AVD prediction, one software-based prefetching technique, MS Delta [1], uses the garbage collector in a runtime managed Java system to detect regular distances between objects in linked data structures whose traversals result in significant number of cache misses. A just-in-time compiler inserts prefetch instructions into the program using the identified regular distances in order to prefetch linked objects in such traversals. Such software-based prefetching techniques require nontrivial support from the compiler, the programmer, or a dynamic optimization and compilation framework. Existing binaries cannot utilize software-based techniques unless they are recompiled or reoptimized using a dynamic optimization framework. AVD prediction, on the contrary, is a purely hardware-based mechanism that does not require any software support and, thus, it can improve the performance of existing binaries. However, as we have shown in Section 7.2, AVD prediction can provide larger performance improvements if software is written or optimized to increase the occurrence of stable AVDs.

9.3 Related Research in Runahead Execution

Chou et al. [6] proposed combining runahead execution with value prediction. The Clear [16] and CAVA [3] microarchitectures predict the values of L2-miss load instructions, speculatively execute instructions without stalling due to L2 cache misses (similarly to runahead execution), and commit the results of speculatively executed instructions to the architectural state in case all previous value predictions are correct. These techniques use conventional value predictors to predict the values of *all* L2-miss load instructions during preexecution, which requires significant hardware support (at least 2K-entry value tables). In contrast, we propose a new predictor to predict the values of only L2-miss address loads, which allows the parallelization of dependent cache misses without significant hardware overhead. As mentioned in [16], predicting the values of all L2-miss instructions during runahead mode sometimes reduces the performance of a runahead processor since instructions dependent on the value-predicted loads need to be executed and they slow down the processing speed during runahead mode. Our goal in this paper is to selectively predict the values of only those load instructions that can lead to the generation of costly dependent cache misses. We note that the AVD prediction mechanism is not specific to runahead execution and can also be employed by conventional processors.

We [24] proposed techniques for increasing the efficiency of runahead execution. These techniques usually increase efficiency by eliminating *useless* runahead periods. In contrast, AVD prediction increases both efficiency and performance by increasing the usefulness of runahead periods. As we have shown in Section 8.1, runahead efficiency techniques are orthogonal to AVD prediction and combining them with AVD prediction yields larger improvements in runahead efficiency.

10 Summary, Contributions, and Future Work

This paper introduces the concept of *stable address-value deltas (AVDs)* and proposes *AVD prediction*, a new method of predicting the values generated by address load instructions by exploiting the stable and regular memory allocation patterns in programs that heavily utilize linked data structures. We provide insights into why stable AVDs exist through code examples from pointer-intensive applications. We also describe the design and implementation of a simple AVD predictor and utilize it to overcome an important limitation of runahead execution: its inability to parallelize dependent L2 cache misses. The proposed AVD prediction mechanism requires neither significant hardware cost or complexity nor hardware support for state recovery. We propose hardware and software optimizations that increase the detection and occurrence of stable AVDs, which can significantly improve the benefits of AVD prediction. Our experimental results show that a simple AVD predictor can significantly improve both the performance and efficiency of a runahead execution processor. Augmenting a runahead execution processor with a small, 16-entry (102-byte) AVD predictor improves the average execution time of a set of pointer-intensive applications by 14.3 percent while it also reduces the executed instructions by 15.5 percent. Our experiments also show that AVD prediction interacts positively with two previously proposed mechanisms, efficiency techniques for runahead execution and stream-based data prefetching.

10.1 Contributions

The major contribution of our paper is a simple and efficient mechanism that allows the prediction of the values loaded by a subset of pointer loads by exploiting stable address-value relationships. Other contributions we make in this paper are:

1. We introduce the concept of stable *address-value deltas* and provide an analysis of the code structures that cause them through code examples from application programs.
2. We propose the design and implementation of a simple, low-hardware-cost predictor that exploits the stable AVDs. We evaluate the design options for an AVD predictor. We also propose and evaluate simple hardware and software optimizations for an implementable AVD predictor.
3. We describe an important limitation of runahead execution: its inability to parallelize dependent long-latency cache misses. We show that this limitation can be reduced by utilizing a simple AVD predictor in a runahead execution processor.
4. We evaluate the interactions of AVD prediction with two previously proposed mechanisms: efficiency techniques for runahead execution and stream-based data prefetching. We show that AVD prediction interacts positively with these two related mechanisms.

10.2 Future Research Directions

Future work in exploiting stable AVDs can proceed in multiple directions. First, the AVD predictor we presented is a simple, last-AVD predictor. More complex AVD predictors that can detect more complex patterns in address-value deltas may be interesting to study and they may further improve performance at the expense of higher hardware cost and complexity. Second, the effectiveness of AVD prediction is highly dependent on the memory allocation patterns in programs. Optimizing the memory allocator, the program structures, and the algorithms used in programs for AVD prediction can increase the occurrence of stable AVDs. Furthermore, in garbage-collected languages, optimizing the behavior of the garbage collector in conjunction with the memory allocator can increase the occurrence of stable AVDs. Hence, software (programmer/compiler/allocator/garbage collector) support can improve the effectiveness of a mechanism that exploits address-value deltas. We intend to examine software algorithms, compiler optimizations, and memory allocator (and garbage collector) optimizations that can benefit AVD prediction in our future work.

Acknowledgments

The authors thank the anonymous reviewers for their feedback. They gratefully acknowledge the Cockrell Foundation and Intel Corporation for their support. This paper is an extended and revised version of [23].

References

[1] A.-R. Adl-Tabatabai, R.L. Hudson, M.J. Serrano, and S. Subramoney, "Prefetch Injection Based on Hardware Monitoring and Object Metadata," *Proc. ACM SIGPLAN '04 Conf. Programming Language Design and Implementation,* pp. 267-276, 2004.

[2] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated Load-Address Predictors," *Proc. 26th Int'l Symp. Computer Architecture,* pp. 54-63, 1999.

[3] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas, "CAVA: Hiding L2 Misses with Checkpoint-Assisted Value Prediction," *IEEE Computer Architecture Letters,* vol. 3, Dec. 2004.

[4] M. Charney, "Correlation-Based Hardware Prefetching," PhD thesis, Cornell Univ., Aug. 1995.

[5] T.M. Chilimbi and M. Hirzel, "Dynamic Hot Data Stream Prefetching for General-Purpose Programs," *Proc. ACM SIGPLAN '02 Conf. Programming Language Design and Implementation,* pp. 199-209, 2002.

[6] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism," *Proc. 31st Int'l Symp. Computer Architecture,* pp. 76-87, 2004.

[7] J.D. Collins, S. Sair, B. Calder, and D.M. Tullsen, "Pointer Cache Assisted Prefetching," *Proc. 35th Int'l Symp. Microarchitecture,* pp. 62-73, 2002.

[8] R. Cooksey, S. Jourdan, and D. Grunwald, "A Stateless, Content-Directed Data Prefetching Mechanism," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 279-290, 2002.

[9] J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss," *Proc. 1997 Int'l Conf. Supercomputing,* pp. 68-75, 1997.

[10] R.J. Eickemeyer and S. Vassiliadis, "A Load-Instruction Unit for Pipelined Processors," *IBM J. Research and Development,* vol. 37, pp. 547-564, 1993.

[11] A. Glew, "MLP yes! ILP no!," *Wild and Crazy Idea Session, Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* Oct. 1998.

[12] M.K. Gowan, L.L. Biro, and D.B. Jackson, "Power Considerations in the Design of the Alpha 21264 Microprocessor," *Proc. 35th Ann. Design Automation Conf.,* pp. 726-731, 1998.

[13] D. Joseph and D. Grunwald, "Prefetching Using Markov Predictors," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 252-263, 1997.

[14] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Int'l Symp. Computer Architecture,* pp. 364-373, 1990.

[15] M. Karlsson, F. Dahlgren, and P. Strenstrom, "A Prefetching Technique for Irregular Accesses to Linked Data Structures," *Proc. Sixth Int'l Symp. High Performance Computer Architecture,* pp. 206-217, 2000.

[16] N. Kırman, M. Kırman, M. Chaudhuri, and J.F. Martínez, "Checkpointed Early Load Retirement," *Proc. 11th Int'l Symp. High Performance Computer Architecture,* pp. 16-27, 2005.

[17] A. KleinOsowski and D.J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *IEEE Computer Architecture Letters,* vol. 1, June 2002.

[18] S.-J. Lee and P.-C. Yew, "On Some Implementation Issues for Value Prediction on Wide-Issue ILP Processors," *Proc. 2000 Int'l Conf. Parallel Architectures and Compilation Techniques,* p. 145, 2000.

[19] M.H. Lipasti, W.J. Schmidt, S.R. Kunkel, and R.R. Roediger, "SPAID: Software Prefetching in Pointer- and Call-Intensive Environments," *Proc. 28th Int'l Symp. Microarchitecture,* pp. 232-236, 1995.

[20] M.H. Lipasti, C. Wilkerson, and J.P. Shen, "Value Locality and Load Value Prediction," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 226-237, 1996.

[21] C.-K. Luk and T.C. Mowry, "Compiler-Based Prefetching for Recursive Data Structures," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 222-233, 1996.

[22] O. Mutlu, H. Kim, D.N. Armstrong, and Y.N. Patt, "An Analysis of the Performance Impact of Wrong-Path Memory References on Out-of-Order and Runahead Execution Processors," *IEEE Trans. Computers,* vol. 54, no. 12, pp. 1556-1571, Dec. 2005.

[23] O. Mutlu, H. Kim, and Y.N. Patt, "Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns," *Proc. 38th Int'l Symp. Microarchitecture,* pp. 233-244, 2005.

[24] O. Mutlu, H. Kim, and Y.N. Patt, "Techniques for Efficient Processing in Runahead Execution Engines," *Proc. 32nd Int'l Symp. Computer Architecture,* pp. 370-381, 2005.

[25] O. Mutlu, J. Stark, C. Wilkerson, and Y.N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," *Proc. Ninth Int'l Symp. High Performance Computer Architecture,* pp. 129-140, 2003.

[26] O. Mutlu, J. Stark, C. Wilkerson, and Y.N. Patt, "Runahead Execution: An Effective Alternative to Large Instruction Windows," *IEEE Micro,* vol. 23, no. 6, pp. 20-25, Nov./Dec. 2003.

[27] P. Racunas, "Reducing Load Latency through Memory Instruction Characterization," PhD thesis, Univ. of Michigan, 2003.

[28] A. Rogers, M.C. Carlisle, J. Reppy, and L. Hendren, "Supporting Dynamic Data Structures on Distributed Memory Machines," *ACM Trans. Programming Languages and Systems,* vol. 17, no. 2, pp. 233-263, Mar. 1995.

[29] A. Roth, A. Moshovos, and G.S. Sohi, "Dependence Based Prefetching for Linked Data Structures," *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 115-126, 1998.

[30] A. Roth and G.S. Sohi, "Effective Jump-Pointer Prefetching for Linked Data Structures," *Proc. 26th Int'l Symp. Computer Architecture,* pp. 111-121, 1999.

[31] Y. Sazeides and J.E. Smith, "The Predictability of Data Values," *Proc. 30th Int'l Symp. Microarchitecture,* pp. 248-257, 1997.

[32] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 45-57, 2002.

[33] Y. Solihin, J. Lee, and J. Torrellas, "Using a User-Level Memory Thread for Correlation Prefetching," *Proc. 29th Int'l Symp. Computer Architecture,* pp. 171-182, 2002.

[34] E. Sprangle and D. Carmean, "Increasing Processor Performance by Implementing Deeper Pipelines," *Proc. 29th Int'l Symp. Computer Architecture,* pp. 25-34, 2002.

[35] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy, "POWER4 System Microarchitecture," IBM technical white paper, Oct. 2001.

[36] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction Using Hybrid Predictors," *Proc. 30th Int'l Symp. Microarchitecture,* pp. 281-290, 1997.

[37] M.V. Wilkes, "The Memory Gap and the Future of High Performance Memories," *ACM Computer Architecture News,* vol. 29, no. 1, pp. 2-7, Mar. 2001.

[38] Y. Wu, "Efficient Discovery of Regular Stride Patterns in Irregular Programs and Its Use in Compiler Prefetching," *Proc. ACM SIGPLAN '02 Conf. Programming Language Design and Implementation,* pp. 210-221, 2002.

[39] W. Wulf and S. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM Computer Architecture News,* vol. 23, no. 1, pp. 20-24, Mar. 1995.

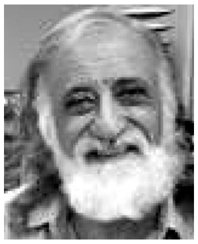
[40] C.-L. Yang and A.R. Lebeck, "Push vs. Pull: Data Movement for Linked Data Structures," *Proc. 2000 Int'l Conf. Supercomputing,* pp. 176-186, 2000.

[41] H. Zhou and T.M. Conte, "Enhancing Memory Level Parallelism via Recovery-Free Value Prediction," *Proc. 17th Int'l Conf. Supercomputing,* pp. 326-335, 2003.

[42] C.B. Zilles, "Benchmark Health Considered Harmful," *Computer Architecture News,* vol. 29, no. 3, pp. 4-5, June 2001.

Onur Mutlu received the BS degrees in computer engineering and psychology from the University of Michigan, Ann Arbor, in 2000 and the MS degree in electrical and computer engineering from the University of Texas-Austin (UT-Austin) in 2002. He received the PhD degree in electrical and computer engineering from UT-Austin in August 2006. He is currently a researcher at Microsoft Research. He is interested in broad computer architecture research, especially in the interactions between programmers, languages, operating systems, compilers, and microarchitecture. He worked at Intel Corporation during summers 2001-2003 and at Advanced Micro Devices during the summers of 2004-2005. He was a recipient of the Intel PhD fellowship in 2004 and the University of Texas George H. Mitchell Award for Excellence in Graduate Research in 2005. He is a member of the IEEE and the IEEE Computer Society.

Hyesoon Kim received the BA degree in mechanical engineering from the Korea Advanced Institute of Science and Technology (KAIST), the MS degree in mechanical engineering from Seoul National University, and the MS degree in electrical and computer engineering from the University of Texas-Austin (UT-Austin). She is a PhD candidate in electrical and computer engineering at UT-Austin. Her research interests include high-performance energy-efficient computer architectures and programmer-compiler-architecture interaction. She researched next-generation engine designs at Hyundai Motor Company Research Labs between 1998-2000. She is a student member of the IEEE and the IEEE Computer Society.



Yale N. Patt received the BS degree from Northeastern University and the MS and PhD degrees from Stanford University, all in electrical engineering. He is the Ernest Cockrell, Jr. Centennial Chair in Engineering and Professor of electrical and computer engineering at the University of Texas-Austin. He continues to thrive on teaching the large (400 students) freshman introductory course in computing and advanced graduate courses in microarchitecture, directing the research of 11 PhD students, and consulting in the microprocessor industry. He is coauthor of *Introduction to Computing Systems: From Bits and Gates to C and Beyond* (McGraw-Hill, second edition, 2004), his preferred approach to introducing freshmen to computing. He has received a number of awards for his research and teaching, including the IEEE/ACM Eckert-Mauchly Award for his research in microarchitecture and the ACM Karl V. Karlstrom Award for his contributions to education. He is a fellow of the IEEE and a member of the IEEE Computer Society. More detail can be found on his Web site, http://www.ece.utexas.edu/~patt.