# Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns

Onur Mutlu Hyesoon Kim Yale N. Patt

Department of Electrical and Computer Engineering
University of Texas at Austin
{onur,hyesoon,patt}@ece.utexas.edu

## 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 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 pre-execution of dependent instructions, including load instructions that incur long-latency cache misses. We describe how, why, and for what kind of loads AVD prediction works and evaluate the design tradeoffs in an implementable AVD predictor. Our analysis shows that stable AVDs exist because of patterns in the way data structures are allocated in memory. 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 12.1%.*

## 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 [28, 26]. 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 [29, 28]. 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 [9], simple power- and area-efficient memory latency tolerance techniques are especially desirable.

Runahead execution [6, 18] is a promising technique that was recently proposed to tolerate long main memory latencies. This technique speculatively pre-executes 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 [18], 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 pre-executed. Some of these pre-executed 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 [19, 3].

Unfortunately, a runahead execution processor cannot parallelize *dependent* long-latency cache misses. A runahead processor cannot pre-execute 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. 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 pre-processing of useless instructions.

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 dynamic techniques have been proposed to predict the values of address loads [15, 24, 1, 4] or to prefetch the addresses generated by them [22, 23, 4]. 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 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

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

Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)
0-7695-2440-0/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

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 pre-execution 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%.

## 2. Motivation

Our goal is to increase the effectiveness of runahead execution with a simple prediction mechanism that overcomes the inability to parallelize dependent long-latency cache misses during runahead mode. We demonstrate that focusing on this limitation of runahead execution has potential to improve processor performance. Figure 1 shows the potential performance improvement possible if runahead execution were able to parallelize all the *dependent long-latency cache misses* that can be generated by instructions that are pre-processed during runahead mode. This graph shows the execution time for four processors on memory- and pointer-intensive benchmarks from Olden and SPEC INT 2000 benchmark suites:[2] from left to right, (1) a processor with no runahead execution, (2) the baseline processor, which employs runahead execution, (3) an ideal runahead processor, which can parallelize dependent L2 cache misses (This processor is simulated by obtaining the correct effective address of *all* L2-miss load instructions using oracle information during runahead mode. Thus, L2 misses dependent on previous L2 misses can be generated during runahead mode using oracle information. This processor is not implementable, but it is intended to demonstrate the performance potential of parallelizing dependent L2 cache misses.), (4) a processor with perfect (100% hit rate) L2 cache. Execution times are normalized to the baseline processor. The baseline runahead processor improves the average execution time of the processor with no runahead execution by 27%. The ideal runahead processor improves the average execution time of the baseline runahead processor by 25%, showing that significant performance potential exists for techniques that enable the parallelization of dependent L2 misses. Table 1, which shows the average number of L2 cache misses initiated during runahead mode, provides insight into the performance improvement possible with the ideal runahead processor. This table shows that the ideal runahead processor significantly increases the memory-level parallelism (the number of useful L2 cache misses parallelized[3]) in a runahead period.

Figure 1 also shows that for two benchmarks (`health` and `tsp`) runahead execution is ineffective. These two benchmarks have particularly low levels of memory-level parallelism, since their core algorithms consist of traversals of linked data structures in which almost all load instructions are dependent on previous load instructions. Due to the scarcity of independent long-latency cache misses (as shown in Table 1), conventional runahead execution cannot significantly improve the performance of `health` and `tsp`. In fact, the overhead of runahead execution results in 4% performance loss on `health`. In contrast, the ideal runahead processor provides significant performance improvement on these two benchmarks (88% on `health` and 32% on `tsp`), alleviating the ineffectiveness of conventional runahead execution.

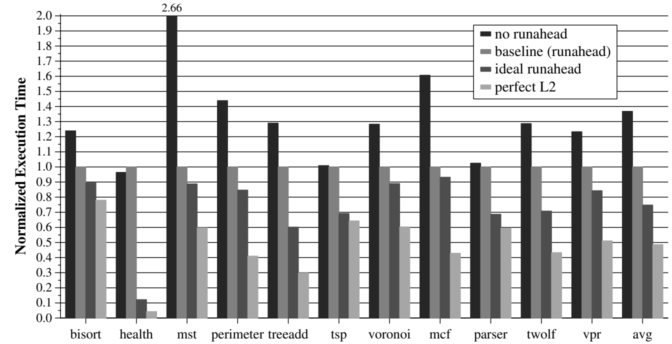


**Figure 1. Performance potential of runahead execution.**

## 3. 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:

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

Figure 2 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.

Figure 2b 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 of Load 1 is predicted by subtracting the AVD returned by the predictor from the effective address of Load 1 such that:

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

[2] Section 6 describes the processor model and the benchmarks.

[3] 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.

Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)
0-7695-2440-0/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

**Table 1. Average number of useful L2 cache misses generated (parallelized) during a runahead period.**

| | bisort | health | mst | perimeter | treeadd | tsp | voronoi | mcf | parser | twolf | vpr | avg |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |
| ideal runahead | 4.58 | 8.43 | 8.77 | 2.06 | 2.87 | 4.42 | 1.43 | 12.75 | 1.56 | 2.79 | 1.19 | 4.62 |

```
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) Code example

| 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) Execution history of Load 1

**Figure 2. Source code example showing a load instruction with a stable AVD (Load 1) and its execution history.**

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 Figure 2 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.[4] Note that this mechanism is similar to the mechanism proposed by Cooksey et al. [5] 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.

# 4. 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 the 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, `p = p->next` (e.g., Load 3 in Figure 2 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 Figure 2 is a leaf address load).

[4] 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.

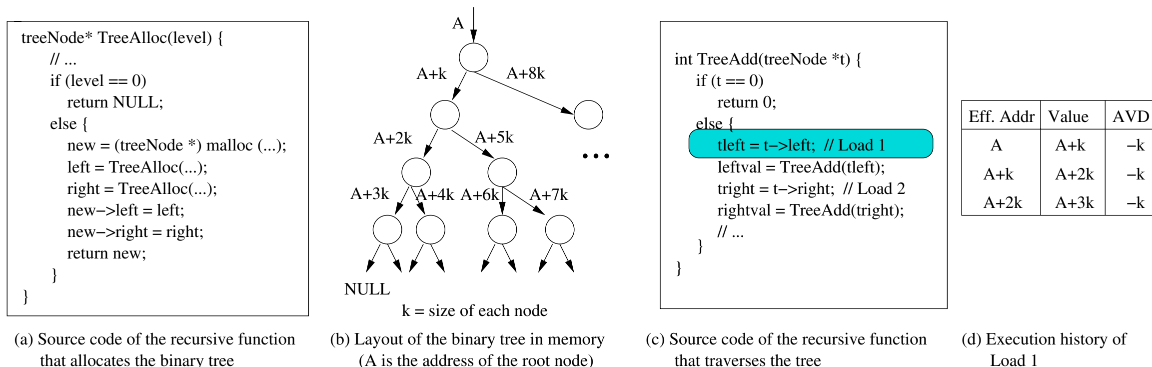
## 4.1. Stable AVDs in Traversal Address Loads

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.

Figure 3 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 (Figure 3a). Each node of the tree is of the same size. The layout of an example resulting binary tree is shown in Figure 3b. 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 (Figure 3c). Load 1 in the traversal function traverses the nodes by loading the pointer to the left child of each node. This load instruction has a stable AVD as can be seen from its example execution history (Figure 3d). 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% 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. Similar traversal loads with stable AVDs exist in `twolf`, `mst`, and `vpr`, which employ linked lists, and `bisort`, `perimeter`, `tsp`, and `voronoi`, which employ binary- or quad-trees.

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 alloca-


IEEE COMPUTER SOCIETY




(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

**Figure 3. An example from `treeadd` showing how stable AVDs can occur for traversal address loads.**

tor 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 the regularity in the allocation and linking of the nodes can disappear if the linked data structure is significantly re-organized during run-time, unless the re-organization 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.

### 4.2. 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. We show two examples to illustrate this behavior.

Figure 4 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 Figure 4a. 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 Figure 4b. 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 Figure 4) 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.[5]

Words are later looked up in the dictionary using the `rabridged_lookup` function (Figure 4c). 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 node and its associated string is constant. The values generated by Load 1 are hard to predict using a traditional 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% of the entries into runahead mode and correctly predicting its value enables the execution of the dependent load instructions in the `dict_match` function.

Note that stable AVDs occurring in leaf address loads continue to be stable even if the linked data structure is significantly re-organized at run-time. 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 re-organization of the linked data structure changes the links between nodes and other nodes, but leaves intact the links between nodes and their fields.

Figure 5 shows an example from `health`, demonstrating the occurrence of stable AVDs in a linked list. This benchmark simulates a health care system in which a list of patients waiting to be serviced is maintained in a linked list. Each node of the linked list contains a pointer to the `patient` structure it is associated with. Each node and the `patient` structure are allocated dynamically as shown in Figure 5a. The allocation of these structures is performed in a regular fashion. First, memory space for a `patient` is allocated. Right after that, memory space for a `List_node` is allocated and it is linked to the `patient` via a pointer. Since `List_node` and `Patient` structures are of fixed size, the distance in memory between a node and its associated `patient` is constant as shown in the layout of the resulting linked list (Figure 5b). The linked list is later traversed in the `check_patients_waiting` function (Figure 5c). The `patient` associated with each node is loaded by Load 1 (`p = list->patient`), which is a leaf address load that is later dereferenced by a data load, Load 2 (`t = p->time`). Load 1 has a stable AVD as shown in its execution history. It causes 67% of the entries into runahead mode and predicting its value correctly enables the servicing of dependent L2 misses caused by Load 2.

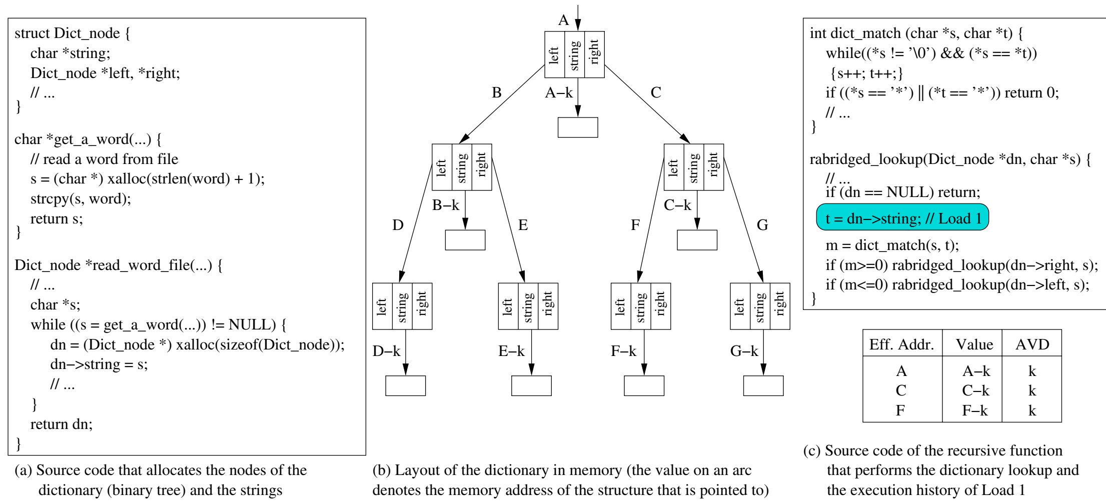
## 5. Design and Operation of a Recovery-Free AVD Predictor

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

[5]The code shown in Figure 4a can be re-written 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 evaluations.


0-7695-2440-0/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

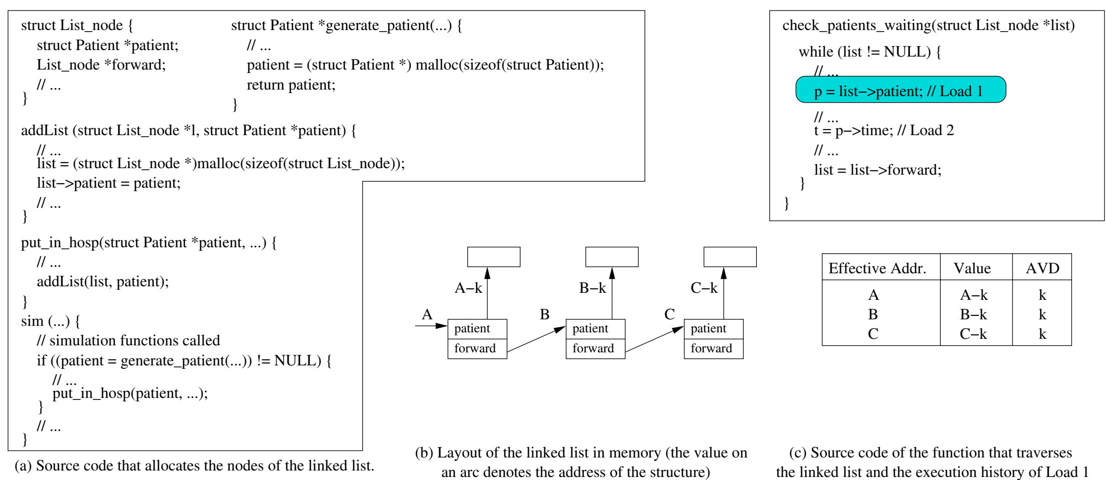



(a) Source code that allocates the nodes of the dictionary (binary tree) and the strings

(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

**Figure 4. An example from `parser` showing how stable AVDs can occur for leaf address loads.**




(a) Source code that allocates the nodes of the linked list.

(b) Layout of the linked list in memory (the value on an arc denotes the address of the structure)

(c) Source code of the function that traverses the linked list and the execution history of Load 1

**Figure 5. An example from `health` showing how stable AVDs can occur for leaf address loads.**

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.

Figure 6 shows the organization of the AVD predictor along with the hardware support needed to update/train it (Figure 6a) and the hardware support needed to make a prediction (Figure 6b). 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; *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.

## 5.1. 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.[6]

The predictor is accessed when a load instruction misses in the L2 cache during runahead mode. The predictor is accessed with the

[6] 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 re-train the predictor and thus helps benchmarks where loads that perform short traversals are common.


0-7695-2440-0/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

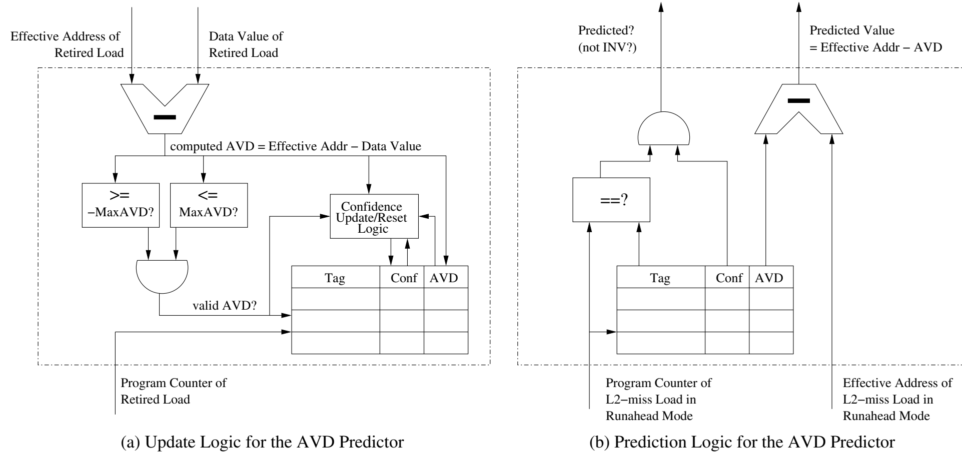



**Figure 6. Organization of the AVD predictor and the hardware support needed for updating/accessing the predictor.**

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 [18]) 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 anyway be updated when the load is re-executed 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[7], 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.

[7]i.e., runahead mode makes no changes to the architectural state of the processor.

## 5.2. 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 7 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[8]). The logic required to implement the AVD predictor is also relatively simple as shown in Figure 6. 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.

# 6. 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. [18] in order to tolerate long L2 cache miss latencies. The parameters of the processor we model are shown in Table 2.

**Table 2. Baseline processor configuration.**

| | |
|---|---|
| Front End | 64KB, 4-way I-cache; 8-wide fetch, decode, rename; 64K-entry gshare/PAs hybrid branch pred.; min. 20-cycle mispred. penalty; 4K-entry, 4-way BTB; 64-entry RAS; 64K-entry indirect target cache |
| Execution Core | 128-entry reorder buffer; 128-entry register file; 128-entry ld/st buffer; store misses do not block retirement unless store buffer is full; 8 general purpose functional units; full bypass network; 8-wide retire |
| Caches | 64KB, 4-way, 2-cycle L1 D-cache, 128 L1 MSHRs, 4 load ports; 1MB, 32-way, 10-cycle unified L2, 1 read/write port; 128 L2 MSHRs; all caches have LRU replacement and 64B line size; 1-cycle AGEN |
| Memory | 500-cycle min. latency; 32 banks; 32B-wide, split-trans. core-to-mem. bus at 4:1 frequency ratio; conflicts, bandwidth, and queueing modeled |

[8]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. `MaxAVD`=65535), 2 bits for confidence, and 2 bits to support a True LRU (Least Recently Used) replacement policy.


0-7695-2440-0/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

**Table 3. Relevant information about the studied benchmarks. IPC and L2 miss rates are shown for the baseline runahead processor.**

| | bisort | health | mst | perimeter | treeadd | tsp | voronoi | mcf | parser | twolf | vpr |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |
| 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% |

We evaluate AVD prediction on eleven pointer-intensive and memory-intensive benchmarks from Olden [21] and SPEC INT 2000 benchmark suites. We examine seven memory-intensive benchmarks from the Olden suite, which gain at least 10% 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 [25]. To reduce simulation time, `mcf` is simulated using the MinneSPEC reduced input set [13]. `Parser` is simulated using the test input set. 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.

Table 3 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 3 and the normalized execution times. 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% 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.

## 7. Experimental Results

Figure 7 shows the performance improvement obtained if 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% (5.5% when `health` is excluded) 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 3), 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% (4.0% 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.

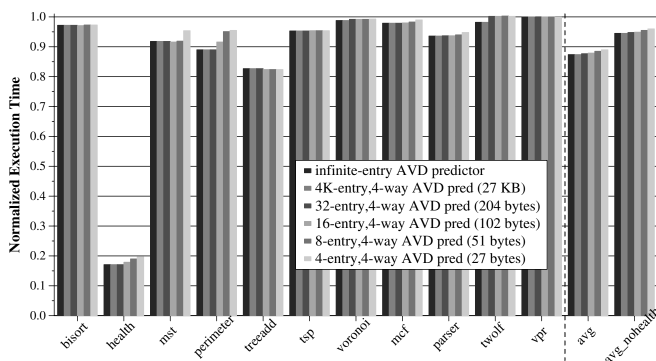



**Figure 7. AVD prediction performance on a runahead processor.**

### 7.1. Effect of MaxAVD

As explained in Section 3, `MaxAVD` is used to dynamically determine which loads are address loads. Choosing a larger `MaxAVD` results in more loads being identified -perhaps incorrectly- as address loads and may increase the contention for entries in the AVD predictor. A smaller `MaxAVD` reduces the number of loads identified as address loads and thus reduces contention for predictor entries, but it may eliminate some address loads with stable AVDs from being considered for AVD prediction. The choice of `MaxAVD` also affects the size of the AVD predictor since the number of bits needed to store the AVD is determined by `MaxAVD`. Figure 8 shows the effect of a number of `MaxAVD` choices on the performance improvement provided by a 16-entry AVD predictor.

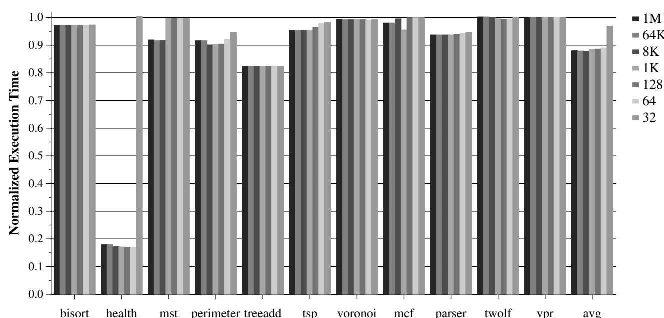



**Figure 8. Effect of `MaxAVD` on execution time (16-entry AVD).**

The best performing `MaxAVD` value is 64K for the 16-entry predictor and 8K for the 4-entry predictor. Unless the AVD is too small (in which case very few address loads are actually identified as address loads), performance is not significantly affected by `MaxAVD`. However, with a 4-entry predictor, a large (1M or 64K) `MaxAVD` provides less performance benefit than smaller `MaxAVD` values in some benchmarks due to the increased contention for predictor en-



0-7695-2440-0/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

tries. We found that most address loads with stable AVDs have AVDs that are within 0-8K range (except for some loads that have stable AVDs within 32K-64K range in `mcf`). This behavior is expected because, as shown in code examples in Section 4, stable AVDs usually occur due to regular memory allocation patterns that happen close together in time. Therefore, addresses that are linked in data structures are close together in memory, resulting in small, stable AVDs in loads that manipulate them.

### 7.2. Effect of Confidence

Figure 9 shows the effect of the confidence threshold needed to make an AVD prediction on performance. A confidence threshold of 2 provides the largest performance improvement for the 16-entry AVD predictor. Not using confidence (i.e., a confidence threshold of 0) in an AVD predictor significantly reduces the performance of the runahead processor because it results in the incorrect prediction of the values of many address loads that do not have stable AVDs. For example, in `bisort` most of the L2-miss address loads are traversal address loads. Since the binary tree traversed by these loads is heavily modified (sorted) during run-time, these traversal address loads do not have stable AVDs. A 16-entry AVD predictor that does not use confidence generates predictions for all these loads but increases the execution time by 180% since almost all the predictions are incorrect. Large confidence values (7 or 15) are also undesirable because they significantly reduce the prediction coverage for address loads with stable AVDs and hence reduce the performance improvement.

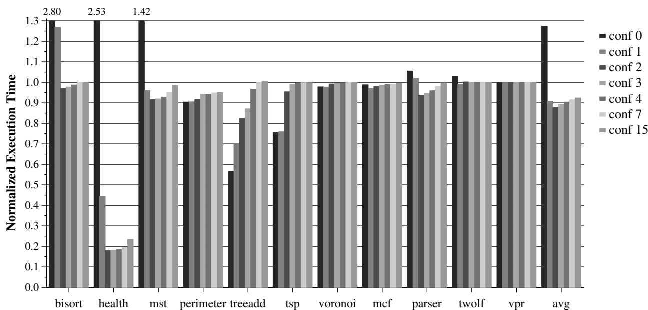



**Figure 9. Effect of confidence threshold on execution time.**

### 7.3. Coverage, Accuracy, and MLP Improvement

Figures 10 and 11 show the effect of the confidence threshold on the coverage and accuracy of the predictor. Coverage is computed as the percentage of L2-miss address loads executed in runahead mode whose values are predicted by the AVD predictor. Accuracy is the percentage of predictions where the predicted value is the same as the actual value. With a confidence threshold of two, about 30% of the L2-miss address loads are predicted and about one half of the predictions are correct, on average. We found that incorrect predictions are not necessarily harmful for performance. Since runahead mode does not have any correctness requirements, incorrect predictions do not result in any recovery overhead. In some cases, even though the predicted AVD is not exactly correct, it is close enough to the correct AVD that it leads to the pre-execution of dependent instructions that generate cache misses that are later needed by correct execution. Thus, a more relevant metric for the goodness of the AVD predictor is the *improvement in the memory-level parallelism* [8, 3]. Table 4 shows the increase in memory-level parallelism achieved with a 16-entry AVD predictor by showing the average number of useful L2 cache misses generated in a runahead period with and without AVD prediction. Note that benchmarks that show large increases in the average number of useful L2 misses with an AVD predictor also show large increases in performance.

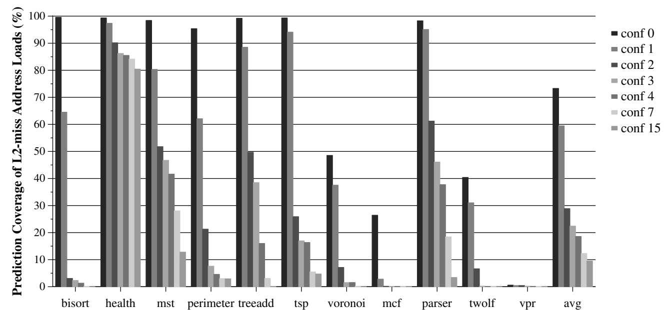



**Figure 10. AVD prediction coverage for a 16-entry predictor.**

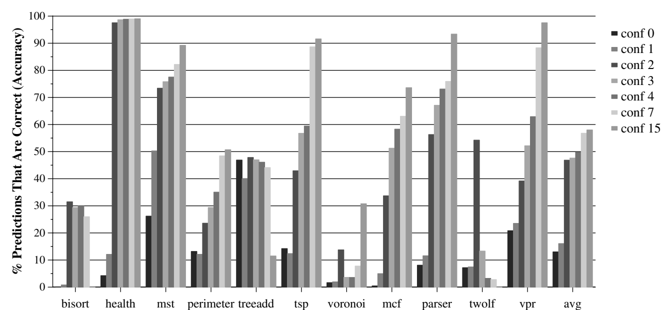



**Figure 11. AVD prediction accuracy for a 16-entry AVD predictor.**

### 7.4. AVD Prediction and Runahead Efficiency

Efficiency is an important concern in designing a runahead execution processor. Runahead execution relies on the pre-execution of instructions to improve performance. This results in a significant increase in the number of processed (executed) instructions as compared to a traditional out-of-order processor. Efficiency of a runahead processor is defined as *the performance increase due to runahead execution* divided by *the increase in executed instructions* [17]. AVD prediction improves efficiency because it both increases performance and decreases the number of executed instructions in a runahead processor. Figure 12 shows that employing AVD prediction reduces the number of instructions processed in a runahead processor by 13.3% with a 16-entry predictor and by 11.8% with a 4-entry predictor. AVD prediction reduces the number of executed instructions because it is able to parallelize and service dependent L2 cache misses during a single runahead period. In a runahead processor without AVD prediction, two dependent L2 misses would cause two separate runahead periods, which are overlapping [17], and hence they would result in the execution of many more instructions than can be executed in a single runahead period.[9]

### 7.5. Effect of Memory Latency

Figure 13 shows the normalized execution time with and without AVD prediction for five processors with different memory latencies.

[9]In fact, an extreme case of inefficiency caused by dependent L2 misses can be seen in health. In this benchmark, using runahead execution increases the number of executed instructions by 27 times, but results in a 4% *increase* in execution time! Using AVD prediction greatly reduces this inefficiency.

Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)
0-7695-2440-0/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

**Table 4. 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 (conf=2) | 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% |

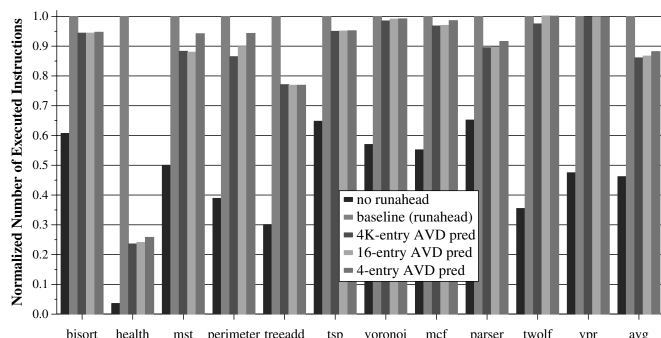



**Figure 12. Effect on number of executed instructions.**

In this figure, execution time is normalized to the baseline runahead processor independently for each memory latency. Execution time improvement provided by a 16-entry AVD predictor ranges from 8.0% for a relatively short 100-cycle memory latency to 13.5% for a 1000-cycle memory latency. AVD prediction consistently improves the effectiveness of runahead execution on processors with different memory latencies, including the one with a short, 100-cycle memory latency where runahead execution is very ineffective and actually increases the execution time by 2%.

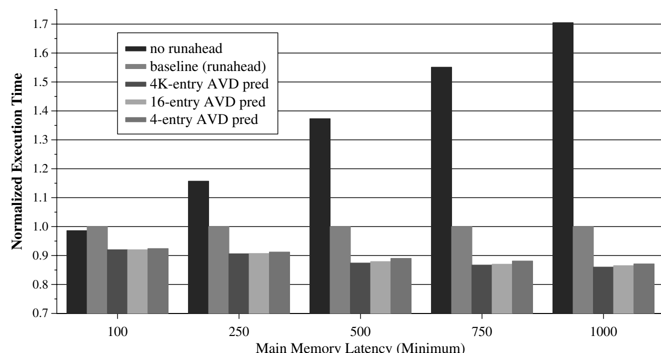



**Figure 13. Effect of memory latency on AVD performance.**

## 7.6. AVD Prediction vs. Stride Value Prediction

We compare the proposed AVD predictor to stride value prediction [24]. 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. Figure 14 shows the normalized execution times obtained with an AVD predictor, a stride value predictor, and a hybrid AVD-stride value predictor.[10] Stride value prediction is more effective when the predictor is larger, but it provides only 4.5% (4.7% w/o `health`) improvement in average execution time even with a 4K-entry predictor versus the 12.6% (5.5% w/o `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% (2.7% w/o `health`), whereas AVD prediction results in 12.1% (5.1% w/o `health`) performance improvement.

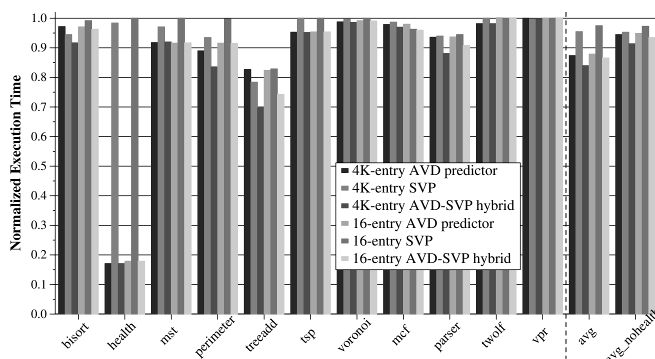



**Figure 14. AVD prediction vs. stride value prediction.**

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 was shown in Figure 5. 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.

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.

Figure 14 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% (6.5% w/o health) improvement in average execution time. As shown in code examples in Section 4, 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

[10] 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.


0-7695-2440-0/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

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

### 7.7. Simple Prefetching with AVD Prediction

So far, we have employed AVD prediction for value prediction purposes, i.e., for predicting the data value of an L2-miss address load and thus enabling the pre-execution of dependent instructions that may generate long-latency cache misses. AVD prediction can also be used for simple prefetching without value prediction. This section evaluates the use of AVD prediction for simple prefetching on the runahead processor and shows that the major performance benefit of AVD prediction comes from the enabling of the pre-execution of dependent instructions.

In the simple prefetching mechanism we evaluate, the value of an L2-miss address load is predicted using AVD prediction during runahead mode. Instead of writing this value into the register file and enabling the execution of dependent instructions, the processor generates a memory request for the predicted value by treating the value as a memory address. A prefetch request for the next and previous sequential cache lines are also generated, since the data structure at the predicted memory address can span multiple cache lines. The destination register of the L2-miss address load is marked as INV in the register file, just like in baseline runahead execution. This mechanism enables the prefetching of only the address loaded by an L2-miss address load that has a stable AVD. However, in contrast to using an AVD predictor for value prediction, it does not enable the prefetches that can be generated further down the dependence chain of an L2-miss load through the execution of dependent instructions.

Figure 15 shows the normalized execution times when AVD prediction is used for simple prefetching and when AVD prediction is used for value prediction as evaluated in previous sections. AVD prediction consistently provides higher performance improvements when used for value prediction than when used for simple prefetching. A 16-entry AVD predictor results in 12.1% performance improvement when it is used for value prediction versus 2.5% performance improvement when it is used for simple prefetching. Hence, the major benefit of AVD prediction comes from the prefetches generated by the execution of the instructions on the dependence chain of L2-miss address loads rather than the prefetching of only the addresses loaded by L2-miss address loads.

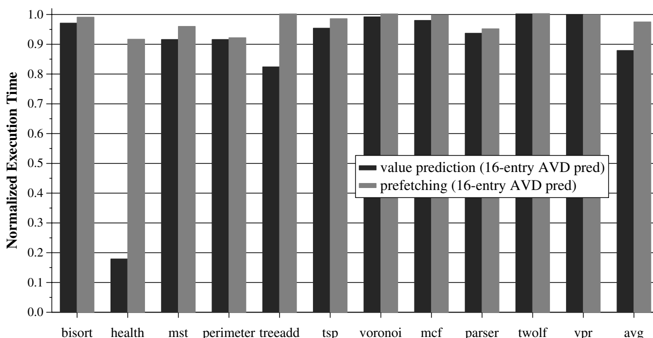



**Figure 15. AVD performance with simple prefetching.**

### 7.8. AVD Prediction on Conventional Processors

We have shown the performance impact of using AVD prediction on runahead execution processors. However, AVD prediction is applicable not only to runahead execution processors. Less aggressive out-of-order execution processors that do not implement runahead execution can also utilize AVD prediction to overcome the serialization of dependent load instructions.

Figure 16 shows the normalized execution times when AVD prediction is used for simple prefetching (as described in Section 7.7) and value prediction on a conventional out-of-order processor.[11] Note that execution time is normalized to the execution time on the conventional out-of-order processor. Using a 16-entry AVD predictor for value prediction improves the average execution time on the conventional out-of-order processor by 4%. Using the same AVD predictor for simple prefetching improves the average execution time by 3.2%. The comparison of these results with the impact of AVD prediction on the runahead execution processor shows that AVD prediction, when used for value prediction, is more effective on the runahead execution processor with the same instruction window size. Since runahead execution enables the processor to execute many more instructions than a conventional out-of-order processor while an L2 miss is in progress, it exposes more dependent load instructions than an out-of-order processor with the same instruction window size. The correct prediction of the values of these load instructions results in higher performance improvements on a runahead processor.

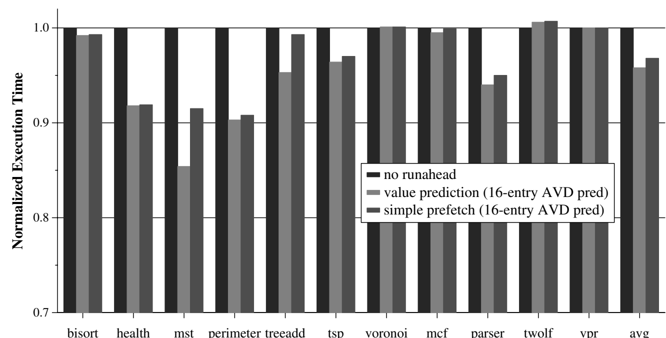



**Figure 16. AVD performance on a non-runahead processor.**

## 8. Related Work

Several previous papers focused on predicting the 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. Other contributions we make in this paper are:

1. We introduce the concept of stable *address-value deltas (AVDs)* and provide an analysis of the code structures that cause them through code examples from application programs.

[11] The parameters for the conventional out-of-order processor are the same as described in Section 6, except the processor does not employ runahead execution. The simple prefetching and value prediction mechanisms evaluated on out-of-order processors are employed for L2-miss loads. We examined using these two mechanisms for all loads or L1-miss loads, but did not see significant performance differences.


IEEE COMPUTER SOCIETY

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.
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.

We hereby briefly discuss the related research in value prediction and prefetching for pointer loads. We also give a brief overview of the related work in runahead execution.

### 8.1. Related Research in Value/Address Prediction

The most relevant work to our research is in the area of predicting the destination register values of load instructions. Load value prediction [15, 24] was proposed to predict the destination register values of loads. Many types of load value predictors were examined, including last value [15], stride [7, 24], FCM (finite context method) [24], and hybrid [27] predictors. While a value predictor recognizes stable/predictable values, an AVD predictor recognizes stable address-value deltas. As shown in code examples in Section 4, 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 7.6.

Load address predictors [7, 1] 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 [20]. Good discussions of the hardware complexity required for complex address/value prediction can be found in [1] and [20].

The pointer cache [4] 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 [4]) 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 [31] 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. Similar to their work, we employ the AVD prediction mechanism only for prefetching purposes, which eliminates the need for processor state recovery.

### 8.2. 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 *not only a prefetching mechanism*. As shown in Section 7.7, 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 pre-execution of dependent load instructions, which may generate prefetches.

Hardware-based pointer prefetchers [22, 23, 4, 10] 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) [23]. A low-overhead content-based hardware pointer prefetcher was recently proposed by Cooksey et al. [5]. 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 [14, 16, 23, 30, 11]. These techniques require non-trivial support from the compiler or the programmer. Existing binaries cannot utilize software-based techniques unless they are re-compiled or re-optimized using a dynamic optimization framework. AVD prediction, on the contrary, is a purely hardware-based mechanism that can improve the performance of the existing binaries.

### 8.3. Related Research in Runahead Execution

Three recent papers proposed combining runahead execution with value prediction [3, 12, 2]. These techniques use conventional value predictors to predict the values of *all* L2-miss load instructions during pre-execution, which requires significant hardware support (at least 2K-entry value tables). In contrast, we propose a novel 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 [12], 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 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.

## 9. Conclusion and Future Work

This paper introduces the concept of *stable address-value deltas (AVDs)* and proposes *AVD prediction*, a novel method of predicting the values generated by address loads 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.


0-7695-2440-0/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

The proposed AVD prediction mechanism requires neither significant hardware cost or complexity nor hardware support for state recovery. Our experimental results show that a simple AVD predictor can significantly improve both the performance and efficiency of a runahead execution processor across a wide range of main memory latencies. For a 500-cycle minimum main memory latency, the execution time improvement provided by a 16-entry (102-byte) AVD predictor is 12.1% over a set of pointer-intensive applications from Olden and SPEC INT 2000 benchmark suites.

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. Hence, software (programmer/compiler/allocator) support can improve the effectiveness of a mechanism that exploits address-value deltas. We intend to examine this possibility in our future work.

## Acknowledgments

Many thanks to Derek Chiou, David Armstrong, Mike Fertig, Santhosh Srinath, Chang Joo Lee, Linda Bigelow and other members of the HPS research group, and the anonymous reviewers for their comments and suggestions. We gratefully acknowledge the Cockrell Foundation and Intel Corporation for their support.

## References

[1] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser. Correlated load-address predictors. In *ISCA-26*, 1999.

[2] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas. CAVA: Hiding L2 misses with checkpoint-assisted value prediction. *Computer Architecture Letters*, 3, Dec. 2004.

[3] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA-31*, 2004.

[4] J. D. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *MICRO-35*, 2002.

[5] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS-X*, 2002.

[6] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS-1997*, 1997.

[7] R. J. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37:547–564, 1993.

[8] A. Glew. MLP yes! ILP no! In *ASPLOS Wild and Crazy Idea Session '98*, Oct. 1998.

[9] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *DAC-35*, 1998.

[10] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *ISCA-24*, 1997.

[11] M. Karlsson, F. Dahlgren, and P. Strenstrom. A prefetching technique for irregular accesses to linked data structures. In *HPCA-6*, 2000.

[12] N. Kırman, M. Kırman, M. Chaudhuri, and J. F. Martínez. Checkpointed early load retirement. In *HPCA-11*, 2005.

[13] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[14] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *MICRO-28*, 1995.

[15] M. H. Lipasti, C. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *ASPLOS-VII*, 1996.

[16] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *ASPLOS-VII*, 1996.

[17] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *ISCA-32*, 2005.

[18] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA-9*, 2003.

[19] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro*, 23(6):20–25, 2003.

[20] P. Racunas. *Reducing Load Latency Through Memory Instruction Characterization*. PhD thesis, University of Michigan, 2003.

[21] A. Rogers, M. C. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.

[22] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *ASPLOS-VIII*, 1998.

[23] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *ISCA-26*, 1999.

[24] Y. Sazeides and J. E. Smith. The predictability of data values. In *MICRO-30*, 1997.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X*, 2002.

[26] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *ISCA-29*, 2002.

[27] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *MICRO-30*, 1997.

[28] M. V. Wilkes. The memory gap and the future of high performance memories. *ACM Computer Architecture News*, 29(1):2–7, Mar. 2001.

[29] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *ACM Computer Architecture News*, 23(1):20–24, Mar. 1995.

[30] C.-L. Yang and A. R. Lebeck. Push vs. pull: Data movement for linked data structures. In *ICS-2000*, 2000.

[31] H. Zhou and T. M. Conte. Enhancing memory level parallelism via recovery-free value prediction. In *ICS-17*, 2003.



IEEE COMPUTER SOCIETY