B-Fetch:Branch Prediction Directed Prefetching for In-Order Processors

Reena Panda*, Paul V. Gratz*, and Daniel A. Jiménez†

*Computer Engineering and Systems Group (CESG),
Department of Electrical and Computer Engineering,Texas A&M University
†Department of Computer Science, The University of Texas at San Antonio
*{reena.panda,pgratz}@tamu.edu †dj@cs.utsa.edu

Abstract—Computer architecture is beset by two opposing trends. Technology scaling and deep pipelining have led to high memory access latencies; meanwhile, power and energy considerations have revived interest in traditional in-order processors. In-order processors, unlike their superscalar counterparts, do not allow execution to continue around data cache misses. In-order processors, therefore, suffer a greater performance penalty in the light of the current high memory access latencies. Memory prefetching is an established technique to reduce the incidence of cache misses and improve performance. In this paper, we introduce B-Fetch, a new technique for data prefetching which combines branch prediction based lookahead deep path speculation with effective address speculation, to efficiently improve performance in in-order processors. Our results show that B-Fetch improves performance 38.8% on SPEC CPU2006 benchmarks, beating a current, state-of-the-art prefetcher design at $\sim 1/3$ the hardware overhead.

Index Terms—Data Cache Prefetching, Memory Systems, Branch Prediction, Value Prediction, In-order Processors

1 Introduction

Modern computer architecture is beset by two opposing, conflicting trends. First, while technology scaling and deep pipelining have led to high processor frequencies, the memory access speed has not scaled accordingly. This discrepancy has led to high memory access latencies which impose a serious constraint on system performance. Meanwhile, power and energy considerations have revived interest in in-order processors. Many modern processors, such as Intel's Atom [6], Tilera's Tile64 [17], and Suns UltraSPARC T1 "Niagara" [8], incorporate a greater number of small in-order cores, versus fewer, large superscalar cores, saving power while improving throughput. In-order processors, unlike their superscalar counterparts, do not allow execution around data cache misses. Put together with trends in memory access latency, in-order processors suffer a greater performance penalty in the light of the current high memory access latencies. In this paper we present a novel data cache prefetch scheme, leveraging both execution path speculation as well as effective address speculation, to efficiently improve performance of in-order processors.

Much prior work focuses on reducing the impact of the memory-wall on processor performance. One such technique is data prefetching, where future processor memory requests are speculatively pre-loaded into the cache to avoid the high miss latencies. Several data prefetching techniques have been proposed [2, 5, 9, 13, 15]. Somogyi et al. proposed the current top-performing, practical prefetcher, the Spatial Memory Streaming (SMS) prefetcher [15] (we consider this "practical" because its storage overhead is a relatively modest 33KB). SMS predicts the future access pattern within a spatial region around a miss, based on a history of access patterns initiated by that missing instruction in the past. While the SMS prefetcher is effective, it indirectly infers future program control-flow to speculate on the misses in a spatial region, as a result its overhead in terms of required state can be high.

Other designs more directly speculate on control flow to determine which data to preload in the caches. Prior branch-prediction directed prefetching schemes either focused on instructions [16], with limited benefit due to high L1I cache hit rates; or as simple augmentations to stride based prefetchers [9, 13], with significantly lower performance than current best of class prefetchers. Although these techniques accurately speculate on which loads will occur, their performance is poor because they do not accurately speculate on the effective address of those loads. In Pinter and Yoaz's prefetcher [13] effective addresses from the last execution of a load are combined with offsets to produce the new expected value, using a technique similar to register value speculation [12]. The key insight and novelty of our technique is, for prefetching, effective address values can be predicted more accurately based upon their variance from the *current architectural state* at an earlier basic block, versus an offset off the *previously generated effective address*, determined by the last execution of that instruction.

Roth et al. proposed a prefetching technique for pointer based data structures which extracts a simplified kernel of the data's pointer reference structure and executes it without the intervening instructions [14]. While this is effective for these types of data structures, it provides no benefit for other types of code. Farooq et al. propose a technique for indirect branch target speculation which also leverages register state information, however their technique requires compiler inserted instruction hints in program code [3]. Our technique bears a passing similarity to Runahead execution [11]. Runahead execution effectively functions as a prefetcher by speculating past stalls incurred by long-latency memory instructions. Our technique, however, should have significantly lower power overhead, as our prefetch pipeline is much smaller and lower complexity than the main pipeline. Furthermore, the prefetch pipeline acts independently and simultaneously with the right-path instructions and need not wait for miss-induced stalls to become active. We are aware of no existing prefetcher design which uses a branch predictor to speculate on control-flow, combined with effective address speculation based upon current architectural state in a light-weight prefetcher design.

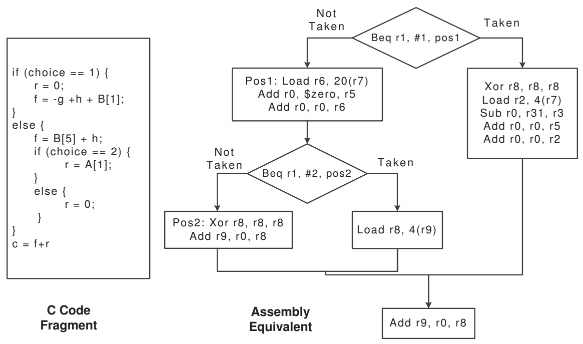
In this paper we propose B-Fetch, a combined control-flow and effective address speculating prefetching scheme. B-Fetch leverages the high prediction accuracies of current-generation branch predictors, combined with novel effective address speculation. We demonstrate that the B-Fetch prefetcher outperforms the best-in-class prefetcher, SMS, at $\sim 1/3$ the hardware overhead. While the focus of this paper is improving in-order processor performance, the B-Fetch technique should perform comparably on superscalar processors.

2 Background

Current memory access latencies are in the order of hundreds of processors cycles. To be effective at masking such high latencies, a prefetcher must anticipate misses and issue prefetches significantly ahead of actual execution. This requires accurate prediction of a) the likely memory instructions to be executed, and b) the likely effective addresses of these instructions.

Program execution path (i.e. which basic blocks are executed

Manuscript submitted: 16-Sep-2011. Manuscript accepted: 21-Nov-2011. Final manuscript received: 21-Nov-2011.



```
if (choice == 1) {
    r = 0;
    f = -g +h + B[1];
}
else {
    f = B[5] + h;
    if (choice == 2) {
        r = A[1];
    }
    else {
        r = 0;
    }
}
c = f+r
```

```
Beq r1, #1, pos1

Pos1: Load r6, 20(r7)
Add r0, $zero, r5
Add r0, r0, r6

Xor r8, r8, r8
Load r2, 4(r7)
Sub r0, r31, r3
Add r0, r0, r5
Add r0, r0, r2

Beq r1, #2, pos2

Pos2: Xor r8, r8, r8
Add r9, r0, r8

Load r8, 4(r9)

Add r9, r0, r8
```

Fig. 1: A C-code fragment and its control flow graph equivalent

and in what sequence) is determined by the direction taken by the component control instructions. The memory access behavior can therefore be linked to the prior control flow behavior. For example, consider a **C** code fragment comprised of an if-else code block in Figure 1. The block of code (if-block or else-block) executed following the control instruction depends on the direction taken by it. Data that gets requested in the future execution phases and its access patterns are dependent on the branch instructions encountered along the path and their corresponding outcomes. We therefore propose to employ a lookahead mechanism that predicts a likely path of execution starting from a current branch and issues prefetches for the memory references down that path.

To accurately predict effective addresses down the predicted path, we leverage the observation that memory references use a particular register for effective address computation. Unlike previously proposed prefetching approaches, which use history based effective address computation techniques, we propose to associate register indices being operated by the memory instructions with their preceding control instructions (the entry points of the basic block) and use this correlation to identify prefetch candidate addresses. This idea is based on the premise that register values at the time of effective address generation are correlated in a predictable way from their corresponding values at a time when their preceding branch instructions were executed. To test this hypothesis, we conducted an experiment using application traces collected from a subset of SPEC CPU2006 benchmarks. We find that register values correlate with future effective addresses across multiple branches with a greater than 50% correlation to a small offset up to four branches in the future (generally higher than the accuracy of prior value prediction techniques [12]). By taking into account the predictable variations off of the register values, we find effective address speculation can be much more accurate. As we will show, B-Fetch not only predicts effective addresses which display regular-access patterns, but also can take advantage of the dynamic values of the registers at run-time to predict irregular and isolated data accesses.

3 Proposed Design

B-Fetch is a data cache prefetching mechanism which employs two speculative components, speculation on a) the memory instructions to be executed, and b) the effective addresses of these instructions. For the first, we implement a lookahead mechanism using branch prediction to predict the future execution path. For the second, we exploit the correlation between effective address values in a basic block and their dependent register values at earlier branch instructions. This correlation is determined in hardware by associating the effective addresses of loads in a given basic block to the value of the source register at preceding branch instructions. Making use of the *current register values* rather than the *effective address history*, B-Fetch can issue useful prefetches even for instructions that exhibit irregular access patterns.

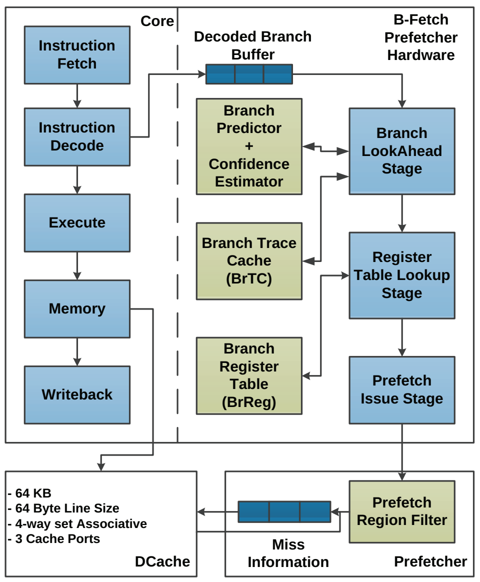


Fig. 2: Overall System Architecture

3.1 Overview

This section discusses the operational modes of the B-Fetch prefetch engine and gives an overview of the B-Fetch pipeline.

Modes of Operation: B-Fetch supports two modes of operation, *Non-Loop* and *Loop* mode. *Non-Loop* mode is used when each branch leads to a new basic block, while *Loop* mode is used when executing loops which cause repeated branches back to the same basic block. When a prefetch address is generated in the *Non-Loop* mode, all cache blocks within a contiguous region of eight cache lines containing the predicted address are issued to the prefetch queue to capture greater variability in the address values from the past architectural register values. In *Loop* mode, while at a given dynamic iteration of a basic block, prefetches are issued for future iterations. Only the line containing the predicted effective address is prefetched. In either mode, in the event of a non-prefetched cache miss, the next sequential line after the missing line is prefetched. Switching between the two modes is handled completely in hardware, if a future basic block is found to be the same as the current basic block, *Loop* mode is enabled. There is no overhead in latency to perform this switch. Some additional state is dedicated to *Loop* mode to accurately characterize how register values change from one loop iteration to the next.

Pipeline Overview: Figure 2 depicts the detailed architecture of a B-Fetch enabled in-order processor, showing the main execution pipeline and the additional hardware for the B-Fetch prefetcher. We note that in this baseline design, the host processor is only a simple five-stage pipeline while many modern processors have much deeper pipelines, this forms a "worst case" for prefetching as the impact of misses on deeper pipelines is higher than on shallow. The B-Fetch hardware forms a separate, 3-stage prefetch pipeline, parallel to the main execution pipeline. The two pipelines are connected via a 3-entry, Decoded Branch Buffer (DBB). As branch instructions are decoded in the main execution pipeline, corresponding PCs are fed into the DBB, which are then fetched into the prefetch pipeline to initiate the prefetching process. The B-Fetch pipeline consists of the following stages:

1) **Branch Lookahead:** where the predicted execution trace is created. This stage contains the branch predictor and the Branch Trace Cache (BrTC) structure to connect the predicted branches with the next branch down that path. This stage also contains a path confidence estimator which estimates path confidence to stop prefetching when

the path confidence is low.

2) **Register-Table Lookup:** where the relationship between the memory instructions in a given basic block and their correlated register values are exposed and exploited to generate candidate prefetch effective addresses. This stage makes use of the Branch-Register (BrReg) Table.
3) **Prefetch Issue:** where prefetch addresses are issued to the prefetch queue, after suitable filtering. The prefetch queue holds prefetch requests until there is available cache bandwidth and no demand requests are pending.

3.2 System Components

The structures that form each of B-Fetch pipeline stages are detailed in this section.

Branch Trace Cache (BrTC): The BrTC enables lookahead across multiple basic blocks. Figure 3 shows a typical entry of the BrTC. The BrTC caches pairs of branch instructions, such that the relationship between the branch at the end of a basic block and the branch plus direction that led to that basic block is established. The intent is to enable jumps from one basic block to another, skipping all the non-control-flow changing instructions in between. It is implemented as a table that is indexed with the current branch PC together with its direction of execution. This table is dynamically filled in during runtime and only commit-time updates of the BrTC entries are allowed to avoid wrong-path updates.

Branch PC & Direction (33-bits)	Next Branch PC (32-bits)	NextBranchIsUnconditional (1-bit)

Fig. 3: Single Branch Trace Cache Entry

Path Confidence Estimator: The path confidence estimator controls the degree of lookahead across basic blocks by keeping track of the cumulative confidence of the predicted execution path. Whenever the cumulative path confidence falls below a threshold value, indicating a likely wrong path prediction, the lookahead process is terminated. B-Fetch calculates the path confidence using an approach similar to that proposed by Malik et al. [10], with the modifications of employing a composite confidence estimator by combining the JRS, up-down and self-counters proposed by Jiménez [7], to estimate individual branch confidence values.

Branch-Register Table (BrReg): The BrReg generates prefetch candidate effective addresses by establishing and exploiting the correlation between the source register indices that are used for effective address computation in memory instructions in a basic block and their preceding branch instructions. An entry is allocated for a memory reference instruction, indexed by its preceding branch PC and the relevant information is kept and updated on subsequent executions.

Branch Tag (30-bits)	RegIndex (5-bits)	RegValue (32-bits)	Offset (16-bits)	Delta (16-bits)	Delta-Valid (1-bit)	Delta-is-Changing (1-bit)	PF Bit (1-bit)	Loop Counter (4-bits)	Seq Num (4-bits)

Fig. 4: Single Branch Register Table Entry

Figure 4 shows a typical BrReg entry. The table is indexed with the PC of the entry branch prior to a given basic block (also contained in the *Branch Tag* field). Additionally, the BrReg table contains the following fields: The *RegIndex* is a multi-entry field that holds the source register indices used for memory address generation in the corresponding basic block, linking source registers to the branch that led to the basic block. This linkage is learned as control instructions and memory instructions commit in program order in the main execution pipeline.

We observe that even if register values and effective address do not exactly match, they still lie within a semi-fixed offset from each other. The *Offset* field retains these fixed-offset relationships. This offset value is learned by monitoring the addresses generated by memory instructions in the main pipeline, compared against the stored *RegValue* field.

Prefetcher	System Component	Number of Entries	Size
B-Fetch	Branch Trace Cache	256	2 KB
	Branch Register Table	128	6.125 KB
	Alternate Register File	64	256 Bytes
	Prefetch Filter	1K	384 Bytes
	Prefetch Queue	100	75 Bytes
	Path Confidence Estimator	2K	2 KB
	Misc.	-	300 Bytes
	TOTAL SIZE		11.1 KB
SMS	Active Generation Table	64	2.937 KB
	Filter Table	32	1.46 KB
	Pattern History Table	2K	28 KB
	TOTAL SIZE		32.4 KB

TABLE 1: Hardware Overhead

Four additional fields are added to increase the prefetching coverage for loop-based program sequences, allowing dynamic identification of loop based code. The *Delta* field holds the difference between the generated effective address over consecutive execution instances of the same instruction. The *Delta-Valid* field is used to signify a valid delta value while the *Delta-is-Changing* field allows hardware identification of loops and aids in retaining an appropriate delta value. The *Loop Counter* field monitors the iteration count of the loop in the lookahead mode. The *Seq Number* is used for bookkeeping, to ensure the proper iteration of the loop updates the tables. Equation 1 shows the prefetch effective address generation formula for both modes of execution.

$$PrefetchAddress = [RegisterValue] + Offset + (LoopCounter \times Delta \times Delta\text{-}Valid) \qquad (1)$$

Prefetch Filtering: B-Fetch adopts several prefetch filtering mechanisms to reduce the useless prefetches generated. First, a *Region-Filtering FIFO Filter* is employed, preventing prefetches for the same cache blocks in close succession. Second, the *Region-Access Density Filter* predicts the usefulness of prefetching a region around a speculative effective address by examining the history of useful prefetches made to that region. It is useful for filtering out the prefetches generated in the non-loop mode. Finally, a *Prefetch Queue Cleanup Filter* scans the prefetch queue, removing prefetched entries for those basic blocks that have already been or are being executed by main pipeline.

3.3 Hardware Cost

The additional hardware requirements of the B-Fetch scheme, versus SMS is summarized in Table 1. The table shows B-Fetch requires $\sim$33% of the table state required by SMS. B-Fetch establishes a branch-based correlation to guide prefetching. This approach is more efficient, in part, because programs typically have more memory instructions than control instructions; therefore, a branch-based prefetcher captures the same correlations at much reduced table sizes.

4 EVALUATION

4.1 Methodology

We evaluate our prefetcher in a simulation environment based on the M5 Simulator [1]. The simulator is used to model a 1-wide, 5-stage in-order pipeline. We note this reference configuration is quite conservative, the benefit of prefetching on an aggressively pipelined configuration are likely much greater due to the higher impact of misses in deeper pipelines. The assumed memory model consists of a 2-level cache hierarchy with 64KB 4-way set-associative L1I and L1D caches and a 2MB 16-way set-associative L2 Cache. L2 cache hits are serviced in 16 cycles and memory accesses are serviced in 60. Although experiments were performed in a single core environment, we expect that the results should hold for multi-programmed workloads in a multi-core environment as prefetching is exclusive to the L1D cache. We plan to explore multiprogrammed and multithreaded workloads in future work.

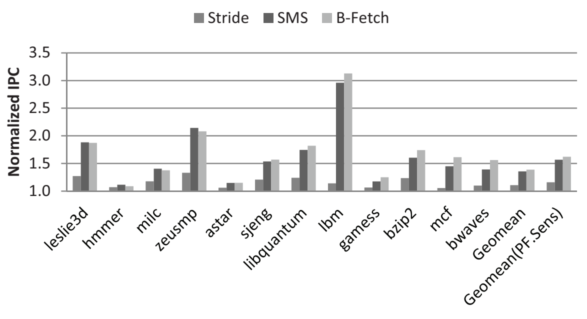


Fig. 5: Normalized IPC of *Stride*, *SMS* and *B-Fetch*.

The test workload consists of the 18 SPEC CPU2006 benchmarks that our simulation infrastructure supports, compiled for the ALPHA ISA. The reference input set is used for each benchmark. All benchmarks are run for the first 1.5 billion instructions. Our initial evaluation showed six of the 18 benchmarks (namd, calculix, GemsFDTD, tonto, h264ref, and soplex) did not contain significant L1D cache misses and as such showed no benefit from either prefetching method, these benchmarks are omitted from the results figure (although they are included in calculating the Geomean). The remaining 12 benchmarks are considered to be "Prefetch Sensitive".

We compare the performance of B-Fetch against the state-of-the-art SMS prefetcher as implemented by Ferdman et al. [4]. Matching the design presented in their paper, we consider SMS with 512 Byte spatial region size, a 64-entry accumulation table and a 2K-entry pattern history table. We also include a 32-entry filter table, as per the original SMS proposal. We also compare B-Fetch's performance against a stride prefetcher.

4.2 Prefetcher Performance

Figure 5 contains the IPC for the simple Stride, SMS and B-Fetch prefetchers normalized against baseline. *Geomean* refers to the results across the entire set of benchmarks and *Geomean (PF. Sens)* refers to performance across the prefetch sensitive benchmarks. The results show the B-Fetch prefetcher provides a mean speedup of 39% (62%) across all (prefetch sensitive) benchmarks. As compared to a stride prefetcher, B-Fetch improves the performance by over 25% (39.4%). Compared against SMS, B-Fetch improves the performance by 2.2% (3.4%), at the cost of $\sim 1/3$ the overhead in storage.

Analysis: B-Fetch provides performance benefits across a range of applications, both integer and floating point. This implies that exploiting the branch-register correlation, as proposed can reliably benefit general applications. Comparing B-Fetch with a 64KB L1 against a 128KB L1, we find B-Fetch still maintains a strong 36% (58%) improvement in IPC across all (prefetch sensitive) benchmarks. Hence, we note B-Fetch's 11.1KB is an effective use of a minimal overhead. Since the B-Fetch prefetcher aggressively issues prefetches based upon future basic blocks, it issues more prefetches than competing designs which must wait for misses to start prefetching. As a result, however, B-Fetch is stalled ~40% of the time because the core is using the L1, and is somewhat less accurate than SMS with 53% (59%) of prefetches masking miss latency across all (prefetch sensitive) benchmarks as compared with 65% for SMS. In future work we will examine throttling when L1 hit rates are high to reduce the useless prefetches produced. We note that while these results only explore the benefit B-Fetch provides in a uniprocessor system, we feel they should generally hold for CMPs as well, though some extra interconnect traffic can be expected due to increased L2 accesses. If interconnect bandwidth began to limit performance, prioritizing demand fetches over prefetches should alleviate it.

5 Conclusions

In this paper, we proposed a data prefetcher that takes advantage of the high prediction accuracies of current-generation branch predictors to accurately generate the future basic block trace and initiates data prefetching for memory instructions in those future basic blocks. We demonstrate that there is a strong correlation between the effective addresses generated by memory instructions and the values of the corresponding source registers at prior branch locations. Making use of the live architectural register values, with the help of the offset-based and loop-based enhancements, our prefetcher is capable of generating accurate and timely prefetches for data exhibiting both regular and irregular access patterns. The current implementation of the B-Fetch provides a mean benefit of 39% over baseline and outperforms the state-of-the-art prefetcher, while incurring a minimal additional hardware cost of 11.1 KB. We note that, while the focus of this paper has been improving in-order processor performance, the B-Fetch scheme should perform comparably on superscalar processors, we plan to explore this in future work. We also plan to explore how B-Fetch performs on other workloads including commercial workloads which tend to be more irregular and hence may benefit our design. Finally, we will explore how the B-Fetch prefetcher might be virtualized to further reduce overheads.

References

[1] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, pp. 52–60, 2006.

[2] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *The International Conference on Supercomputing (ICS)*, 1997, pp. 68–75.

[3] M. Farooq, L. Chen, and L. John, "Value based btb indexing for indirect jump prediction," in *16th International Symposium on High Performance Computer Architecture (HPCA)*, jan. 2010, pp. 1 –11.

[4] M. Ferdman, S. Somogyi, and B. Falsafi, "Spatial memory streaming with rotated patterns," in *The 1st JILP Data Prefetching Championship*, 2009.

[5] T. fu Chen and J. loup Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, pp. 609–623, 1995.

[6] T. Halfhill, "Intel's tiny atom," *Microprocessor Report*, vol. 22, no. 4, p. 1, 2008.

[7] D. A. Jimenez, "Composite confidence estimators for enhanced speculation control," in *The 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2009, pp. 161–168.

[8] K. Krewell, "Suns niagara pours on the cores," vol. 18, no. 9, 2004, pp. 11–13.

[9] Y. Liu and D. R. Kaeli, "Branch-directed and stride-based data cache prefetching," in *The International Conference on Computer Design*, ser. ICCD, 1996, pp. 225–230.

[10] K. Malik, M. Agarwal, V. Dhar, and M. Frank, "Paco: Probability-based path confidence prediction," in *The 14th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2008, pp. 50–61.

[11] 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 *The 9th International Symposium on High-Performance Computer Architecture (HPCA)*, 2003.

[12] T. Nakra, R. Gupta, and M. Soffa, "Global Context-Based Value Prediction," in *Fifth International Symposium On High-Performance Computer Architecture (HPCA)*. IEEE, 1999, pp. 4–12.

[13] S. Pinter and A. Yoaz, "Tango: a hardware-based data prefetching technique for superscalar processors," in *The 29th annual ACM/IEEE international symposium on Microarchitecture (Micro)*. IEEE Computer Society, 1996, pp. 214–225.

[14] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *The Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998, pp. 115–126.

[15] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *The 33rd annual international symposium on Computer Architecture (ISCA)*, 2006, pp. 252–263.

[16] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak, "Branch history guided instruction prefetching," in *The 7th International Conference on High Performance Computer Architecture (HPCA)*, 2001, pp. 291–300.

[17] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.