

The Design of Cost-Effective Stride-Prefetching for Modern Processors

Hassan Al-Sukhni* James Holt* Daniel A. Connors** Mike Snyder* Matt Smittle* Brian Grayson*

*Freescale Semiconductor, Inc.
Austin, TX

**Dept. of Electrical and Computer Engineering
University of Colorado at Boulder

Abstract—Data prefetching of regular access patterns is an effective mechanism to hide the memory latency for modern microprocessors. However, to be included in an architecture design, prefetching systems must be cost-effective and have little impact to the microarchitecture. For example, while many proposed prefetching systems use the full program counter (PC) to help detect patterns with arbitrary strides, such systems are impractical and prohibitive. To overcome the issues related to using the entire PC for effective prefetching, this paper combines other instruction attributes with a small subset of the PC to help detect the regularity in program data accesses. Such detection is enabled by a finite state machine that resolves data stream allocation, maintains prefetch priorities, and manages prefetch run-ahead. The experimental results suggest that as little as 4 bits of the PC are sufficient to achieve within 1% of the same prefetching effectiveness as using the full PC.

I. INTRODUCTION

Prefetching is an appealing technique to overcome the memory latency problem in modern processors [1]. Several approaches have been proposed using both hardware [2], [3], [4], [5], [6], [7] and software [2], [8] to enable prefetching. There are a number of tradeoffs in characterizing the best prefetching mechanism, but hardware techniques are generally shown to have the best potential to adapt to run-time program characteristics. For example, stream buffers prefetching is a particularly interesting hardware technique because of its ability to detect and prefetch memory accesses exhibiting unit-strides [9]. Unit-strides appear in regular accesses when data is located in sequential memory cache lines. However, stream buffers do not detect strides larger than a cache line, referred to as *arbitrary-strides*.

An effective way of detecting arbitrary-strides is to correlate the addresses of memory accesses with the program counters (PCs) of the load instructions generating the accesses [10]. Design trends in modern processors increase performance by increasing the processor clock frequency, in turn, requiring deeper pipelines [11] (more stages). In such pipelines, data access operations are handled by specialized hardware, referred to as the load-store unit (LSU). The LSU is typically located at the back-end of the processor pipeline, where the PC is not available. In practice, transmitting the full PC across the pipeline stages (as required by most proposed prefetching schemes) is expensive. This expense will increase as modern processor architectures transition from 32-bit to 64-bit PCs, requiring double the size in both pipeline registers and data

TABLE I
ESTIMATED STRUCTURAL REQUIREMENTS TO USE FULL PROGRAM
COUNTER FOR PREFETCHING.

Stage	Latch bits	Logic bits
Issue	176	88
Slot	44	44
Reservation	44	44
LSU(AQ)	308	44
Exe pipe	308	44
PEs (8)	352	704
Total bits	1232	968

paths.

Table I shows estimated structural requirements for extending a modern high-performance microarchitecture design to transmit the full PC throughout the pipeline. In this example, the PC needs to be transmitted through pipeline stages: issue, issue slots, reservation stations, execution pipeline stages, LSU, and finally to the prefetching hardware components referred to in the table as Prefetch Engines (PEs). The data shown assumes that the implemented microarchitecture supports a 44-bit PC after necessary virtual to physical address translations. The table shows the required additional latching and logic bits for each stage. The overall required number of bits, to transmit the PC to the PE processing it, is 1,232 latch bits and 968 logic bits. Note that these bits imply added requirements in terms of timing, floor planning, power consumption, verification, etc. Therefore, this paper presents an alternative approach to using the full PC in the detection of arbitrary strides in the memory accesses. The approach uses a subset of the PC combined with other attributes of the instruction generating the access stream to identify such a stream within the memory accesses.

The main trade-off for consideration in the design of effective prefetching for a microprocessor is performance, which is governed by prefetch accuracy, coverage, and timeliness. In the case of stream buffer prefetching [9], accuracy is the biggest limitation. Poor accuracy results not only in missed opportunities, but in bad prefetches that actually hurt the overall system performance in two distinct ways. The first is that prefetched data can replace useful data already in the various levels of the cache system. This problem is often referred to as cache pollution. Cache pollution can be avoided by prefetching into dedicated buffers [9], [12] or to lower level

caches, e.g. the L2 data cache [13]. However, such a solution results in additional delays that limit the benefits of accurate prefetches. The second way that bad prefetches degrade performance is by incorrectly using resources like queues, buffers and bandwidth that otherwise could be used by program instructions. Arbitration techniques that give prefetches lower priority to access the system resources can help reduce this effect [14], [15], but they cannot eliminate it. Therefore, it is important to devise mechanisms to minimize the number of bad prefetches. As such, the second contribution of this paper is a finite state machine implementable within a modern microprocessor that manages stride detection and dynamic control of prefetching to improve accuracy.

The remainder of the paper is organized as follows: Section II presents the proposed prefetching approach, Load Attribute Prefetching (LAP). Section III presents our evaluation methodology. Experimental results are presented in Section IV, and finally, conclusions are drawn in Section V.

II. APPROACH

Because the PC of missing load instructions is used to uniquely identify streams, a prefetch system that uses only a small subset of the PC to enable arbitrary stride detection may result in making several missing load instructions appear to be the same instruction. In order to minimize such aliasing of load instructions, other attributes of the instruction encoding can be used to identify streams with strided patterns. Such attributes are usually needed for other reasons within the pipeline and therefore do not require major structural additions or impact the amount of information that must be propagated between stages.

The LAP approach capitalizes on these observations. LAP is demonstrated using the PowerPC®¹ instruction set, and can be extended to other architectures. Load instructions in the PowerPC® instruction set architecture, are of the form: $ld\ rD, \delta(rA)$ where rD is the destination register, rA is the base register of the load instruction and δ is a displacement that is either specified as an immediate offset within the instruction or in another register rB . Both rA and rB are five-bit fields in the load instruction. These fields (including the offset) are usually transmitted across the pipeline (at least up to the register rename stage) in modern architectures.

The LAP prefetching system extends the traditional idea of stride prefetching by introducing a set of hardware components that accurately and dynamically detect exploitable address patterns. In contrast to traditional schemes, potential prefetch addresses are generated by coordinating many pieces of strategically tagged in-flight information through a dedicated state machine. The proposed architecture mechanism consists of the four components: Stream Identification, Stride Detection, Allocation Filters, Run-ahead Prefetching and Feedback.

Stream Identification: In order to effectively associate address requests and prefetches with a particular instruction, a stream identifier (*Stream_ID*) is designated. The *Stream_ID*

is used as a reference index into different prefetch structures which generally should not have a large index for faster access. The proposed LAP approach uses, in addition to the partial PC, the least significant 4 bits of the instruction's base index register rA . The base index register is selected with the intuition that critical loads with high-percentage miss rates access addresses not common to other loads. As such, the base register value as well as its index should generally be unique among other loads in a dynamic code sequence. The *Stream_ID* is composed by concatenating the 4 lower bits of the source register identifier rA with the selected portion of the PC.

Each of the identified streams is allocated to one of a number of prefetching hardware components called *prefetch engines* (PEs). Each PE is controlled by a state machine that implements the other three components of the LAP mechanism (Figure 1). The PE can be in either an active or inactive state. In the inactive states, the PE detects the program strided patterns. While in the active states, the PE makes predictions and issues prefetches. These states are ordered, reflecting how confident the PE is in its ability to predict addresses that are usable by the running program. The states, ordered from least to highest confidence, are: *NQD*, *SP*, *SPD*, *LC1*, *HC1*, *HC2*, *HC4*, and *HC6*. The states below *LC1* are the inactive states, while the rest are active. The number appearing in the active states indicates the number of prefetches that the PE can have outstanding in the system. Table II presents the main function of each of these states.

Stride Detection When an address is generated by a load operation, the corresponding stream identifier for the address is used to find a matching PE (previously allocated for the stream ID). If none of the PEs is allocated to the computed stream identifier (referred to as a *PE miss* event), a free PE (if any is free) is allocated to the stream identifier. The allocated PE transitions from the idle state *OFF* to the stride prediction state *SP*. An initial stride prediction is made to be the L1 cache line size. This prediction is stored in the PE and is referred to as the *program stride*. The missed address is also recorded in the PE and is referred to as the *program address*.

When the processor experiences a load miss, the corresponding instruction's stream identifier is used to find the matching PE. The difference between the missing address and the (previously recorded) program address is referred to as the *current stride*. The current stride is compared against the recorded program stride. If the two agree, a *stride hit* condition is detected and the stride is copied to another store in the PE, referred to as the *prefetch stride*. The program address is updated with the missing address. This same address is added to the prefetch stride and the result is recorded in another store of the PE referred to as the *prefetch address*. However, if the current stride is different from the program stride, then a *stride miss* condition is detected. In this case, the program stride and the program address are both updated with the computed stride and the missing address, respectively. This stride detection process repeats until either a strided stream is detected or the PE gets re-allocated to another stream identifier due to

¹PowerPC is a registered trademark of IBM Corporation

stream. Such a miss may be the result of structural, bandwidth or other limitations. These limitations do not necessarily mean that the PE is failing to effectively prefetch the required data for the program. Therefore, the PE is allowed to regain high confidence levels, without killing the whole stream as was done in prior techniques [9].

Feedback and Prefetched Moving Window: To overcome the cache pollution problem, stream buffers prefetching usually stores its prefetched data into dedicated prefetch buffers (hence the name). However, this solution means that accessing the prefetched data requires an additional number of cycles beyond accessing the L1 cache, which will reduce the benefits of prefetching. With LAP, prefetches are allowed to go directly to the L1 cache. Although this may result in cache pollution, the PE’s ability to minimize useless prefetches is helpful in reducing harmful effects, as illustrated in the experimental results of Section IV. The challenge associated with prefetching to the L1 cache is recognizing cache hits to prefetched cache lines and feeding this information back to the PEs to adjust their confidence levels accordingly. Reported work tags each L1 data cache line as being prefetched or not [16] to identify hits to prefetched data. This tagging increases the size of the L1 cache. In addition, the information is lost once the prefetched data is moved to lower memory levels. Instead of tagging the L1 cache lines, a heuristic called *Prefetched Moving Window* (PMW) is used in LAP.

The PMW is a range of addresses between the program’s last known accessed address and the most recent prefetched address. A cache hit that falls within the PMW is assumed to be to a prefetched cache line, otherwise the hit is ignored by the PE. The last known accessed address is updated with every hit or miss of the instruction assigned to the PE, which results in sliding the PMW as the program executes. Using the PMW enables tracking prefetched cache lines across the different memory levels without needing to tag every cache line in the memory hierarchy.

III. METHODOLOGY

The results shown in this work were obtained from a cycle-accurate simulator being developed in concert with a new high-performance micro-architecture from Freescale Semiconductor, Inc. The simulator models separate 32KB 8-way instruction (I) and data (D) caches, a unified 1MB 8-way L2 cache, an integrated peripheral bus, and dual integrated DDR controllers. The hit latency for the L1 cache is 3 cycles. The average total latency for an L2 cache hit is 25 cycles, and the average total latency for a DRAM read is 150 cycles.

This work uses the SPEC2000 suite of benchmarks. Several traces statistically represent each benchmark, such that each trace consists of several million instructions. Each trace has a weight representing its contribution to the overall benchmark. The reported numbers in this paper are those of the weighted sum of each trace metric. This is similar in concept to benchmark representation used in SimPoint [17]. Trace representativeness was verified against actual hardware (a previous processor) by using IPC as a verification metric.

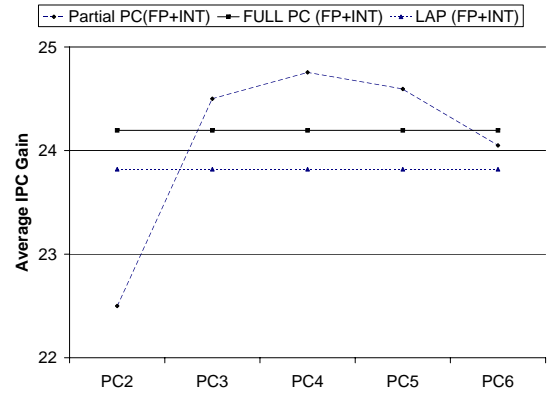


Fig. 2. Performance of PC bits.

The simulated IPC, obtained by simulating the traces on the matching cycle-accurate simulator and weighting the IPCs of each sub-trace appropriately, was compared against the IPC from execution on hardware. All benchmarks had less than a 15% delta between actual and simulated IPC.

IV. EXPERIMENTAL EVALUATION

In order to find the optimal subset of the PC to use in LAP, a number of bits (2,3,4,5 and 6) of the PC were used as a Stream_ID instead of the full PC. Because the instructions used in the simulation are all four bytes long, the least significant two bits were always zero. These two bits were first shifted out of the PC, and then the required number of bits (e.g. 2) was taken from the least significant bits remaining of the PC (e.g. bits 40 and 41 of the 44-bit PC, where bit 43 is the least significant).

Figure 2 shows the average percentage gain in IPC over no prefetching for all the benchmarks in both the SPEC2K suites (INT and FP). The gain is shown for three mechanisms: *Partial*, *Full* and *LAP*. Partial represents using a number of PC bits (2,3,4,5 and 6) to compose the Stream_ID. Full uses the full PC, and LAP uses 4 bits of the PC and 4 bits of *rA*. Figure 2 indicates that Partial using 4 bits of the PC results in the best performance gain. It is interesting to see that using this subset of the PC results in an overall gain of about 24.8%, while using the full PC results in a gain of about 24.2%. This added performance is a result of folding several streams into the same Stream_ID, which results in the engine reaching the active states faster than when only one load is being tracked. LAP is shown to have less performance advantage than Full with 4 bits. However, this loss is accompanied with large gain in accuracy as demonstrated in the next experiment.

Figure 3 shows the average prefetching accuracy of the above configurations. Accuracy is the percentage of used prefetches among all prefetches issued. The figure indicates that the accuracy has strong correlation with the number of PC bits used. It also indicates that combining *rA* with 4 bits of the PC in LAP comes within 4% of the accuracy achieved

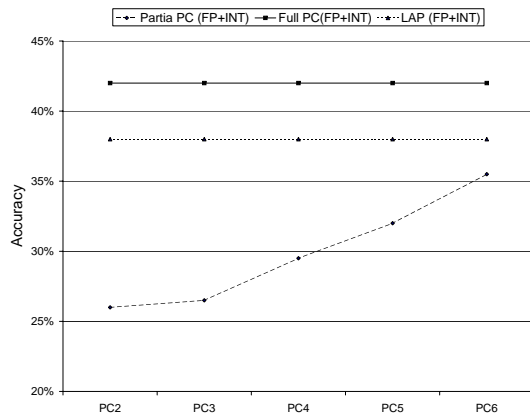


Fig. 3. Accuracy of PC bits.

when the full PC is used, and better than using only 4 bits of the PC by about 25%. The reasonable tradeoff between performance and accuracy of LAP makes it a good choice.

Figure 4 depicts the percentage gain in IPC for each benchmark of the SPEC2K suite for (1) LAP using 4 bits of the PC and rA and (2) using the full PC. The figure indicates that the SPEC2K INT suite of benchmarks contains a significant amount of regularity in its accesses. For example, the benchmark *181.mcf* gains about 200% in IPC with LAP. The reason for this gain is that *181.mcf* allocates its data structures at once, which results in all of its data nodes being spatially consecutive, and it walks that data structure in its allocation order in much of its processing. The figure also indicates that the proposed LAP performs very similar to using the full PC, except for the case of *188.ammp*, the reason for which is under investigation.

A similar trend in the accuracy for (1) LAP using 4 bits of the PC with rA and (2) using the full PC is presented in Figure 5. This figure shows the percentage of used prefetches among all prefetches issued to the lower memory levels, with an average of 40% for LAP compared to an average of 44% for the full PC.

Figure 6 depicts prefetch timeliness: the percentage of prefetches that were used from the L1 cache before they were replaced out of all used prefetches. This figure indicates that, on average, about 85% of the used prefetches were timely and completely masked the memory latency. It also demonstrates that the timeliness of LAP prefetching is similar to that of using the full PC.

Previous approaches to stream buffers prefetching stored prefetched data in buffers until they were needed by missing loads. This was done to avoid cache pollution by prefetches that are not accurate. In contrast, the LAP approach allows cache pollution and presents a comprehensive mechanism to minimize bad prefetches. Figure 7 illustrates the benefits of prefetching directly into the L1 cache, by presenting the IPC gain for two schemes. The first scheme is referred to as *Buffer*, which works by storing prefetches in a buffer and then accesses that buffer to acquire prefetched data for missing

loads when they get to the LSU. The second scheme, referred to as *L1 Cache*, writes prefetched data to the L1 cache as soon as they are serviced from lower memory levels. The reason for better gains with pollution is that accessing the separate prefetch buffers takes 4 cycles longer than accessing the L1 cache. The extra cycles occur because the buffers need to be accessed serially after determining the L1 cache miss. Otherwise, accessing the buffers in parallel with the L1 cache results in needing to drive several circuits (the cache and the buffer) at the same time, which will slow accessing the critical L1 cache. By polluting the L1 cache, these extra cycles are saved when prefetches are timely and accurate.

V. CONCLUSIONS

This paper presents LAP prefetching, a cost effective prefetching system that extends the traditional idea of stride prefetching by introducing a set of hardware components that accurately and dynamically detect exploitable address regularity. Potential prefetch addresses are generated by coordinating many pieces of strategically tagged in-flight information through a dedicated state machine. A confidence mechanism is proposed that enhances the accuracy of stride prefetching and allows the LAP approach to prefetch directly to the L1 cache overcoming the traditional limitations of cache pollution. A prefetched moving window heuristic is presented to enable pollution without the need to mark each cache line in the L1 cache. The experiments show that LAP prefetching using as few as 4 bits of the program counter along with other attributes of the instruction can come to within 1% of the performance achieved using a full PC.

REFERENCES

- [1] D. M. Pressel, "Fundamental limitations on the use of prefetching and stream buffers for scientific applications," in *Proceedings of the 10th ACM Symposium on Applied computing*, March 2001, pp. 554–560.
- [2] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *Proceedings of the seventh international conference on ASPLOS*, 1996, pp. 222–233.
- [3] B. Cahoon and K. S. McKinley, "Data flow analysis for software prefetching linked data structures in java," in *Proceedings of the 2001 international conference on PACT*, 2001, pp. 52–63.
- [4] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *Proceedings of the eighth international conference on ASPLOS*, 1998, pp. 115–126.
- [5] A. Roth and G. S. Sohi, "Effective jump-pointer prefetching for linked data structures," in *Proceedings of the 26th annual international symposium on Computer architecture*, 1999, pp. 111–121.
- [6] C.-L. Yang and A. R. Lebeck, "Push vs. pull: data movement for linked data structures," in *Proceedings of the 14th international conference on Supercomputing*, 2000, pp. 176–186.
- [7] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Cache-conscious structure layout," in *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, 1999, pp. 1–12.
- [8] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proceedings of the Fifth International Conference on ASPLOS*, Oct 1992, pp. 62–73.
- [9] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th ISCA*, May 1990, pp. 364–373.
- [10] P. C. K. Farkas and Z. Vranesic, "Memory system design considerations for dynamically scheduled processors," in *24th Annual International Symposium on Computer Architecture*, June 1997.

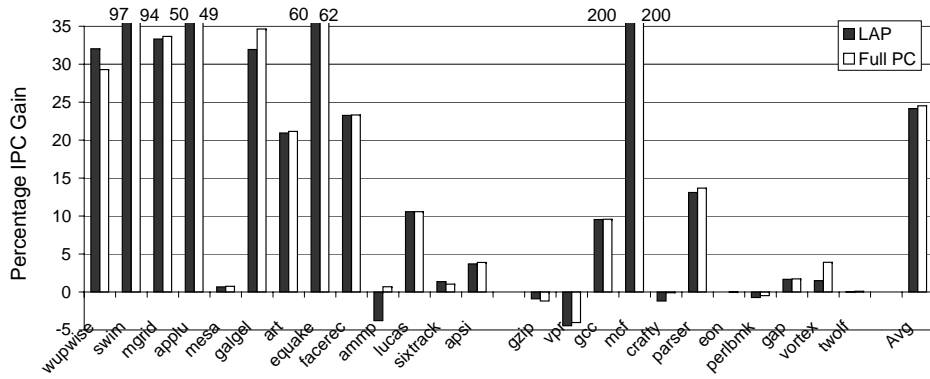


Fig. 4. Performance gain.

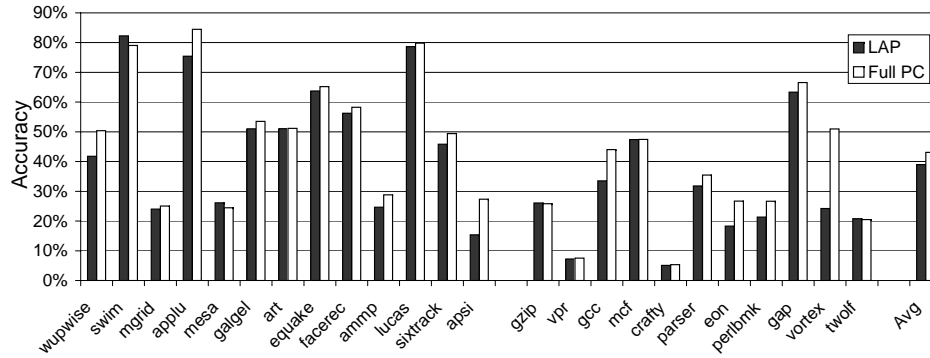


Fig. 5. Prefetching Accuracy.

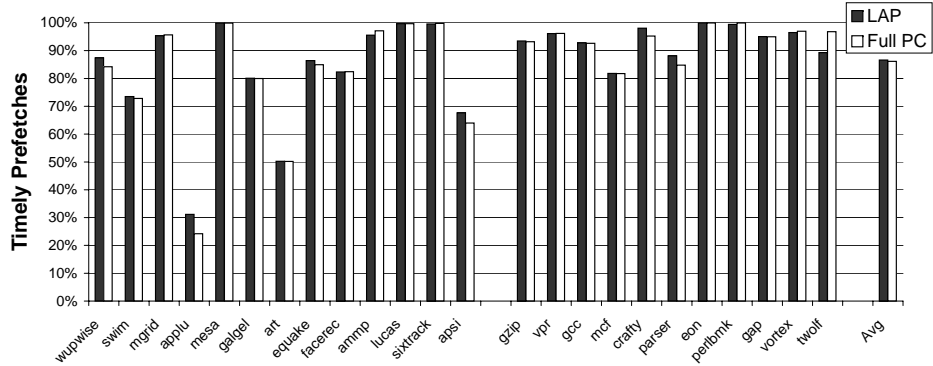


Fig. 6. Prefetch timeliness.

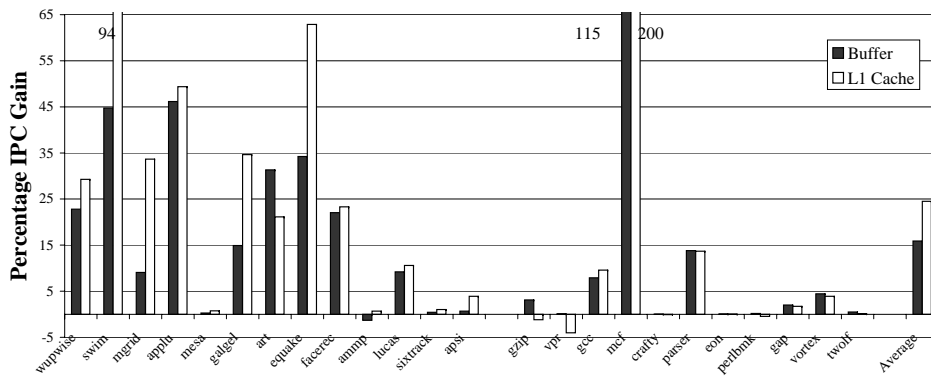


Fig. 7. Prefetching into the L1 cache.

- [11] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," in *Proceedings of the 29th ISCA*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 25–34.
- [12] S. Palacharla and R. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.
- [13] R. J. Eickemeyer and S. Vassiliadis, "A load-instruction unit for pipelined processors," *IBM Journal of Research and Development*, vol. 27, no. 4, pp. 547–564, July 1993.
- [14] H. Al-Sukhni, I. Bratt, and D. A. Connors, "Compiler directed content aware prefetching for dynamic data structures," in *Proceedings of the 2003 PACT International Conference*, 2003, pp. 91–100.
- [15] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," in *Proceedings of the 10th International Conference on ASPLOS*, October 2002, pp. 201–213.
- [16] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, "Effective stream-based and execution-based data prefetching," in *Proceedings of the International Conference on Supercomputing*, July 2004.
- [17] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on ASPLOS*, October 2002.