Received 30 May 2022, accepted 21 July 2022, date of publication 29 July 2022, date of current version 25 August 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3195046

RESEARCH ARTICLE

A Cross-Prefetcher Schedule Optimization Methodology

RĂZVAN NIȚU[1], LINGFENG PEI[2], AND TREVOR E. CARLSON[2], (Senior Member, IEEE)
[1]Automatic Control and Engineering Faculty, University POLITEHNICA of Bucharest, 060042 Bucharest, Romania
[2]Department of Computer Science, National University of Singapore, Singapore 119077

Corresponding author: Răzvan Niţu (razvan.nitu1305@upb.ro)

ABSTRACT Prefetching offers the potential to significantly improve performance by speculatively loading application data so that it is available before it is needed. By their very nature, prefetching techniques are application behavior dependant. This implies that no universal prefetching solution exists. A combination of prefetching strategies need to be used to target a diverse set of applications. In this work, we develop the first comprehensive mathematical framework that allows a designer to better understand the prefetching opportunities of an application. We first use dynamic analysis to study the memory access behavior of an application and measure a series of metrics to both identify the optimized schedule, and estimate its achievable performance. To validate our model, we implement and evaluate three different prefetching strategies: helper threads, software prefetching and FPGA prefetching. We show that, for each individual scenario, our framework correctly generates the optimized schedule of prefetches and predicts the performance improvement with an accuracy of more than 95%. Using our framework, developers can choose the best prefetching strategy and parameters for their specific workload and use case.

INDEX TERMS Analytical model, computer architecture, FPGA, optimization, prefetching, program analysis.

I. INTRODUCTION

As the speed gap between modern processors and the memory system is ever increasing [26], [56], the bottleneck of memory accessing in today's Von-Neumann machines becomes the pain-point that inspires various optimizing techniques such as caching [22] and prefetching [7], [8], [38], [46].

Prefetching is a fundamental technology of most high-performance systems today [24], [50], [53]–[55]. The goal of the prefetching is to retrieve, in a timely manner, data from a high latency memory, typically DRAM, and place it in fast-to-access cache memory. One key feature of a prefetcher is that it aims to fetch the data that is needed *before the computation unit accesses and uses it*. Prefetching can significantly reduce the time a CPU needs to wait when accessing data.

Existing prefetchers implemented in hardware [9], [14], [18]–[20], [28], [30], [31], [33], [38], [46], [47], [49], [51], [58] provide fixed-function operation and can not fundamentally change to adapt to the application, limiting attainable performance.

We argue that future prefetchers need to be configurable to support different strategies, possibly to the extent that they are configured by software. Memory access patterns are well known to be application dependent, which makes it hard to prefetch in an accurate and timely manner. For example, different prefetching distances, i.e. how far ahead the prefetcher sends requests, can lead to up to 10× variation in performance [32]. Therefore, we argue that prefetching needs to be driven by dynamic application behavior [3], [5], [32].

This opens up a large design space for the developer: Which prefetching strategy should be selected? When should a prefetch request be sent out? Is it worthwhile to keep the current strategy or is there a benefit to switch to a new one (while considering the potential overhead of this change)? Which parameters should one select if the prefetch strategy is parameterizable? Until now, such questions have not been possible to address in a systematic way.

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Pilato.

In this work, we propose a novel analytical framework that, based on measurements of application execution, can suggest close-to-optimal prefetcher strategies. Our framework provides two results. First, for a given prefetching strategy, the framework outputs an optimized schedule of prefetches that is both accurate and timely. The prefetching plan can be used to improve application performance. In our experiments, we show that the speed-up obtained while using the generated prefetch schedule is at or near optimal, seeing speed-ups between 1.16× and 2.05×.

Second, a performance estimate of the application that uses the mentioned prefetching schedule is computed. This estimate can be used to select between different prefetching strategies. In our experiments, the difference between the estimated and the measured speed-up is less than 5%.

Prior to our work, prefetching has been viewed as a black box. Developers have been using trial-and-error techniques for developing prefetchers hoping to meet their performance targets. In contrast, our framework brings transparency to prefetching by providing the analytical tools for a developer to understand the prefetching capabilities, or limitations, of an application that runs on a given system. Additionally, it offers the possibility to obtain the information required to select the best solution from a basket of options.

Below, we list the main contributions of this work.

- We propose a mathematical framework to both understand and predict potential prefetcher performance. The framework abstracts the technique of prefetching and is general enough to cover most prefetching scenarios.
- We develop a methodology of evaluating the prefetching capabilities of an application to allow developers to evaluate its suitability for a given hardware configuration.
- We describe how memory-level parallelism (MLP) for prefetching can close the gap to optimal performance.
- We evaluate the accuracy of our framework in the context of helper threads, software prefetching and FPGA prefetching.

The remainder of the paper is organized as follows: Section II details the motivation of this work and provides a high-level overview of our methodology, Section III extensively describes the analytical model, and Section IV details the methodology to apply our work. Section V presents our experimental results, Section VI discusses relevant work, and we present our conclusions in Section VIII.

II. MOTIVATION AND OVERVIEW

A. MOTIVATION

Though a plethora of prefetching techniques exist, no single solution can outperform all others in every situation [5], [8]. As such, understanding the applicability of a prefetching technique in a given scenario is fundamental to choosing the right solution. To that end we propose an analytical model that aids the programmer in understanding the prefetching capabilities of the application on a given system.

Software controlled prefetching techniques, such as software prefetching and helper threads, have a better view of the program in general; thus, they are more flexible and may adapt to a larger spectrum of applications than traditional hardware-based solutions [21]. However, up until this point, the proposed solutions for software controlled prefetching provide a trial-and-error mechanism of identifying the relevant values for parameters such as prefetch distance. In this work, we advance the state of the art by developing a mathematical framework that computes an optimized schedule of prefetches and estimates its performance.

B. METHODOLOGY OVERVIEW

Our work is based on the idea that prefetcher timeliness and the amount of work done between prefetch events are the two key metrics needed when developing an optimal software prefetching strategy.

To that end, we measure these characteristics to allow us to understand the potential prefetching benefits available to an application on a specific hardware platform (See Section III for details). These metrics are: (a) the time it takes the CPU and the prefetcher to access non-cache memory, (b) the time it takes the CPU to access the cache, (c) the available computation that is present between two consecutive cache misses triggered by the same load instruction and (d) the communication latency between the CPU and the prefetcher. The *cache* is typically the first level, but can refer to any level.

Figure 1 presents a high level overview of our proposed methodology. We first analyze the application to identify problematic load instructions and collect the mentioned metrics. We then input the designated prefetch technique and, if necessary, implement the prefetch kernel. A prefetch kernel is represented by the code that is run to compute the prefetch addresses and issue the prefetch requests.

We also analyze the data prefetch latency to compute the time it takes to prefetch a data item. Finally, we apply our mathematical formula to identify the optimized schedule and compute a performance estimate. Steps (2) to (4) may be repeated for any number of prefetching techniques to understand which strategy is best for the given application.

Previous work [3], [5] has used dynamic analysis to identify problematic loads, however, to the best of our knowledge, we are the first to push this analysis further by examining application runtime latencies. The benefit of this proposal is that we can use this information to both timely and accurately prefetch the necessary application data. By analyzing these values, we show that it is possible to understand what percentage of the total data accesses can be prefetched in a timely manner, and what it the best prefetching strategy for a given application.

III. ANATOMY OF PREFETCHING

A. RELEVANT METRICS

We utilize a set of metrics to describe prefetching and build our mathematical framework. For simplicity, we will consider

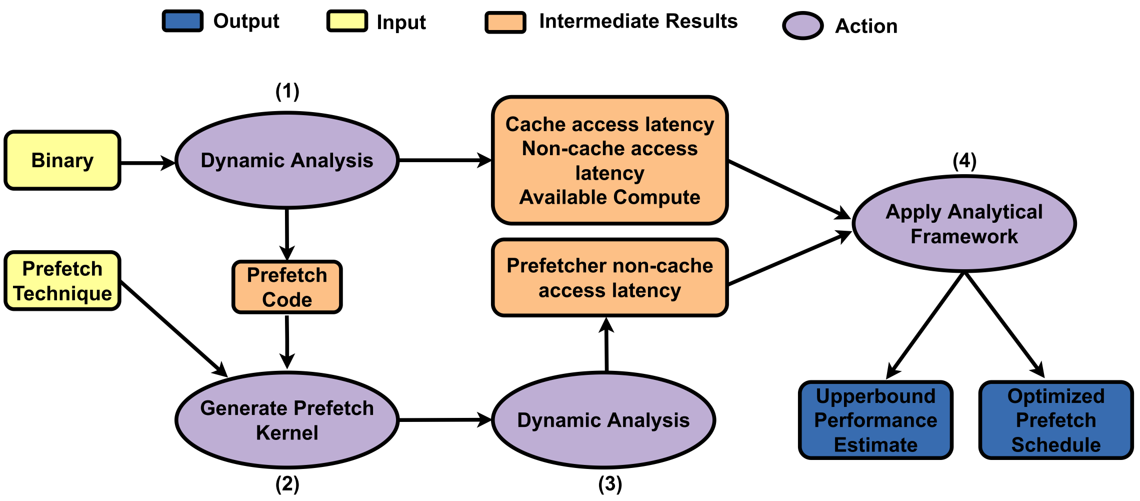

FIGURE 1. Methodology to evaluate an application for prefetching capabilities. (1) Analyze binary, (2) Generate prefetch kernel, (3) Analyze prefetcher, (4) Obtain the performance estimate and the optimized prefetch schedule.

only single program, single threaded workloads. However, the analytical framework can be extended to multi-program, multi-threaded applications.

T_{action}^{entity} is the time for the entity (*cpu* or *pf*) to perform an action (initialization (*init*), accessing the *cache*, or *mem*, and the latency of *compute*).

T_{mem}^{cpu} denotes the time it takes the CPU to read one data element from the high latency memory. Similarly, T_{mem}^{pf} represents the time it takes the prefetching implementation to perform the same operation. T_{cache}^{cpu} represents the time it takes the CPU to read one element of data from cache. Both T_{cache} and T_{mem} include the time that is necessary to compute the address for the read request. T_{init}^{pf} is the start up time for the prefetching implementation. It represents the time between the moment when the computation unit issues a prefetch request and the moment when the prefetching implementation actually starts running. $T_{compute}^{cpu}$ is used to indicate the time between 2 consecutive read requests issued by the CPU that have the same instruction pointer and that tend to miss in the cache.

B. FORMALIZING PREFETCHING

Given a specific application, we would like to determine an optimized strategy for prefetching considering T_{mem}^{cpu}, T_{mem}^{pf}, T_{cache}^{cpu}, $T_{compute}^{cpu}$ and T_{init}^{pf}.

Assuming that $T_{cache}^{cpu} + T_{compute}^{cpu} < T_{mem}^{cpu} + T_{compute}^{cpu}$, i.e., accessing data from cache is faster than accessing data from memory, there are 3 possible scenarios for the prefetching implementation:

1) $T_{mem}^{pf} < T_{cache}^{cpu} + T_{compute}^{cpu}$ This case may seem counter intuitive, however it is possible to achieve this by having a large amount of parallelism in the prefetching implementation or a large amount of computation time on the CPU side.
2) $T_{cache}^{cpu} + T_{compute}^{cpu} < T_{mem}^{pf} < T_{mem}^{cpu} + T_{compute}^{cpu}$ This is the case for current software and hardware prefetching techniques.
3) $T_{mem}^{pf} > T_{mem}^{cpu} + T_{compute}^{cpu}$ In this situation the ability to prefetch is severely limited, however benefits can still be obtained.

We start by assuming an ideal scenario and incrementally close the gap between it and real world situations.

1) INFINITE CACHE

We start by assuming that the system has an infinite amount of fully-associative cache memory, therefore once cache lines are allocated, they are never evicted. This enables us to first determine the maximum amount of data that can be prefetched in a timely manner. We assume that the working set is known. Next we consider N_{miss}, the number of cache accesses that miss. As the cache is infinitely sized, in order to avoid paying the communication latency between the prefetching implementation and the CPU, the CPU will issue a single request. In turn, the prefetching implementation will fetch the entire data set required for the application.

Scenario 1: $T_{mem}^{pf} < T_{cache}^{cpu} + T_{compute}^{cpu}$. If it takes the prefetching implementation less time to access one non-cached data item than it takes the computation unit to access a cached data item, then all of the N_{miss} data elements that are missing from the cache can be fetched in one pass. Assuming the prefetching implementation and the computation unit are launched at the same time, we pay a minor delay of T_{init}^{pf} that can be translated into N_{lost} number of accesses that are still going to miss before the CPU starts accessing the prefetched region of data:

$$N_{lost} = \frac{T_{init}^{pf}}{T_{mem}^{cpu}}. \tag{1}$$

Scenario 2: $T^{cpu}_{cache} + T^{cpu}_{compute} < T^{pf}_{mem} < T^{cpu}_{mem} + T^{cpu}_{compute}$. In this scenario, the prefetching implementation is able to fetch data ahead of the computation unit when the latter reads the data from non-cache memory. However, when the CPU starts accessing cached data it will eventually catch up. Without any prefetching involved, we can express the application total run time as:

$$N_{miss} * (T^{cpu}_{mem} + T^{cpu}_{compute}) \tag{2}$$

But if prefetching is going to be used, then some of these accesses are going to be turned in cache hits, therefore the new application total run time is going to be:

$$Y * (T^{cpu}_{mem} + T^{cpu}_{compute}) + (N_{miss} - Y) * (T^{cpu}_{cache} + T^{cpu}_{compute}) \tag{3}$$

where Y represents the number of cache misses that the CPU will still produce. During that time the prefetching implementation should be able to fetch the N_{miss}-Y data items, therefore, Equation 3 can also be expressed as:

$$(N_{miss} - Y) * T^{pf}_{mem} + T^{pf}_{init} \tag{4}$$

Equalising Equations 3 and 4 we are able to deduce Y:

$$Y = \frac{N_{miss} * (T^{pf}_{mem} - T^{cpu}_{cache} - T^{cpu}_{compute}) + T^{pf}_{init}}{T^{pf}_{mem} - T^{cpu}_{cache} + T^{cpu}_{mem}}. \tag{5}$$

The value of Y is optimal for prefetching. Since the cache is infinite, the optimal procedure is to start prefetching at the same time as the CPU starts its processing and fetch N_{miss}-Y data items starting with the Yth missing access.

Scenario 3: $T^{pf}_{mem} > T^{cpu}_{mem} + T^{cpu}_{compute}$. This situation is similar to the preceding one because there is still the need to sacrifice some accesses in order to prefetch others. However, in this case the number of sacrificed accesses is going to be very large because of the slow access time.

2) FINITE CACHE AND EVICTIONS

In this scenario we take a step closer to the real world. We assume that the cache has a fixed size, $Cache_{size}$, that is known both to the prefetcher and the computation unit, and therefore multiple requests should be issued to the prefetching implementation if the entire prefetchable data does not fit into the cache. In this situation, we may apply Equation 5, however, N_{miss} is replaced with a divisor of $Cache_{size}$. The occurrence of cache evictions cannot be identified in a deterministic manner because they depend on the overall system load, however, in practice, N_{miss} can be evaluated with different values until an optimized one is identified. In our experiments using Zynq hardware, we have seen that the ideal value for N_{miss} occurs when the prefetched data size for one prefetch request occupies $\frac{1}{4} * Cache_{size}$.

3) SPEED-UP

Given the above formulas we are also able to compute the maximum expected speed-up. The total run time of an application without any prefetching is computed using Equation 2.

TABLE 1. FPGA platform.

Name	Zynq XC7Z020-1CLG400C
Processor	Cortex-A9 processor
Number of Cores	2
Frequency	650MHz
Memory Controller	DDR3, 8 DMA channels, 4 AXI3 slave ports
Programmable logic	Artix-7 family

The run time of the application with prefetching involved is computed using Equation 3. Therefore, the speed-up is computed as the division of the two:

$$S = \frac{N_{miss} * (T^{cpu}_{mem} + T^{cpu}_{compute})}{Y * (T^{cpu}_{mem} - T^{cpu}_{cache}) + N_{miss} * (T^{cpu}_{cache} + T^{cpu}_{compute})}. \tag{6}$$

IV. METHODOLOGY

To identify the values for the parameters discussed in section III-A we take the following steps. We dynamically analyze the application to identify the number of missing loads that occur and the responsible loops that cause them. We collect this information by performing test runs on real hardware (although, simulation techniques can also be employed). Next, we work to understand the application source code to identify the memory access patterns of the application. We ask the following questions: how many iterations does the loop have? How many cache misses occur per iteration? This step can be done manually or automatically as discussed by Ayers et. al [5]. By combining the knowledge obtained in the previous steps, we differentiate between loads that are part of the address generation of another load and loads that fetch actual data needed for the computation. After this step, we can divide the total execution time of the loop to the number of data loads to obtain the approximate value of $T^{cpu}_{mem} + T^{cpu}_{compute}$. To identify the value of $T^{cpu}_{cache} + T^{cpu}_{compute}$ we apply the same procedure, except that we populate the cache, in advance, with the otherwise missing data. An alternative is to use simulation to obtain this information. Dividing the resulting runtime by the number of cache misses that we obtained earlier, we arrive at the value of $T^{cpu}_{cache} + T^{cpu}_{compute}$. Note that it is not necessary to separate the compute value from the access latency because they are both present together in all formulas. Next, we implement the prefetch kernel, according to the chosen prefetching strategy, that performs prefetch requests for the faulting loads. We measure the runtime of the prefetching implementation and we divide it by the same number of cache misses that we used earlier. This operation will result in identifying the value of T^{pf}_{mem}. The value of T^{pf}_{init} is identified by measuring the runtime of a prefetcher implementation that does not perform any operation. In the case of software prefetching and helper threads, we consider this value to be negligible. After we identify the corresponding values of the relevant metrics, we update the original program to issue prefetch requests using the derived parameters. In this work, we tackle single process, single

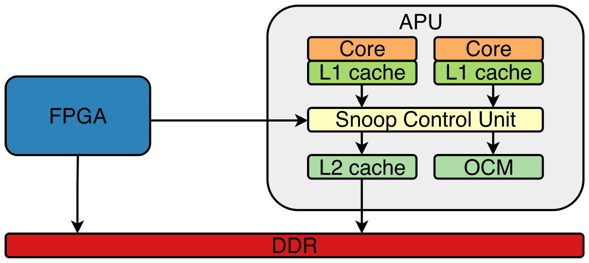

FIGURE 2. Zynq architecture. The FPGA can perform L2 cache coherent requests.

program workloads that have a deterministic memory access behavior (i.e. the work set is known in advance).

V. EVALUATION

To apply our analysis, we use a Xilinx Zynq Z1 board [29]. Figure 2 highlights the architecture of the board and Table 1 presents component details. The Application Processing Unit (APU) consists of two cores, each with its private L1 cache. The processors share the L2 cache and the On Chip Memory (OCM). The FPGA has a direct link to the L2 cache memory through the accelerator coherency port (ACP).

This architecture is flexible enough to support multiple prefetching strategies: (1) helper threads by using a core to prefetch data for the other core, (2) software prefetching by using the processors' preload instruction and (3) FPGA prefetching by using the FPGA to prefetch data for one (or both) of the CPUs.

We analyze and optimize three micro-benchmarks and two realistic applications. The selection of applications covers the most common memory access patterns and the relevant aspects presented in Section III-B.

A. BENCHMARK DESCRIPTION

Simple stride and **Complex stride** represent two demonstration micro-benchmarks developed by us. Both have $T^{cpu}_{compute}$ close to 0 and compute the sum of a number of array elements. Simple stride exhibits the *a[i]* pattern, whereas complex stride is of the form *a[4*i*(i+1)]*. While hardware stride prefetching can handle *simple stride*, it has difficulty with *complex stride*.

IntSort is part of the NAS Parallel Benchmark suite [6]. The main computation path is formed by a loop containing an indirect memory access and almost no extra computation besides the address calculation.

LBM is part of the SPEC CPU2006 [27] suite and features stride array accesses with large amounts of compute. In this situation, even if prefetching is perfect, the speed-up is limited by the amount of compute present in the loop.

Linked list. To increase the memory level parallelism in this benchmark, we have implemented a linked list by using jump pointers similar to previous work [47]. By using jump pointers, we are effectively introducing MLP.

To obtain the baseline performance measurements, we simply measure the runtime of the mentioned applications without performing any type of prefetching.

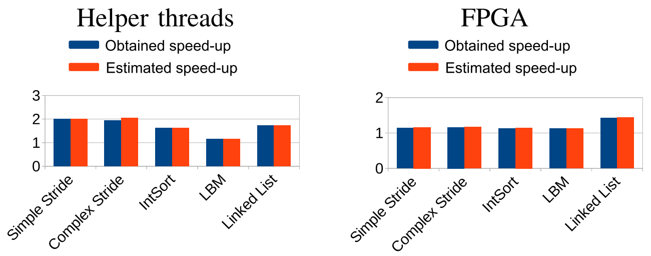

FIGURE 3. Helper threads and FPGA prefetching speed-up over no prefetching. The difference between the predicted and the obtained speed-ups is less than 5%.

TABLE 2. Helper thread metrics (latencies in microseconds).

Benchmark	T^{pf}_{mem}	$T^{cpu}_{mem} + T^{cpu}_{compute}$	$T^{cpu}_{cache} + T^{cpu}_{compute}$	T^{pf}_{init}	**% compulsory misses**
Simple Stride	0.050	0.070	0.019	0	33.6
Complex Stride	0.072	0.117	0.040	0	22.0
IntSort	0.164	0.164	0.058	0	38.0
LBM	0.090	0.412	0.360	0	0.0
Linked List	0.180	0.180	0.020	0	41.6

B. HELPER THREADS

For each of the benchmarks, we have implemented an additional prefetch helper thread using the pthreads library (the Zynq processor has two cores). In this scenario, prefetching is done into the L1 cache of the other processor. The main thread occasionally sends prefetching requests to the prefetcher thread by specifying the start address and the chunk size.

C. SOFTWARE PREFETCHING

We perform software prefetching for each of the benchmarks by introducing a preload instruction in the loop bodies of our experimental applications. The data is prefetched into the L1 cache of the CPU. We have determined the baseline prefetch distance by using a trial-and-error strategy. We use software prefetching only to synthetically add compute - see Section V-F - because the nature of software prefetching does not permit the issuance of multiple prefetch requests per iteration. This limitation arises from the fact that address computation instructions are still executed by the processor, consuming pipeline slots. For tight loops, these instructions may require more time to compute than the original work performed inside the loop. However, our framework may still accurately predict the speed-up for this scenario, as seen in Figure 4.

D. FPGA PREFETCHING

We implement FPGA prefetchers using Vivado HLS and Design 2019.1 for each of the mentioned benchmarks. The FPGA prefetcher is able to fetch the data into the shared L2 cache of the Zynq processor. Alternatively, the OCM could be used for prefetching, however, the OCM exhibits the same access latency as the L2 cache [43], [48], but is non-coherent. Similar to the helper thread implementation, the application running on the CPU triggers the FPGA prefetcher.

TABLE 3. FPGA prefetching - metrics for each benchmark (latencies are measured in microseconds).

Benchmark	T_{mem}^{pf}	T_{mem}^{cpu} + $T_{compute}^{cpu}$	T_{cache}^{cpu} + $T_{compute}^{cpu}$	T_{init}^{pf}	% compulsory misses
Simple Stride	0.140	0.048	0.024	1.5	72.5
Complex Stride	0.162	0.056	0.017	1.5	73.2
IntSort	0.203	0.080	0.047	1.5	66.1
LBM	0.132	0.300	0.255	1.5	0.0
Linked List	0.156	0.152	0.044	1.5	43.0

E. PERFORMANCE RESULTS

Table 2 and Table 3 present the values of the observed metrics. It can be seen that the FPGA has a high latency link to DRAM through the L2 cache controller of the CPU. Although we have utilized the full parallelism potential in the FPGA, the architectural constraints, such as total number of outstanding memory requests, severely limit the prefetching capabilities. As a result, only a small percentage of the total data items can be prefetched. In contrast, the helper thread implementation benefits from a shorter DRAM access latency and therefore is able to prefetch a larger portion of the data accesses.

Figure 3 highlights the obtained speed-up by applying the optimized schedule of prefetches obtained from our framework and compares it to the expected value that is computed by using our formulas. Since the helper thread implementation has a smaller time to access the non-cached memory, it performs better in all of the tested scenarios. It can also be observed that in all situations the difference between the expected speed-up and the measured one is less that 5%.

F. COMPUTE

To test our assumption that the best scenario for prefetching is obtained when $T_{mem}^{pf} <= T_{cache}^{cpu} + T_{compute}^{cpu}$ we have synthetically added compute to the benchmarks and observed the effect on performance.

Figure 4 highlights the impact of synthetically adding compute to the applications on ideal and measured speed-up. The ideal speed-up is computed by using our framework and represents the speed-up that would be obtained if all of the accessed data items would be present in the cache when the processor needs them. While the amount of compute increases per iteration, the ideal speed-up decreases because the benefits of prefetching are overshadowed by the added compute. However, there is more time for the prefetching implementation to bring the needed data into the cache. Therefore, we observe that in order to be able to prefetch all of the data elements, it is necessary that $T_{mem}^{pf} <= T_{cache}^{cpu} + T_{compute}^{cpu}$. Theoretically, this can be achieved either by increasing the amount of compute per loop iteration or by issuing additional prefetch requests in parallel. For example, by adding another prefetcher helper thread, it is expected that the T_{mem}^{pf} will decrease, ideally, by a factor of 2. The implication for systems is that, **by improving the memory-level parallelism (MLP) available to prefetchers, one can reduce the gap between achievable and optimal prefetching performance**.

TABLE 4. Memory access patterns.

Pattern	Example
Constant	*ptr
Stream	a[i]
Indirect	a[b[i]]
Pointer chase	next = current→next

Our experiments show that by looking at the runtime latencies of data accesses, it is possible to understand the amount of prefetching available in a given scenario and how to schedule the prefetch requests. Moreover, using this information it is possible to compute a performance estimate of the prefetch-enhanced application.

VI. RELATED WORK

Prefetching is a standard technique that has been used in many different ways and in many situations. We briefly outline relevant work and elaborate, further, our approach.

For prefetcher performance it is important to understand the memory access patterns of algorithms and applications. Ayers *et al.* [5] have recently reinforced this and they developed a classification of memory access patterns, highlighted in Table 4, that can be used to express most memory access types. This classification offers insights into whether a specific type of prefetching is suitable for a specific workload. We note that the same algorithm and indeed application can exhibit different access patterns in different phases of execution.

Prefetching can be implemented in hardware, software, as well as a combination of hardware and software. Table 5 groups similar prefetching techniques into categories and highlights the relevant attributes of each technique.

Hardware prefetching techniques require a specialized physical unit that handles the monitoring of memory accesses and automatically generates prefetch requests. This unit is commonly tightly coupled to the execution unit, normally a processor core. This allows for low latency communication between the core and the prefetch hardware unit. The hardware units tend not to support anything but a general prefetch method which may not be optimal for all algorithms or applications. In our work and in this paper, we show that latency is not crucial for performance allowing a loosely-coupled and program controlled accelerator to carry out prefetching effectively. This also allows the prefetchers in our approach to implement specialized and more complicated prefetch methods.

There exist several different types of methods commonly implemented by hardware prefetch units. This includes stride, history based and irregular prefetchers.

Stride prefetchers [14], [20], [28], [30], [38], [49] represent the most common form of hardware prefetcher employed in current systems. Simple and easy to implement, stride prefetchers benefit a subset of memory access patterns [35], namely, regular streaming access patterns. For other types, the stride prefetcher may actually worsen performance by replacing useful data with prefetched data that is not used.

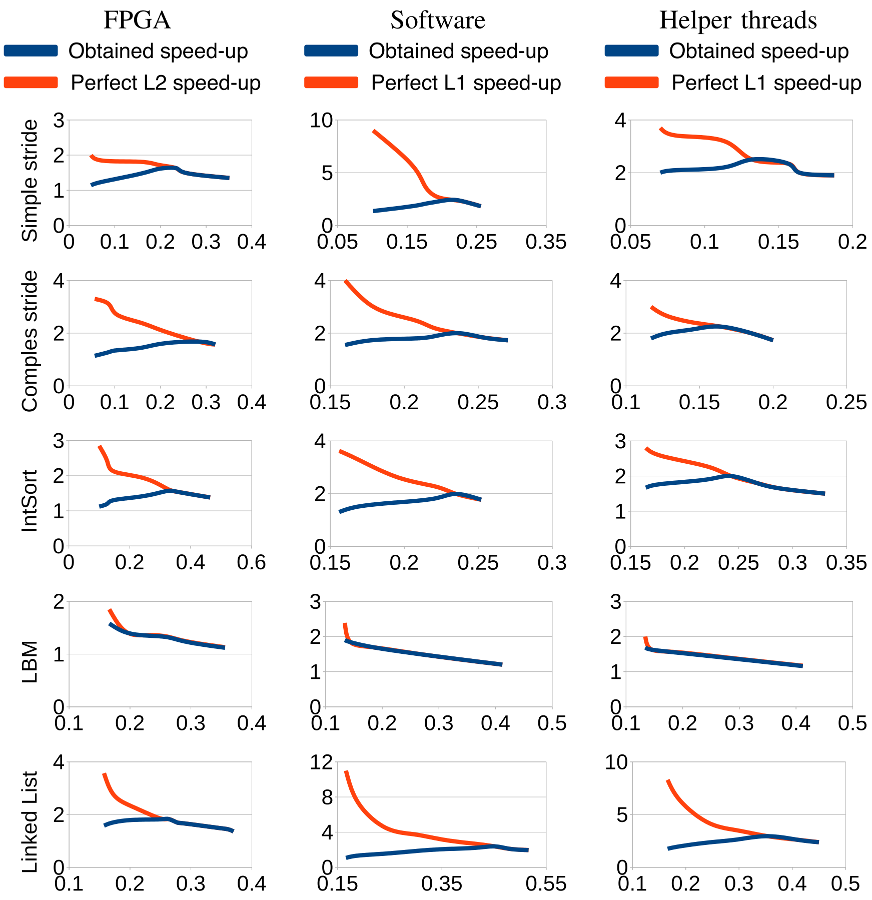

FIGURE 4. **The effect of adding synthetic compute on the ideal speed-up vs. the measured speed-up. The x-axis represents the added compute, measured in microseconds. The y-axis represents the speed-up obtained. The best performing configuration for prefetching is represented as the convergence point of the ideal (orange) and real speed-up (blue).**

History based prefetchers [31], [33], [46], [51] have the ability to prefetch more complex access patterns by storing a sequence of prior accesses and predict future accesses based on it. However, to achieve good performance, history based prefetchers require a large amount of memory, up to megabytes, to store the necessary information. In addition, pointer-chasing and indirect memory patterns are not supported because of their irregular nature.

Irregular prefetchers target complex access patterns (pointer chasing and indirect) and can be divided into 2 categories: specialized and general.

Specialized irregular hardware prefetching units target a single access pattern. Multiple solutions have been proposed for both pointer-fetching prefetchers [18], [19], [47] and indirect access prefetchers [9], [58]. Although these units provide significant performance benefits, they lack generality.

Run-ahead execution prefetchers [25], [44], [45] may prefetch many types of memory access patterns by speculatively pre-executing the program's own code. By closely mimicking the access patterns of the application, this technique is highly general, supporting many types of memory access patterns. However, this approach requires prohibitive amounts of analysis hardware to identify the instruction streams that cause cache misses. Once identified, the instruction stream is executed ahead of time on a separate core. This leads to the inability of prefetching data for loads that contain a long latency load in their address computation.

Although hardware prefetching techniques may prove beneficial in certain scenarios, they lack the flexibility required to adapt to any kind of access pattern. We overcome this limitation by dynamically analyzing the application before execution and specifically targeting the long latency loads.

Software prefetching techniques rely on prefetch hints or instructions that are inserted in the source code. These generate pre-load instructions that are executed before the actual load. These instructions are committed immediately and therefore do not stall the pipeline. This approach has the advantage that it does not require extra hardware since most architectures implement a form of prefetch instruction. However, software prefetching techniques suffer from two major shortcomings: (1) inserting prefetch instructions that accurately target long latency loads is difficult and (2) accesses that involve multiple long latency loads will continue to stall the pipeline and therefore require extra computation that masks the prefetch.

TABLE 5. **Prefetching techniques.**

Approach	SW	HW	Pattern	Prefetch Schedule Analysis
Stride - hardware [14], [20], [28], [30], [38], [49]	✗	✓	Simple stream	Dynamic at runtime
Pointer fetching - hardware [18], [19], [47]	✗	✓	Pointer chase	Dynamic at runtime
Indirect - hardware [9], [58]	✗	✓	Indirect	Dynamic at runtime
History based [31], [33], [46], [51]	✗	✓	Complex stream	Dynamic at runtime
Run-ahead [25], [44], [45]	✗	✓	All	Dynamic at runtime
Helper threads - hardware [12], [13], [16], [17], [41]	✓	✓	All	Static
Software prefetching stride - static [23] [52]	✓	✗	Stride	Static
Software prefetching stride - dynamic [40] [42]	✓	✗	Stride	Dynamic upfront
Software prefetching indirect [2]	✓	✗	Indirect	Static
Software prefetching - all patterns [5]	✓	✗	All	Dynamic upfront
Helper threads - software [11], [34], [36], [37]	✓	✗	All	Static
Pointer fetching - software [4], [15], [39], [57]	✓	✓	Pointer chase	Static
Programmable prefetcher [3]	✓	✓	All	Static
FPGA prefetching (this work)	✓	✓	All	Dynamic upfront

Static analysis prefetching relies on the compiler to (1) identify memory accesses that will cause a cache miss and (2) automatically insert prefetch instructions. Due to the static nature of the analysis, the set of patterns identified is limited to simple stride [23], [52] and indirect [2] accesses. However, in most cases, the speed-up resulted from static analysis prefetching is inferior to manually inserted prefetching code.

Dynamic analysis prefetching [40], [42] [5] leverages runtime information with regard to the last level cache miss of each load instruction request to appropriately target them for software prefetching. The prefetch instructions are then manually inserted in the source code. This approach offers the benefit of accurately identifying the problematic loads at the cost of extra upfront dynamic analysis.

In this work, we adopt dynamic analysis and manually insert prefetch triggers and allow for multiple accesses to occur in parallel. Our model can be used to identify the roofline speed-up for ideal software prefetching.

Helper threads [11]–[13], [16], [17], [34], [36], [37], [41] tackle prefetching by statically extracting the code for delinquent loads and running it on a spare thread context. This approach can optimally target any access pattern by increasing the number of helper threads. Furthermore, it is flexible enough to be implemented both in hardware [12], [13], [16], [17], [41] and software [11], [34], [36], [37]. However, even using a single extra thread comes at an increased energy penalty on high performance cores. Moreover, accesses that require loads in their address computation will stall, and in the absence of a hardware event queue, the synchronization of loads becomes costly in terms of both implementation and performance.

Programmable hardware techniques employ specialized hardware units that are able to run specific address computation instructions. Jones *et al.* have proposed a programmable prefetcher specifically designed for graph workloads that targets specific traversals [1]. Yi *et al.* have designed a hybrid prefetcher that targets indirect memory accesses [10]. Several approaches have targeted linked list data structures [4], [15], [39], [57]. A more general approach has been developed by Ainsworth and Jones [3] that uses multiple small in-order cores to run prefetch kernels that are indicated in software. This work has shown significant speed-ups for load-intensive applications, however, the design is not able to deal with the pointer chasing pattern and the prefetch kernel size is limited to only a few instructions, whereas previous work [5] reports prefetch kernels that require up to 80 instructions.

Summary. Prefetching is a well studied technique and a large range of solutions have been proposed, both general and pattern specific. Each technique has its strengths and weaknesses as highlighted in Table 5.

To better the prefetching potential of an application, we have devised an analytical framework that evaluates prefetching in a given scenario and helps computer architects understand what are the optimal conditions for prefetching.

Our approach uses dynamic analysis to (1) identify the instructions that cause long latency loads and (2) to determine what are the access latencies for both cached and non-cache accesses. Using this information, we determine the optimal schedule of prefetches for a given prefetching technique, taking into account hardware limitations.

VII. DISCUSSION

In this work we have demonstrated that prefetching performance can be predicted and we have tested our model for single program, single processor applications. We have performed our measurements both on top of an operating system and on the baremetal hardware. However, we have not experimented with various degrees of system utilization. Since an application's optimal prefetch schedule depends on the load of the system at a specific point, this aspect remains to be investigated in future work. One idea that could be used to improve the current status is to simulate the application in a maximally utilized system and deduce the worst case scenario values for the metrics. By using these worst case values, it is possible to tune our prefetch schedule so that it performs optimally irrespective of the system load.

VIII. CONCLUSION

Some might see prefetching as a black-box, where one attempts to optimize the strategy in a trial-and-error fashion. As an alternative, this work has taken the first steps toward a rigorous analysis of prefetching, opening the door to new possibilities for both hardware and software systems.

In this work, we propose a novel mathematical framework to abstract prefetching into its fundamental components. With this understanding, one can now, in an up-front manner, determine how much prefetching can improve the performance of key workloads. Our methodology applies to specific hardware/software pairs under study to present a variety of potential prefetching solutions.

In addition to presenting a new analytical understanding of prefetching, in this work we present how one can optimize FPGA, helper-thread and software-prefetching-based systems to maximize performance. The result is a significant speed-up for a set of applications that are among the most difficult to optimize (those without a significant amount of compute that can be used to hide the memory latency). Understanding the system requirements with prefetching can also lead to improved hardware designs that can take advantage of the level of optimization provided by our methodology.

This work presents a hardware-validated model and methodology that can accurately predict high-performing prefetching schedules.

ACKNOWLEDGMENT

The authors would like to express their gratitude to Dr. Sven Karlsson and Udaree Kanewala for their valuable and constructive suggestions during the planning and construction of this work. This work was supported by Singapore MoE AcRF grants T1-251RES1705 and T1-251RES2102.

REFERENCES

[1] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proc. Int. Conf. Supercomputing*, Jun. 2016, pp. 1–11.

[2] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Feb. 2017, pp. 305–317.

[3] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," in *Proc. 23rd Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, Mar. 2018, pp. 578–592.

[4] H. Al-Sukhni, I. Bratt, and D. A. Connors, "Compiler-directed content-aware prefetching for dynamic data structures," in *Proc. Oceans Conf. Exhib. Conf.*, Sep./Oct. 2003, pp. 91–100.

[5] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, Mar. 2020, pp. 513–526.

[6] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, D. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and K. Weeratunga, "The NAS parallel benchmarks," *Int. J. High Perform. Comput. Appl.*, vol. 5, no. 3, pp. 63–73, 1991.

[7] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2019, pp. 399–411.

[8] M. Bakhshalipour, S. Tabaeiaghdaei, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Evaluation of hardware data prefetchers on server processors," *ACM Comput. Surveys*, vol. 52, no. 3, pp. 1–29, May 2020.

[9] M. Cavus, R. Sendag, and J. J. Yi, "Array tracking prefetcher for indirect accesses," in *Proc. IEEE 36th Int. Conf. Comput. Design (ICCD)*, Oct. 2018, pp. 132–139.

[10] M. Cavus, R. Sendag, and J. J. Yi, "Informed prefetching for indirect memory accesses," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 1, pp. 1–29, 2020.

[11] C. Jung, D. Lim, J. Lee, and Y. Solihin, "Helper thread prefetching for loosely-coupled multiprocessor systems," in *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp. (PDPS)*, 2006, pp. 1–10.

[12] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt, "Microarchitectural support for precomputation microthreads," in *Proc. 35th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2002, pp. 74–84.

[13] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (SSMT)," in *Proc. 26th Int. Symp. Comput. Archit. (ISCA)*, 1999, pp. 186–195.

[14] T.-F. Chen and J.-L. Baer, "Reducing memory latency via non-blocking and prefetching caches," in *Proc. 5th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, 1992, pp. 51–61.

[15] S. Choi, N. Kohout, S. Pamnani, D. Kim, and D. Yeung, "A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching," *ACM Trans. Comput. Syst.*, vol. 22, no. 2, pp. 214–280, May 2004.

[16] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *Proc. 28th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2001, pp. 14–25.

[17] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *Proc. 34th ACM/IEEE Int. Symp. Microarchitecture (MICRO)*, 2001, pp. 306–317.

[18] J. Collins, S. Sair, B. Calder, and D. M. Tullsen, "Pointer cache assisted prefetching," in *Proc. 35th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2002, pp. 62–73.

[19] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," in *Proc. 10th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, 2002, pp. 279–290.

[20] F. Dahlgren and P. Stenstrom, "Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors," in *Proc. 1st IEEE Symp. High Perform. Comput. Archit. (HPCA)*, Jan. 1995, pp. 68–77.

[21] B. Falsafi and T. F. Wenisch, "A primer on hardware prefetching," *Synth. Lectures Comput. Archit.*, vol. 9, no. 1, pp. 1–67, May 2014.

[22] A. Farshin, A. Roozbeh, G. Q. Maguire, and D. Kostić, "Make the most out of last level cache in Intel processors," in *Proc. 14th EuroSys Conf.*, Mar. 2019, pp. 1–17.

[23] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, "Compiler-directed data prefetching in multiprocessors with memory hierarchies," in *Proc. Int. Conf. Supercomput. 25th Anniversary*, 1990, pp. 128–142.

[24] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G. Chiu, P. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, Mar./Apr. 2012.

[25] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous runahead: Transparent hardware acceleration for memory intensive workloads," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2011.

[27] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[28] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2006, pp. 397–408.

[29] Digilent. *PYNQ-Z1*. Accessed: Feb. 29, 2022. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/pynq-z1/start

[30] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," in *Proc. 23rd Int. Conf. Supercomput. (ICS)*, 2009, pp. 499–500.

[31] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2013, pp. 247–259.

[32] S. Jamilan, T. A. Khan, G. Ayers, B. Kasikci, and H. Litz, "APT-GET: Profile-guided timely software prefetching," in *Proc. 17th Eur. Conf. Comput. Syst.*, Mar. 2022, pp. 747–764.

[33] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proc. 24th Annu. Int. Symp. Comput. Archit. (ISCA)*, 1997, pp. 252–263.

[34] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," in *Proc. 16th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, 2011, pp. 393–404.

[35] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proc. 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2015, pp. 158–169.

[36] D. Kim and D. Yeung, "Design and evaluation of compiler algorithms for pre-execution," in *Proc. 10th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, 2002, pp. 159–170.

[37] D. Kim and D. Yeung, "A study of source-level compiler algorithms for automatic construction of pre-execution code," *ACM Trans. Comput. Syst.*, vol. 22, no. 3, pp. 326–379, Aug. 2004.

[38] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[39] N. Kohout, S. Choi, D. Kim, and D. Yeung, "Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes," in *Proc. Int. Conf. Parallel Architectures Compilation Techn. (PACT)*, 2001, pp. 268–279.

[40] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen, "The performance of runtime data cache prefetching in a dynamic optimization system," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2003, pp. 180–190.

[41] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham, "Dynamic helper threaded prefetching on the sun UltraSPARC CMP processor," in *Proc. 38th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Nov. 2005, pp. 1–12.

[42] C.-K. Luk, R. Muth, H. Patil, R. Weiss, P. G. Lowney, and R. Cohn, "Profile-guided post-link stride prefetching," in *Proc. 16th Int. Conf. Supercomput. (ICS)*, 2002, pp. 167–178.

[43] S. W. Min, S. Huang, M. El-Hadedy, J. Xiong, D. Chen, and W.-M. Hwu, "Analysis and optimization of I/O cache coherency strategies for SoC-FPGA device," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 301–306.

[44] 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 *Proc. 9th Int. Symp. High-Perform. Comput. Archit., (HPCA)*, 2003, pp. 129–140.

[45] O. Mutlu, H. Kim, and Y. N. Patt, "Techniques for efficient processing in runahead execution engines," in *Proc. 32nd Int. Symp. Comput. Archit. (ISCA)*, 2005, pp. 370–381.

[46] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 118–131.

[47] A. Roth and G. S. Sohi, "Effective jump-pointer prefetching for linked data structures," in *Proc. 26th Int. Symp. Comput. Archit. (ISCA)*, 1999, pp. 111–121.

[48] M. Sadri, C. Weis, N. Wehn, and L. Benini, "Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ," in *Proc. 10th FPGAworld Conf. (FPGAworld)*, 2013, pp. 1–8.

[49] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proc. 48th Int. Symp. Microarchitecture (MICRO)*, Dec. 2015, pp. 1–12.

[50] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu, "Knights landing: Second-generation Intel Xeon Phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar./Apr. 2016.

[51] Y. Solihin, J. Lee, and J. Torrellas, "Using a user-level memory thread for correlation prefetching," in *Proc. 29th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2002, vol. 30, no. 2, pp. 171–182.

[52] S. W. Son, M. Kandemir, M. Karakoy, and D. Chakrabarti, "A compiler-directed data prefetching scheme for chip multiprocessors," in *Proc. 14th ACM SIGPLAN Symp. Princ. Pract. Parallel Program. (PPoPP)*, 2008, pp. 209–218.

[53] D. Suggs, M. Subramony, and D. Bouvier, "The AMD 'Zen 2' processor," *IEEE Micro*, vol. 40, no. 2, pp. 45–52, Mar./Apr. 2020.

[54] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM J. Res. Develop.*, vol. 46, no. 1, pp. 5–25, Jan. 2002.

[55] K. Viswanathan, *Disclosure of Hardware Prefetcher Control on Some Intel Processors*. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html

[56] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[57] C.-L. Yang and A. Lebeck, "A programmable memory hierarchy for prefetching linked data structures," in *High Performance Computing* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2002.

[58] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in *Proc. 48th Int. Symp. Microarchitecture (MICRO)*, Dec. 2015, pp. 178–190.

RĂZVAN NIȚU received the B.Sc. and M.Sc. degrees in computer science and engineering from the University POLITEHNICA of Bucharest (UPB), Bucharest, Romania, where he is currently pursuing the Ph.D. degree in programming languages and security. His research interests include programming languages, security, computer architecture, and education techniques.

LINGFENG PEI received the B.S. degree from the Huazhong University of Science and Technology, majoring in the IoT engineering, and the M.Sc. degree in computer science from the National University of Singapore (NUS), where she is currently pursuing the Ph.D. degree in computer science. Her current research interests include prefetching, hardware security, and FPGAs.

TREVOR E. CARLSON (Senior Member, IEEE) is currently an Assistant Professor at the National University of Singapore working to develop high-efficiency microarchitectures that can meet the performance and needs of the future IoT and server applications. He also co-develops the Sniper Multi-Core Simulator.