Compiler-Assisted Data Streaming for Regular Code Structures

Nuno Neves, *Member, IEEE,* Pedro Tomás, *Senior Member, IEEE,* and Nuno Roma, *Senior Member, IEEE*

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Rua Alves Redol, 9, 1000-029 Lisboa - Portugal
Email: {nuno.neves,pedro.tomas,nuno.roma}@inesc-id.pt

Abstract—The performance of modern processors is often limited by execution stalls resulting from long memory access latencies. Compile-time optimizations, deep cache hierarchies and prefetching mechanisms already provide significant performance gains, by performing memory accesses in parallel with computation. However, they are reaching a throughput improvement limit. Hence, new solutions that effectively exploit the memory access patterns to improve processing throughput are required. To achieve this objective, a new compiler-assisted data streaming method is proposed. It leverages static analysis and code transformations with an on-chip data streaming support as a viable alternative to prefetching mechanisms for regular code structures. Static analysis is used to identify and encode memory accesses with a dedicated representation. Then, a code transformation algorithm detaches data indexation and address calculation from computation, allowing for a significant code reduction. An on-chip data stream controller, attached to the L1 data cache, is used to autonomously generate memory accesses from the pattern representation and reorganize the data transfers in streams, with the aid of stream buffers. When compared with state-of-the-art prefetchers, the proposed solution provides up to 26% of code reduction, an IPC improvement of 2.4x, and an average performance improvement of 40%.

Index Terms—Compiler Static Analysis, Data Streaming, Regular Code Structures, Indirect Memory Accesses.

1 Introduction

The performance of Central Processing Units (CPUs) is often limited by the adverse impact of stalls due to long memory access latencies. Although caches promote an attenuation of such impact, the memory access latency is bound by the characteristics of the data access pattern and can only be fully mitigated by hiding it behind computation. From the vast set of approaches that have been considered, code optimization [1], [2], [3], [4], [5] and data (pre-)fetching [6], [7], [8], [9], [10], [11], [12], [13] have shown significant performance improvements.

At the hardware level, prefetching mechanisms monitor the cache miss stream, detecting the application memory access pattern [10], [9], [12], and prefetching data ahead of request. At the software level, compiler tools rely on static code analysis [1], [2], [3] to infer the memory access pattern and/or critical memory instructions [14]. This information allows the application of code transformations (e.g., data access reorganization [14], [5], code optimizations [4], and software prefetching [15], [16]) with the goal of hiding the memory access latency behind computation. The inferred access pattern can also be used at runtime for assisted execution [14] and assisted data prefetching [17], [13].

Existing prefetching methods are particularly successful when dealing with specific memory access issues, such as reduced data-locality [6], [7], [18], complex memory access patterns [19], [20], [21] or large datasets that do not fit in cache [17], [22]. In fact, prefetching technology has evolved to a point where the main concern is no longer the memory access pattern detection and prediction, but the timeliness and effectiveness of the procedure itself. This led to the emergence of new prefetchers [9], [10], [11], [23] that combine multiple hardware modules, with different data fetch granularities and prediction heuristics, across different cache levels. However, despite the improved throughput, resulting from a high accuracy and coverage of data access prediction, the added gains provided by each new generation of prefetchers are becoming limited.

To tackle such limitations, several works [24], [25], [26], [8], [21] exploit the fact that complex memory access patterns are most often generated by regular code structures that are compile-time detectable. These include common data indexation schemes based on affine loop transformations and on irregular data accesses, such as indirect memory accesses in the format A[B[i]]. Due to their deterministic representation, such patterns can be detected at compile-time and described by affine relations, and then dynamically resolved at runtime by on-chip data fetching modules (*data streaming*). As a result, it is possible to explicitly detach the addressing of the memory accesses from the processor to accelerate data acquisition and increase the processing throughput.

While some works have exploited such approaches to detect indirect memory accesses for stream prefetching [21], actual data streaming approaches [24], [25], [26], [8] have

The final version of this article has been published in IEEE Transactions on Computers (doi: 10.1109/TC.2020.2990302).

been limited to deterministic data access sequences (i.e., they lack support for irregular accesses). Moreover, they have only been deployed in application-specific accelerators [24], [25], [8] and usually require the programmer to hand-model the access pattern and to program it with custom application code.

This paper presents a new data streaming approach for compile-time detectable memory access patterns as an alternative to complex prefetching schemes for regular data access code structures. Hence, contrasting to state-of-the-art prefetchers, the proposed approach features: (1) a compilation tool that leverages static analysis to detect and explicitly describe memory access patterns (addressing the limitations of inaccurate memory access predictions); and (2) a dedicated controller to stream data (upon request) directly from the memory to the processor (addressing the performance saturation caused by prefetching timeliness). It operates as follows:

- At compile-time, the application memory access pattern is identified, described and encoded in a multi-level affine model, which also allows encoding data dependencies between accesses. The tool is able to infer deterministic access patterns and indirect memory accesses from regular code structures. As a result of this explicit representation of memory access patterns, their corresponding indexation and address calculation (by the CPU) becomes redundant. Accordingly, the tool performs a code transformation pass that replaces the subscript and indexation of each encoded load instruction with a stream reference (represented by a pointer). This reduces the number of instructions per loop iteration, accelerating the execution of the code.
- At runtime, a Data Stream Controller (DSC), collocated with the L1 data cache, generates memory accesses from the encoded representation and reorganizes data in streams. Hence, the DSC becomes responsible for fetching and buffering streams and for serving them (upon request) to the processor. In such circumstances, the L1 data cache is bypassed for data stream acquisition and access, avoiding cache pollution and early evictions resulting from poor data fetch timeliness.

The proposed compilation tool was integrated in the LLVM compiler and deployed as a Clang front-end tool. The DSC was implemented on the Gem5 simulator [27]. The resulting data streaming solution outperforms a typical stride prefetcher, attaining speedups as high as 1.9x. Furthermore, the implicit set of code transformations reduces the code size as much as 26%, leading to a combined performance improvement as high as 2.6x. As a result, the combination of the proposed techniques outperforms state-of-the-art prefetchers by 40% (on average).

2 Background and Motivation

Just as caches, the simplest and most common prefetching solutions exploit the spatial locality of memory accesses. Sequential prefetchers usually anticipate the loading of data in the cache lines based on the most recently accessed address ($y_{current}$). Being this a hardly efficient approach, stride-based prefetchers analyze individual accesses and calculate

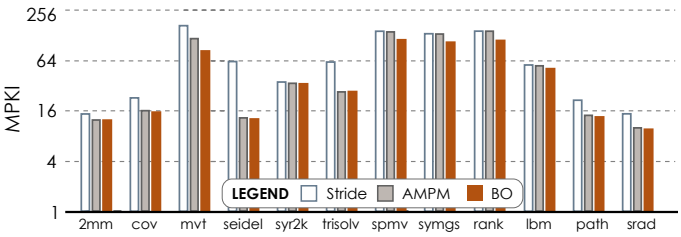

Fig. 1. L1 cache behaviour (in misses per kilo-instruction - MPKI) for regular code structures, for the state-of-the-art AMPM [9] and Best-Offset [10] prefetchers, compared to a traditional stride prefetcher

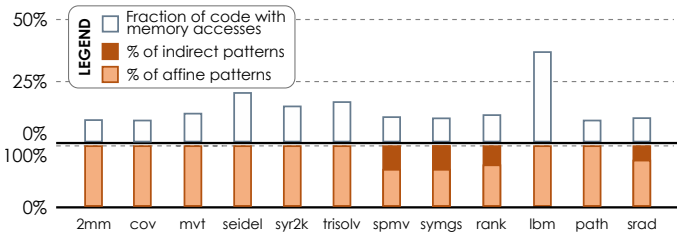

Fig. 2. Analysis of regular code structures, regarding (top) the fraction of code corresponding to memory accesses and (bottom) the characterization of compile-time detectable memory accesses.

the distance between consecutive addresses (*stride*), to predict the sequence of future accesses (up to a given *degree*):

$$y(degree) = y_{current} + degree \times stride \quad (1)$$

Spatial Prefetching: Some of these approaches also adopt intelligent prediction tables, indexed by the Program Counter (PC) or by the memory address, to infer more complex patterns and increase prefetching coverage. As an example, Shevgoor et al. [19] introduced the Variable Length Delta Prefetcher (VLDP), which maintains multiple prediction tables to store delta histories between subsequent cache line misses within physical pages. Ishii et al. [9] proposed the AMPM stride prefetcher to identify hot zones in memory. It uses a dynamically constructed bitmap to infer strided patterns in the access stream. Somogyi et al. [22] proposed the SMS prefetcher to identify code-correlated spatial access patterns and stream predicted blocks to the cache. Despite their attained coverage, these approaches assume that memory accesses maintain a regular behaviour over time, struggling to deal with highly complex and non-sequential patterns.

Correlation Prefetching: Modern prefetchers overcome such limitations by exploiting structural and temporal correlation heuristics, attaining high prediction accuracies and data coverage. For example, [12] proposes a context-based memory prefetcher that approximates spatial and temporal locality using reinforced learning. Recently, Bakhshalipour et al. [7] proposed the Bingo prefetcher, which associates observed spatial patterns to both short and long events to improve prefetching coverage and accuracy.

Yu et al. [21] proposed IMP, targeting indirect memory accesses. The detection mechanism correlates the values obtained by an index stream with subsequent cache misses to infer indirection. The IMP provided a breakthrough in this domain (which previously required the use of software prefetching approaches [15], [16]).

Offset Prefetching: Current prefetching methods have attained such a precision that the focus has shifted from the detection and prediction of indexing patterns to the timeliness of the prefetching procedure [11]. Michaud [10] proposed the BO prefetcher, implementing a selection mechanism that dynamically sets the prefetching offset depending on application behavior.

Despite their successful approach, the performance gain for each new generation of prefetchers has saturated (see Fig. 1), particularly on highly regular code structures (see Fig. 2), which dominate the majority of applications. In fact, the regularity and deterministic nature of most of these patterns can be detected by simple prefetching schemes.

Structural Representation and Data Streaming: By exploiting such deterministic nature, recent studies have shown that it is possible to obtain accurate representations of complex and deterministic access patterns to substantially accelerate data acquisition in application-specific domains. Hussain [26] proposed an Advanced Pattern-based Memory Controller (APMC) that supports up to 3D regular data-fetching mechanisms, such as scatter-gather and strided accesses with programmable tiling. The Hotstream framework [24] adopts a programmable approach that eases the description of regular data-patterns. A dynamic descriptor graph specification was proposed in [8] to encode arbitrarily complex (but deterministic) data-patterns. However, such strategies have been limited to regular data patterns and, since no compiler support is given, they require a manual encoding of the pattern.

Compiler-Assisted Methods: Notwithstanding, alternatives have been considered based on compiler-aided approaches. Guo et al. [13] utilize compiler information to reduce prefetch-related energy consumption by filtering prefetching requests with very small strides and prefetching only selected memory accesses identified by the compiler, and by applying different prefetching schemes depending on the predicted memory access patterns. Ebrahimi el al. [17] target pointer-based applications with compiler-guided prefetch filtering to inform the hardware about which pointer addresses to prefetch.

Several other compiler tools have also emerged in the scope of code static analysis. In particular, memory access tracing [28], [29], complexity [30] and polyhedral [1], [2] analysis or memory access profiling [3] have been used to analyze and optimize data indexation [4], [5].

Opportunity: To overcome the ever smaller gains associated to each new prefetcher generation for regular code structures, and surpass the currently observed performance plateau, alternative solutions are necessary. Hence, by observing that the memory access pattern in such applications is often known at compile-time (as it can be observed in Fig. 2 by the percentage of affine data patterns), there is an opportunity to explicitly extract and generate runtime data streams (even when these depend on the runtime value of one or more variables).

Proposed Solution: Based on this key idea, the proposed approach exploits data streaming schemes as a viable alternative to predictive prefetchers for data patterns generated by regular code structures. Instead of solely providing hints to prefetching hardware such as other compiler-assisted methods, it takes a step further by explicitly exposing and extracting the memory access pattern (directly from the source code) to provide support for data streaming. To do so, the proposed approach performs static analysis to extract the memory access pattern and to encode it with a dedicated representation, which is only fully decoded at runtime, when the value of depending variables is known. This mechanism is combined with a code transformation step that leverages the explicit detachment of memory accesses from computation, providing code reductions and accelerating the execution.

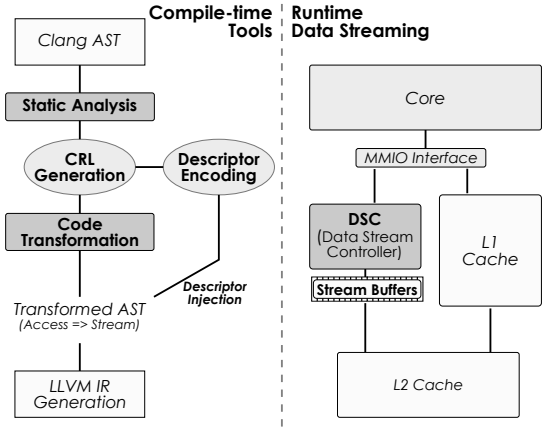

Fig. 3. Overview of the proposed compiler-assisted data streaming mechanism. It comprises a compile-time code analysis and transformation tool and an on-chip data streaming module.

3 Compiler-Assisted Streaming

The block diagram of the proposed compiler-assisted data streaming mechanism is depicted in Fig. 3. It comprises a compile-time tool (see Section 4) and a data-streaming module, colocated with the L1 data cache (see Section 4.5). The compile-time tool was integrated in the LLVM framework to perform static analysis over an annotated region of code. It starts by parsing the Clang Abstract Syntax Tree (AST) to identify the context of each data access in the region of interest (indicated in the annotation directive, as in Fig. 4.A). The context of each access is translated to a Context Representation Language (CRL) that gathers all the information regarding each access (see Fig. 4.C and Section 4.3). Such information includes all the dependencies for the address generation, including nested loop context (providing temporal information), indexing ranges and data dimensionality (for address calculation) and data-dependent access hierarchies (for indirect memory access representation).

A code transformation mechanism is also proposed to convert each extracted memory access into a data stream access (see Fig. 4.D and Section 4.5). This is possible because the memory addressing sequence is fully and exactly encoded, making the corresponding application code redundant. Such a transformation results in an explicit detachment of the memory access generation from the computational operations, and a consequent reduction in the number of instructions per loop.

The extracted data-pattern is encoded with a *Context Descriptor* (see Fig. 4.E and Section 4.4) and embedded in the application code. At runtime, the *Context Descriptor* is loaded into a dedicated DSC hardware module (see Fig. 3

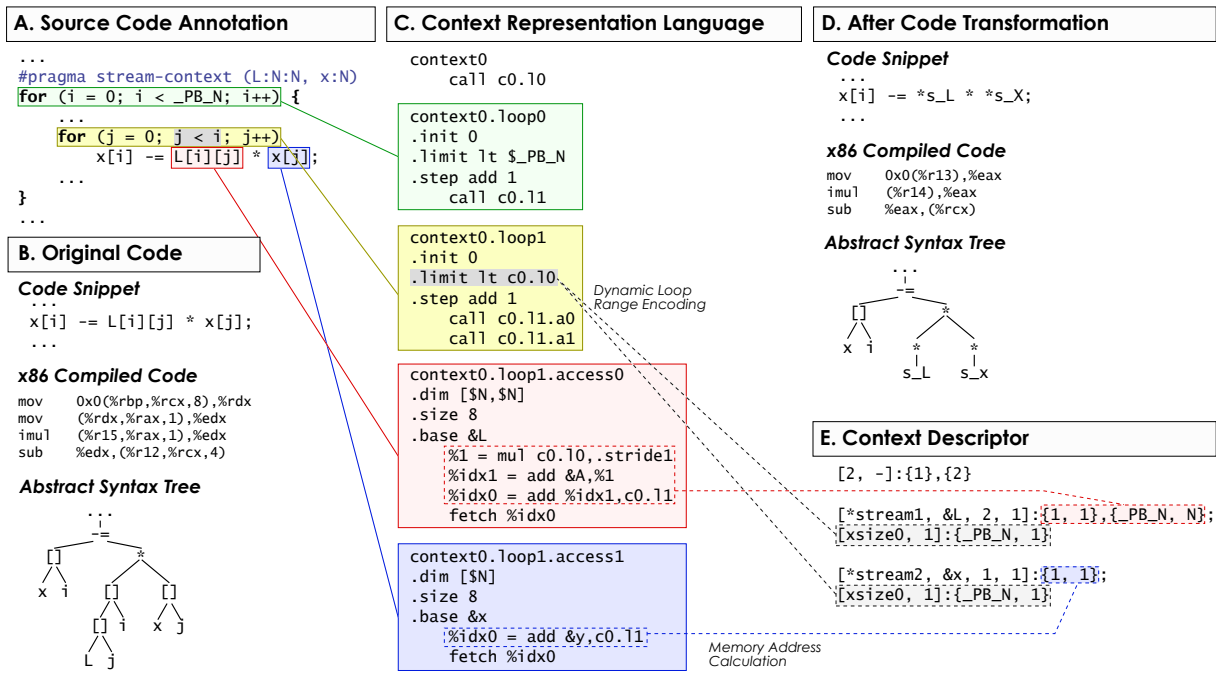

Fig. 4. Depiction of the several translations and code transformations performed by the proposed compilation tool, for a code snippet of the `trisolv` benchmark, from the Polybench [31] suite.

and Section 5), collocated with the L1 data cache. The DSC autonomously handles the stream generation and data fetching procedure. Each generated stream is stored in internal buffers and it is subsequently read by the core and/or used by the DSC to generate data-dependent streams.

Hence, the key advantage of the proposed scheme arises from not requiring any predictive schemes and access monitoring for stride detection. Instead, it relies on a static analysis performed at compile-time to encode memory access patterns in *Context Descriptors*, ensuring an exact coverage of the memory access sequence and avoiding redundant and over-prefetching scenarios that occur in predictive approaches. Moreover, its data stream buffering capabilities minimize premature cache evictions of prefetched data, resulting in an implicit increase of the timeliness of the acquired data. Hence, it results in a two-fold advantage over conventional prefetchers: (1) it accelerates the execution of code by reducing the total number of executed instructions; and (2) it minimizes latency in the memory access stream.

4 Access Pattern Detection

The proposed compile-time method for memory access pattern extraction and code transformation was built on the Clang LibTooling library [32]. This C++ interface provides full control over the compiler's front-end resources, as described in the following sections.

4.1 Compiler Module Overview

The compile-time method is divided in five steps, as depicted in Fig. 4: *i)* region extraction; *ii)* code translation to AST; *iii)* AST static analysis and translation to CRL; *iv)* AST to data streaming transformation; and *v) Context Descriptor* generation and code injection.

The region extraction phase was devised to be as simple as possible, requiring a programmer effort no higher than that of adopting other mainstream annotation schemes (such as the OpenMP library). It makes use of the pragma handling routines from the LibTooling library, therefore requiring any region-of-interest to be delimited by the directive (see Fig. 4.A):

```
#pragma stream-context (var:size[:size], ...)
```

It allows to configure and make the tool aware of the targeted array variables (and their size, per dimension) for data stream description. Moreover, such a scheme ensures that only relevant accesses are considered by the tool, avoiding the blind encoding of bad candidates for data streaming (e.g., an array that is often reused and is small enough to fit in the L1 cache is not a good candidate for streaming).

The application code is then transformed in a Clang translation unit, which is passed to the front-end tools to generate the corresponding AST (see Fig. 4.B bottom). At this point, the compilation flow slightly diverges from the typical compiler, which would proceed to generate the LLVM Intermediate Representation (IR) code.

Instead, the tool starts by generating its own internal memory access high-level representation, through a dedicated Context Representation Language (CRL), as detailed in Section 4.3 (see Fig. 4.C). This is achieved through a direct translation of the source code region-of-interest by parsing the Clang AST with a typical depth-first tree analysis. The translation mechanism is capable of inferring and representing deterministic and indirect array-based memory access sequences, by detecting affine relations in the regular access structures that result from typical for-loop coding schemes for array indexation.

After parsing the initial AST, a transformation pass modifies the AST subtree of each data access, to generate stream

accesses instead. This is done by creating an array with a stream reference per extracted access, and by transforming the n-dimensional array indexation (`idx`) with a stream access (see Fig. 4.D), i.e.:

```
<array_name>[idx0]..[idxN] → *stream_<name>
```

The address sequence of each memory access in the generated CRL is then translated to a low-level *Context Descriptor* and tagged with a corresponding stream address. This translation is performed by extracting the access pattern from the CRL and by encoding the necessary parameters to calculate the address sequence of each memory access (see Section 4.4 and Fig. 4.E).

Finally, the *Context Descriptor* is embedded in the original code by injecting a set of inline store instructions to send the descriptor data to the DSC, through a memory mapped interface. This allows sending information to the DSC that can only be obtained in runtime, such as the base address of the described array variables (see Section 5). From this point on, the control is passed to the LLVM compiler and the typical compilation flow is resumed.

4.2 Memory Access Modeling

The performed analysis is based on a formal mathematical model that captures the tagged deterministic memory access patterns. The key idea is to provide an n-dimensional (n-D) affine representation of the address sequence.

While existing predictive prefetchers make use of runtime-detected strides to calculate the sequence of addresses (with a unidimensional model based on near-temporal and near-spatial locality - see Eq. 1), the considered model aims at describing the exact sequence of addresses. It works by capturing nested loop-based indexation and loop- or data-dependent (indirect) index dynamic ranges, where current models fall short.

Accordingly, the following model to describe the address sequence for a given memory access is proposed:

$$y(X) = y_{base} + \sum_{k=0}^{dim_y} x_k \times stride_k \qquad (2)$$
$$\textbf{with} \quad x_k \in [\alpha_k,\ \beta_k],\ X = \{x_0, \ldots, x_{dim_y}\} \quad ,$$

where each stream access $y(X)$ is described as the sum of a base address value y_{base}, with dim_y pairs of increment variables (or indexes) x_k and $stride_k$ multiplication factors. Each increment variable x_k is represented by a integer range, with limits α_k and β_k (see Fig. 5).

While the model in Eq. 2 already represents a broad range of patterns, further complexity can still be achieved by combining multiple functions. Such combinations can be performed by determining the base address and/or the upper and lower bounds of each increment variable with another affine function (compare index ranges in Fig. 5).

4.3 Context Representation Language

The Context Representation Language (CRL) was specifically designed to gather and represent the information required to calculate the address sequence of a given memory access (or its **context**). The language is built on a basic instruction set and three container structures: *context*, *loop* and

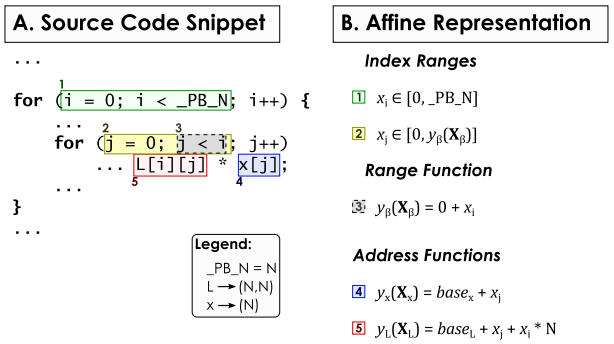

Fig. 5. Affine representation of the memory accesses in a code snippet of the `trisolv` benchmark, from the Polybench [31] suite.

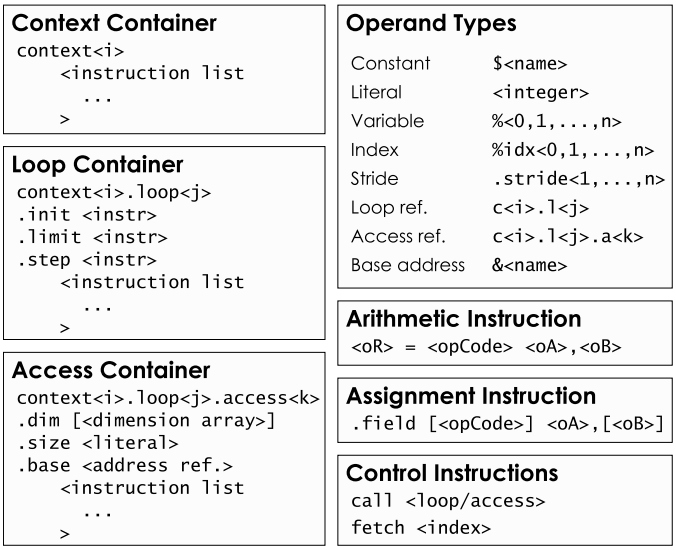

Fig. 6. CRL reference sheet.

access (see Fig. 6). Each container is composed of a unique identification, an header with configuration fields specific to each container type, and a body with a set of instructions. A CRL translation example is depicted in Fig. 4.C.

The *context container* serves as the entry point for the encoding. It encapsulates all the information concerning the indexing variables, base addresses, dimensionality and size of the array variables within a nested loop. This allows strides defined in different dimensions to be inferred and multiplied by indexing variables, in order to calculate the sequence of memory addresses based on the model in Eq. 2.

All memory accesses to a given array variable (in the same context) are encapsulated in an *access container*. Each access is represented by a `fetch` instruction preceded by a set of arithmetic instructions that calculate the address (see Fig. 4), based on (see Fig. 6): *i)* a dimension vector (`.dim`), to store the size of each dimension of the array; *ii)* the size of the array data type (`.type`); and *iii)* the array base address (`.base`), indicated by an address reference.

Each loop in the target region of code is encoded by a *loop container* (see Fig. 6). The container body encapsulates: `call` control instructions to the lower level loop and access containers; any necessary arithmetic instructions; and a container header that stores the range of the loop iteration variable (typically used as an indexing variable for array

structures). The latter includes: *i)* the variable initialization (`.init`); *ii)* the iteration limit value (`.limit`); and *iii)* the iteration step (`.step`). With this encoding, when the variable is used in a subsequent instruction (e.g., for address calculation), it is coded as a reference to the loop container (`c<i>.l<j>`).

The CRL provides a simple instruction set (see Fig. 6), with three instruction types (*arithmetic*, *assignment* and *control*) and up to three operands. Arithmetic instructions are encoded in the format `oR = opCode oA,oB`, where `opCode` specifies an integer arithmetic operation (`add`, `sub`, `mul`, etc.) over operands `oA` and `oB`, and destination `oR`.

Assignment instructions allow a composed configuration of container fields. They are encoded with one or two source operands and an optional operation code (representing an integer arithmetic operation or an inequality binary operator), in the format `.field [opCode] oA[,oB]`, where `.field` is the container field. This provides an encoding for the initialization, condition and step of each iteration variable, supporting the assignment of literals, variables, conditions and instructions, to each field.

The workflow between containers and memory access operations is encoded with control instructions, in the formats `call <ref>` and `fetch %idx<n>`, where `<ref>` is a reference to a loop container (`c<i>.l<j>`) or an access container (`c<i>.l<j>.a<k>`), and `%idx<n>` is an index.

4.4 Context Descriptor Specification

After being generated, the CRL is parsed and encoded in a low-level memory access *Context Descriptor*, which allows resolving the addresses calculation based on the variables defined in Eq. 2.

Fig. 7 presents the designed *Context Descriptor* specification, with support for multi-dimensional data patterns. It is composed of a top-level *Context Header*, which indicates the number (`acc`) and memory locations (a_id_{acc}) of a set of *Access Descriptors*, each describing one memory access pattern. It also contains a reference to a subsequent *Context Header* (see below), allowing multiple contexts to be described and solved in sequence (mirroring the original loop order).

The *Access Descriptor* defines a data access pattern by means of: *i)* an header tuple, containing a stream pointer (`stream`), the base address (`base`), the descriptor dimensionality (dim_k), and the number of modifier chains (`mod` - see below); and *ii)* pairs of $xsize_k$ and $stride_k$ fields, representing the x_k range (in number of iterations) and $stride_k$, respectively (see Eq. 2). These pairs have an implicit hierarchy (the rightmost pair has the higher position), where each pair is completely iterated (once) for each instance of the pair in the upper position of the hierarchy level.

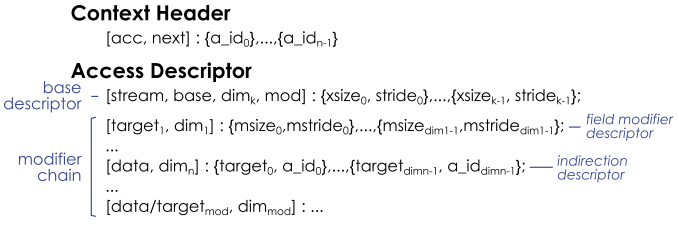

Fig. 7. Context Descriptor specification.

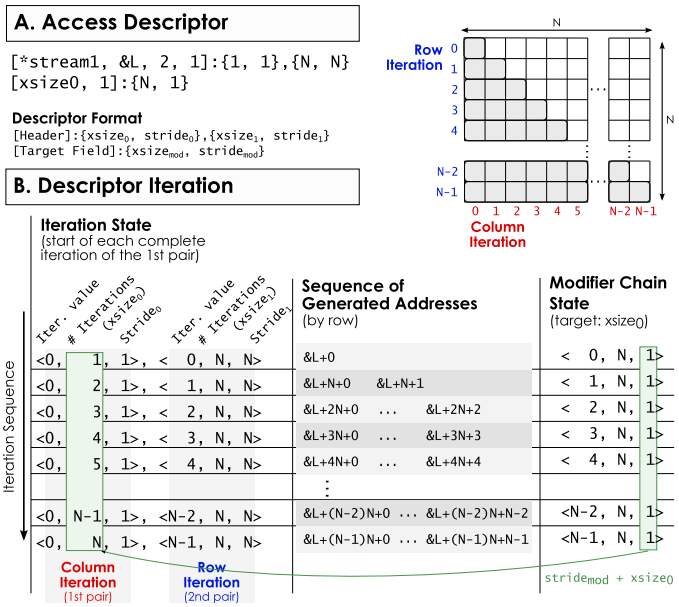

Fig. 8. *Access Descriptor* of a triangular matrix: the modifier chain increases the matrix column id after each full iteration of the first pair in the base descriptor, while the second pair adds a stride value to a subsequent matrix row.

Since each x_k range can be delimited by affine functions, the *Access Descriptor* provides an optional modifier chain, composed of multiple descriptors, in which the header indicates the target field ($target_{mod}$) to be modified (see Fig. 7). The modifier chain applies a field modifier descriptor to the target field every time the corresponding pair is fully iterated. As a result, the *Access Descriptor* is only fully iterated after each modifier descriptor completes its iteration.

Fig. 8 depicts the resolution of an *Access Descriptor* encoding of a N×N triangular matrix access pattern. The base descriptor is composed of two `{xsize, stride}` pairs, where the first (`{1,1}`) describes contiguous row accesses (initially with size 1) and the second (`{N,N}`) applies a stride to skip to the next row (N times). The descriptor relies on a modifier chain (with a field modifier - `[xsize0]:{N,1}`) that adds its own `stride` value to the `xsize` field of the first pair (in the base descriptor) each time it is completely iterated. As a result, each time the base descriptor iterates to a new row, the number of contiguous accesses is increased by 1, hence producing a triangular scan pattern.

The modifier chain can also represent data-dependent patterns (in the format of indirect memory accesses - e.g., A[B[i]]) by encoding data-indexation dependencies between *Access Descriptors*, through an indirection descriptor. It is composed of pairs of target fields ($target_n$) and descriptor identifications (a_id_n) (see Fig. 7). The indirection descriptor applies the encoded data dependencies similarly to a field modifier descriptor. However, it is the actual data from the stream generated by the a_id_n descriptor that is used to modify the target field in the *Access Descriptor*. An encoding example of indirect accesses is depicted in Fig. 9.

4.5 Automatic Code Transformation

By detaching memory addressing and data accesses from the original code, array indexing instructions become re-

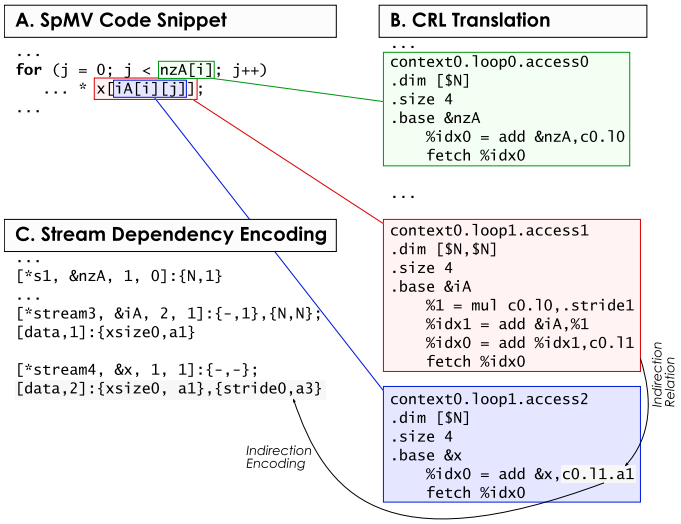

Fig. 9. Indirection representation and descriptor encoding of a code snippet of the `spmv` kernel, adapted from the HPCG [33] benchmark.

dundant. Hence, they can be eliminated, by transforming the array accesses into stream accesses.

The proposed code transformation pass is performed in two steps. Initially, it transforms each extracted data access from the former array subscript indexation into a stream access. Such a procedure relies on the fact that an n-dimensional array access is represented in Clang AST by a sub-tree, where the root node represents the array subscript operator (i.e. `[.]`) for the array's first dimension (see Fig. 4.B). Accordingly, to transform each extracted data access into a predefined data stream reference, it is only necessary to replace the subscript sub-tree by a pointer expression, as depicted in Fig. 4.D. This is done by tracing each data access captured by the CRL in the original AST and performing its *in-situ* transformation.

Finally, the *Context Descriptor* is embedded in the source code. To mitigate the overhead of loading it from the main memory, it is encoded by a set of inline store instructions that send the descriptor data to the DSC through a memory-mapped interface (see Section 5.1). This set of instructions is injected in the code after all the descriptor configuration parameters (e.g., array base addresses and sizes) are defined.

5 Runtime Data Streaming

To ensure the stream generation and data fetching, the DSC (see Fig. 10) is implemented as an hardware module that works in parallel with the L1 data cache.

To operate, the DSC starts by capturing the set of injected stores that send the *Context Descriptor* through a memory mapped interface, saving it in a local *Descriptor Memory*. Afterwards, the DSC initiates the generation of the addresses for each data stream, by fetching and buffering the corresponding data. Such data is immediately served to the core, by answering to requests to the stream references encoded in the *Context Descriptor*.

5.1 Stream Communication and Interface

From the core perspective, a data stream retrieval is simply represented by a read operation from a specific pointer. However, the stream is explicitly represented in the DSC by a structural address space (supported by buffering structures) that references data in a temporal sequence, detached from the physical address of each data block.

Accordingly, to provide the cores with an effective interface to transparently perform data stream accesses, the DSC offers a programmable interface that is memory-mapped to the core's memory access channel (see Fig. 10). When a request to a data stream reference is performed (corresponding to an *Access Descriptor*), it is redirected to the DSC (instead of the cache) and directly served with data from the corresponding stream.

5.2 Stream Address Generation

The iteration state of each *Access Descriptor* is kept in a dedicated *Descriptor Table* (see Fig. 10). This table is managed by a *Context Controller* that generates the addresses of a given context by solving the descriptor's *Context Headers*.

The memory addresses are calculated in a dedicated Address Generation Unit (AGU). It comprises two parallel functional blocks, each composed of an adder and a register set, responsible for iterating one $\{\texttt{xsize}_k, \texttt{stride}_k\}$ pair. The *stride control* block successively adds the $\texttt{stride}_k$ fields of the descriptor to its base address, while the *xsize control* block counts the iterations of each pair. When a base descriptor is fully iterated, the AGU applies its modifier chain (if available) and resets the descriptor with new field values.

The generated addresses are loaded into the *Stream Buffer* that was assigned to that stream. Once the corresponding data is fetched from memory, the generated address entries are replaced with the data, and later sent to the core (upon request). In the presence of data dependencies between streams (indirect memory accesses), the necessary data is read by the AGU, to iterate the dependent descriptor. The data dependence path is sent to the AGU (by the *Context Controller*), according to the *Access Descriptor* modifier chain.

5.3 Memory Access

In the CPU, each application data structure is usually allocated over a contiguous virtual memory address range. This contrasts with the operation of the DSC, which operates on the physical memory space. While typical prefetchers avoid this issue by stopping the address generation, waiting for the CPU to resynchronize the physical address offset, this is not possible with the introduced detachment of the address generation. Hence, the AGU is equipped with appropriate page crossing detection logic and, upon detection, it consults the CPU's Translation Lookaside Buffer (TLB) to obtain the page offset for the new address.

Moreover, whenever a new address is generated, it is passed to the *Request Filter* (see Fig. 10). This module maintains the last data block that was previously fetched for each stream (in a set of cacheline-sized registers) and serves the generated address with the corresponding data, by filling the matching entry in the *Stream Buffers*. Then, whenever a new address crosses the available data block address range, the *Request Filter* autonomously issues a new memory access for the new line. The issued requests are inserted in a *Request Queue* (also implemented by a buffer), and subsequently sent to the memory hierarchy.

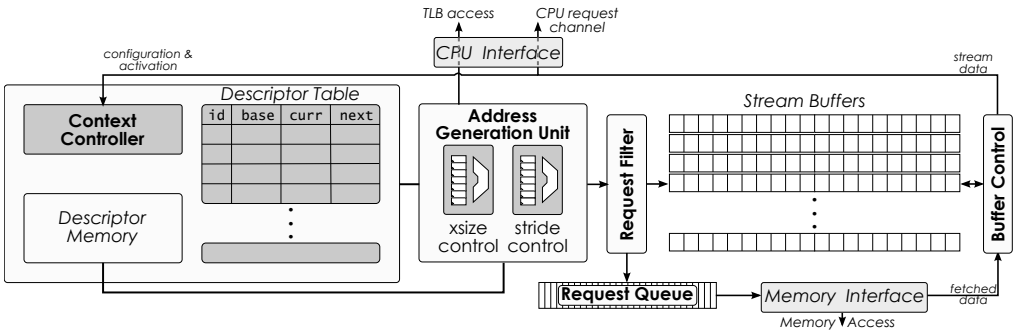

Fig. 10. Data Stream Controller architecture.

TABLE 1
Considered system configuration, comprising a memory chain similar to the one equipping the Intel Skylake microarchitecture [35].

CORE	**Frequency**	3 GHz
	Core Model	x86-64, Out-of-Order
CACHE CONFIGURATION	**Cache line Size**	64 bytes
	L1 I/D Cache	32 KB, 8-way, 4-cycle lat.
	L2 Cache	256 KB, 8-way, 20-cycle lat.
	L3 Cache	2 MB, 16-way, 36-cycle lat.
MAIN MEMORY	**Size**	4096 MB
	DRAM Model	Micron MT41J512M8
		11-11-11 DDR3-1600
		8 banks/rank, 2 ranks/MC, t_{RCD},t_{RP},t_{CL}=13.75 ns, t_{CK}=1.25 ns

TABLE 2
Considered evaluation benchmarks.

POLYHEDRAL LOOP COMPUTATION	`2mm`	Multiple Matrix Multiplications
	`cov`	Covariance Computation
	`mvt`	Matrix-Vector Product and Transpose
	`seidel`	2D Seidel Stencil
	`syr2k`	Symmetric Rank-2k Update
	`trisolv`	Dense Triangular Solver
SPARSE LINEAR ALGEBRA	`spmv`(*)	Sparse Matrix-Vector Multiplication
	`symgs`(*)	Symmetric Gauss-Seidel (Sparse Triangular Solvers)
	`rank`(*)	Graph-based Website Ranking
APPLICATION SPECIFIC	`lbm`	Computational Fluid Dynamics
	`path`	PathFinder - 2D Shortest Path
	`srad`(*)	SRAD Diffusion Method

(*) Benchmarks characterized by indirect memory accesses.

6 METHODOLOGY

The proposed static analysis and code transformations were implemented in LLVM 6.0 [34]. The DSC architecture was prototyped in the Gem5 simulator [27], by considering the setup presented in Table 1. It comprises a memory chain similar to the one equipping Skylake microarchitecture, based on information released by Intel [35].

6.1 Workloads

The implemented system was evaluated with a considerable selection of benchmarks, characterized by memory access patterns with regular code structures, from the C-Polybench [31], SPEC CPU 2017 and Rodinia [36] suites, and kernels from the HPCG [33] benchmark (see Table 2). This selection took into consideration the aim of obtaining a representative set of memory access patterns, kernels and real applications from a vast set of signal processing application domains, as described in the following paragraphs.

Polyhedral Loop Computation. Nested loop computations in the affine domain are all-around. Their deterministic nature is particularly suited for memory access pattern description and data streaming. A subset of kernels was selected from the C-Polybench [31] suite, comprising different combinations of pattern complexity, data reutilization and dataset dimensionality.

Sparse Linear Algebra. Sparse linear algebra algorithms usually represent data structures in Compressed Sparse Row (CSR) format, requiring operations between sparse and dense arrays to be implemented through indirection (i.e. `A[B[i]]`). Hence, the sparse matrix-vector multiplication kernel and the symmetric Gauss-Seidel method (two consecutive sparse triangular solvers) were adapted from the HPCG [33] benchmark. Since the CSR representation is also used in graph analytics, to store neighbouring vertices in sparse arrays, the PageRank [37] algorithm was also considered to evaluate this class of applications.

Application-Specific. A particular demanding subset of scientific applications were also used in this evaluation, namely: the SRAD diffusion method (ultrasonic and radar imaging), which exploits data access indirection; the PathFinder algorithm (search the shortest path of a 2D grid), which uses large data sets and high data reutilization; and the LBM algorithm (incompressible fluids simulation in 3D), characterized by a high dimensionality and computational intensity. The SRAD and PathFinder applications are part of the Rodinia [36] benchmark suite and the LBM algorithm was selected from SPEC CPU 2017. To support the proposed compilation tool, all benchmarks were modified by including the annotation scheme described in Section 4.1. Accordingly, the main computational functions and kernels of each benchmark were fully annotated for acceleration.

6.2 Reference Prefetching Setups

To validate the proposed data streaming mechanism, it was compared against four state-of-the-art runtime prefetchers (see Table 3 for the configuration parameters), namely:

Baseline (BASE): represents the most established prefetching scheme, i.e. a typical stride prefetcher, comprising a stride/confidence table indexed by the PC.

AMPM Prefetcher [9]: combines a stride prefetcher at the L1 cache (for fine-grained prefetching), with an L2 hardware module that uses a memory access map and pattern matching scheme to detect all possible strides in parallel.

Best-Offset (BO) Prefetcher [10]: relies on a stride prefetcher at the L1 cache, but introduces a different module at

TABLE 3
Reference prefetching and proposed setups.

BASELINE SETUP	L1 Stride Prefetcher; 16x4-entry PC table Confidence threshold: 4; Prefetch degree: 16
AMPM SETUP	L1 stride prefetcher (Baseline) L2 AMPM prefetcher [9] 256-entry access map; 5.2KB storage
BO SETUP	L1 stride prefetcher (Baseline) L2 Best-Offset prefetcher [10] 256-entry RR table; 4KB storage
IMP SETUP	L1 IMP prefetcher [21]; 16-entry PT table 4-entry IPD table; Max. prefetch degree: 16
DSC SETUP (PROPOSED)	16-entry Descriptor Table 16 32x8-Byte Stream Buffers 1KB Descriptor Memory

the L2 cache (a generalization of next-line prefetching), that dynamically sets the prefetching offset depending on the application behavior, while accounting for prefetch timeliness.

Indirect Memory Prefetcher (IMP) [21]: combines a stream prefetcher at the L1 cache with an indirect pattern detector (IPD) that computes coefficients between memory accesses to detect indirection between address pairs.

7 EVALUATION

The proposed compiler-assisted data streaming solution was evaluated by first studying the impact of code transformation, followed by a performance evaluation.

7.1 Pattern Description and Code Reduction

Fig. 11 depicts the observed code transformation and memory access encoding results. Reductions as high as 26% in the code size, and up to 28% in the number of commited instructions, are observed for the considered benchmarks. Such a reduction is bound by both the number of converted memory loads and by the complexity of their address calculation. For instance, only 40% of the loads in the `cov` benchmark were converted to streams; however, since the majority of loads represent matrix accesses, the removal of address calculation operations results in more than 13% of code size reduction. On the other hand, the `seidel` benchmark is characterized by a reduced percentage of address calculation instructions, resulting in only a 5% reduction.

Benchmarks with indirect memory accesses (`spmv`, `symgs`, `rank` and `srad`) take the most advantage of the code transformations. The conversion of indirect memory

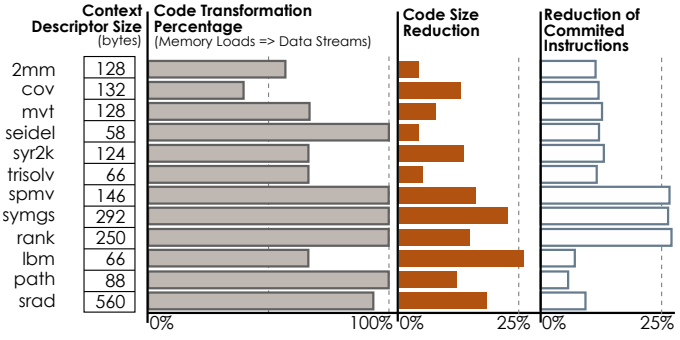

Fig. 11. Context descriptor size; percentage of streamed accesses and code reduction; and resulting impact in runtime-commited instructions.

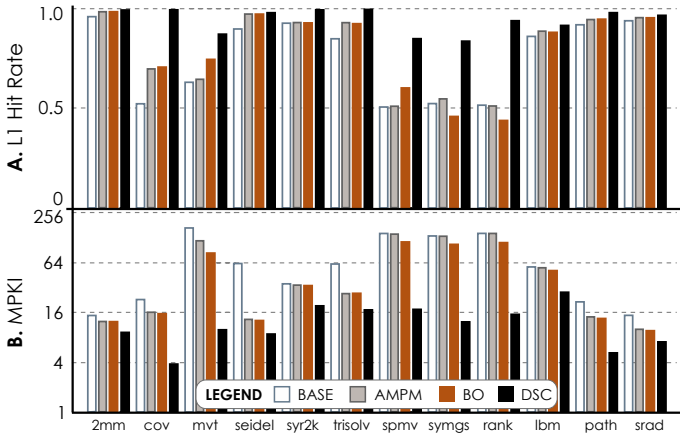

Fig. 12. L1 cache behaviour comparison in (A) hit-rate and (B) misses per kilo-instruction (MPKI).

accesses (in the `A[B[i]]` format) to single pointer references (`*stream`) eliminates both indexing and memory access instructions, resulting in a significant code reduction (up to 23%). Naturally, such a reduction imposes a larger descriptor size (see Fig. 11), due to the necessary inter-stream dependency encoding. For the `srad`, despite a conversion of 90% of loads to data streams, there is not a major reduction in the code size and commited instructions. This is mainly due to the computational complexity of `srad`, with a high percentage of code performing computing operations. Similarly, `lbm` is also characterized by a high computing intensity. However, it is still possible to attain a high reduction in code size (due to the elimination of indexation code for high dimensional accesses).

7.2 Memory Access Optimization

Fig. 12 shows the impact of each considred prefetching method in the L1 data cache behaviour (evaluated using the hit-rate and the misses-per-instruction (MPKI) metrics). In the evaluation of the proposed approach (`DSC`), both the data cache and stream buffer hit-rates are considered. For most polyhedral applications, the `BASE` stride prefetcher already provides high hit-rates, due to the regularity of the memory accesses. Nonetheless, the `AMPM` and `BO` are still able to improve the cache performance, due to an L2 high prefetching coverage. However, this is not the case when the access pattern complexity increases, as it can be observed in `cov` and `mvt`, where the datasets are large and require data reutilization; and in `spmv`, `symgs` and `rank`, where the memory accesses are irregular due to indirection.

The proposed `DSC` takes advantage over the other prefetching methods thanks to its memory access generation procedure. Since the data stream acquisition initiates before the execution of the corresponding requests and since data is not eliminated until it is read by the processor (contrarily to eventual evictions from cache caused by prefetched data in the other setups), data is promptly available ahead of time until it is needed. Moreover, the ability to exactly describe the sequence of addresses mitigates data locality issues. This is highlighted by the observed average hit rate (and corresponding MPKI improvement - see Fig. 12.B) of 95% with the `DSC`, when compared to the average 79% and 80% hit-rates observed in the `AMPM` and `BO` setups, respectively.

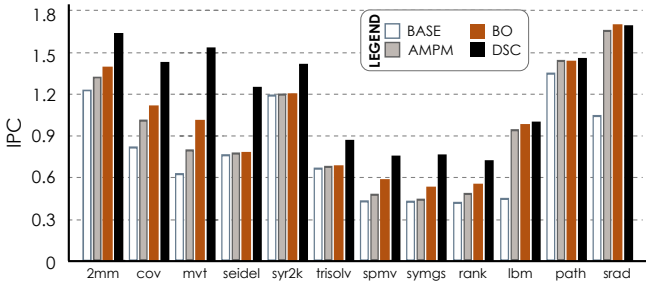

Fig. 13. Absolute IPC comparison.

7.3 Performance Evaluation

Fig. 13 shows the observed average instructions-per-cycle (IPC) metric for each setup. Despite the reduction of the number of executed instructions, the proposed scheme still allows for a significant IPC increase (up to 2.4x and 1.7x when compared to BASE and AMPM/BO, respectively).

The acceleration due to code reduction is particularly evident in the polyhedral benchmarks (see Fig. 14). Since the memory access patterns are easily detected by the BASE stride prefetcher, the improvements provided by AMPM and BO are limited to their ability to move data to the L2 cache in a more timely manner (as it occurs in *mvt* due to its poor data locality). However, due to the elimination of redundant array indexation (accounting for up to 40% of the achieved speedup - see Fig 14.B), the DSC is capable of further boosting the performance up to 2.63x over BASE.

The advantages of data streaming are also reflected in the overall system performance (see Fig. 14.A). This is especially evident when the application is characterized by large data sets, such as in 2mm and cov, which operate over multiple and large matrices. While the AMPM and BO prefetchers can easily detect the patterns and feed the L2 cache, the data set size inherently results in a large amount of L1 evictions. However, the DSC is capable of fetching and buffering data streams ahead of time, resulting in 1.5x and 2x speedups over the other setups, respectively in 2mm and cov.

Performance gains are also evident when reusing data structures larger than the L1 cache capacity (as in mvt, lbme and path, where a large dense matrix is read multiple times). It can also be observed the gains resulting from the coarse data movement into the L2 cache by AMPM and BO, making it available for reutilization. However, the data acquisition timeliness of the DSC still provides a performance boost of 10% for lbm and path, when compared to the other prefetchers. In the case of mvt, the matrix is also accessed in transposed order, resulting in L1 data-locality-related issues. However, the combination of the pattern description and code reduction provided by the proposed data streaming mechanism results in 1.9x and 1.5x speedups (in mvt) over the AMPM and BO setups, respectively.

The results obtained with spmv, symgs and rank show the capability of the proposed data streaming mechanism to deal with indirect memory accesses. While the BO correlation heuristics provide visible performance gains when compared to the base stride and AMPM prefetchers, it is still limited by the irregularity present in the data accesses. However, the DSC is capable of producing the exact sequence of addresses (after indirection) ahead of time and without polluting the L1 with unnecessary data. Such an advantage

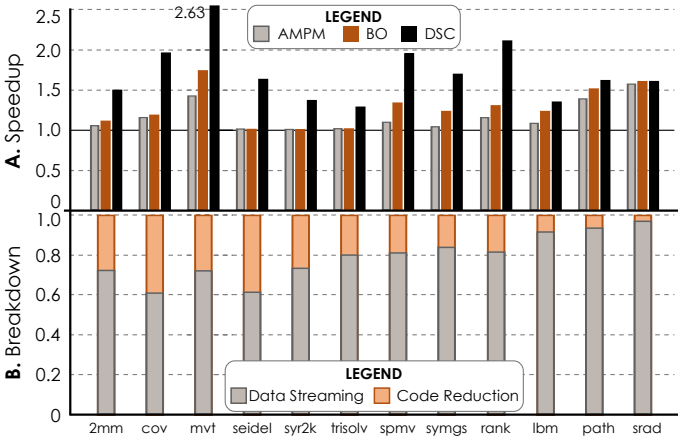

Fig. 14. Execution time speedup comparison (using BASE as reference) and breakdown of performance gains in streaming and code reduction.

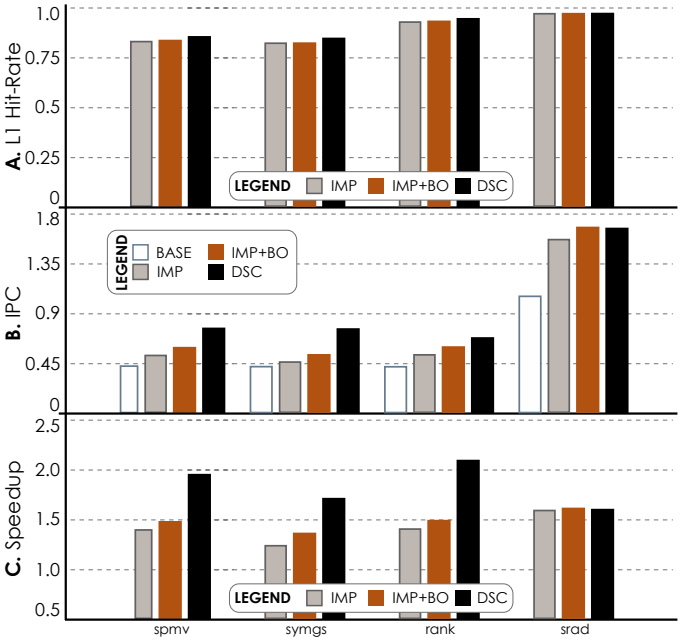

Fig. 15. Indirect memory access streaming comparison regarding (A) L1 cache behaviour, (B) absolute IPC and (C) execution speedup.

(accounting for about 80% of the achieved speedup - see Fig 14.B), when combined with the performed code reductions, results in 45%/37%/60% performance increases for spmv/symgs/rank, when compared to the BO setup.

7.4 Indirect Memory Access Streaming

Despite the ability of the BO prefetcher to deal with data access irregularity caused by indirection, the proposed DSC was also compared with the IMP [21] prefetcher, since it represents a state-of-the-art in memory access indirection (see Fig. 15). To achieve a fair comparison, only the spmv, symgs, rank and srad benchmarks were considered, since only these include indirect memory accesses.

According to the obtained results, the IMP and the DSC show similar performance when comparing the impact in the L1 data cache behaviour (see Fig. 15.A). This result is consistent with the fact that, upon detecting an indirection, the behaviour of IMP [21] becomes similar to that of the DSC when streaming an indirect pattern. As a result, when not

considering the impact of code reduction in the proposed data streaming, the throughput gains of both approaches is similar, when compared to the BASE setup. However, when considering the additional impact of the introduced code reduction (representing an average 17% performance increase for the four benchmarks), the DSC provides an overall IPC improvement over IMP of up to 1.7x, and a consequent average performance gain of 30% (see Fig. 15.C).

Finally, the performance gains of the DSC are still visible when compared with a setup combining the IMP (L1) and the BO (L2) prefetchers (IMP+BO). Despite a 5% performance improvement over IMP, resulting from an improved L2 cache behaviour, the proposed mechanism is still capable of achieving a 1.24x speedup (see Fig. 15.C).

7.5 Discussion

7.5.1 Performance Gains

The considered AMPM [9], BO [10] and IMP [21] setups show significant performance improvements when compared to BASE. This is a direct result of their greater prefetching coverage and accuracy and, in the particular case of BO [10], of the timeliness of the prefetching procedure. However, when comparing the proposed DSC with such highly effective approaches, it is observed that the DSC still provides an increased memory access throughput (resulting from the exact data acquisition and implicit timeliness of the data streaming), matching and improving their performance. This gain is stretched through the acceleration that is introduced by the code reduction, providing further 20% improvements (see Fig 14.B) over predictive prefetching.

Moreover, by comparing the obtained results, and considering the performance wall that is currently observed in the offered gains of the newest generations of prefetchers, it is possible to predict that for regular code structures the proposed method would still be able to match and improve the performance over other recent prefetching methods, such as the Bingo prefetcher [7]. In fact, even against a mechanism with ideal accuracy and timeliness, the code reduction of the proposed method would still allow improvements.

7.5.2 Resource Overhead

Despite the performance gains offered by the proposed approach, it requires simpler hardware structures when compared to the other reference prefetchers. While the amount of storage for data streams (4KB stream buffering) and pattern description (1KB descriptor memory) is similar to the storage required by AMPM (5.2KB) and BO (4KB), the required logic complexity is much smaller. In particular, the DSC only requires 2 adders for the AGU and compare logic to detect cacheline crossing in the address generation procedure. In contrast, the AMPM prefetcher [9] requires a significant amount of logic to match up to 256 stride patterns to find prefetch candidates on each access. On the other hand, the BO prefetcher [10] requires 3 adders to compute the position of a cacheline inside a page, while the recent request table is accessed through a hash function. When compared to IMP [21], the DSC requires a larger amount of resources (IMP requires less than 1KB of storage and lower logic complexity). However, its architecture was solely designed for detecting indirection in the memory access stream and is not suited for other types of data patterns.

7.5.3 L1 Cache Bypass Limitations

In the particular case of the srad benchmark, the proposed approach has a similar performance to all the other considered setups. This is a consequence of the used data set, which is small enough to fit in the L1 cache. Since the data accesses that are performed by the DSC are currently done directly to the L2, it increases the access latency. While this impact is mitigated by the provided code reduction and data stream pre-acquisition, there is still room for improvement. Future implementations of the DSC will consider a snoop-like access to the L1 cache tags (i.e. without causing demand misses) to directly copy data from the L1 and speedup the data access. Such an approach can further improve the gains, specially in the presence of high L1 data reutilization.

8 Conclusions

A new compiler-assisted data streaming mechanism for regular code structures was proposed in this paper as an alternative to current predictive prefetching mechanisms. It is based on the notion that particular application memory access patterns can be explicitly extracted at compilation time and used to generate the corresponding data streams for the processor. To attain this objective, it relies on a compilation tool that performs static analysis over an annotated region of code to identify, describe and encode the memory access patterns, supporting both affine complex patterns and indirect memory accesses. The encoded memory accesses are then automatically converted to data stream accesses, ultimately resulting in a reduced number of instructions and accelerating the execution of the code. At runtime, a dedicated DSC is used to generate data streams from the encoded representation.

The obtained results show that the implemented compilation tools achieve significant code reductions and consequent IPC improvements in the processor. As a consequence, the proposed data streaming method showed to improve performance by up to 2.6x, when compared to a typical stride prefetcher. Moreover, it showed to outperform state-of-the-art prefetchers by 40% (on average).

Acknowledgments

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects UIDB/50021/2020 and PTDC/EEI-HAC/30485/2017, and by funds from the European Union Horizon 2020 research and innovation programme under grant agreement No. 826647.

References

[1] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in llvm," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, 2011.

[2] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache, "Graphite: Polyhedral analyses and optimizations for gcc," in *Proceedings of the 2006 GCC Developers Summit*. Citeseer, 2006.

[3] Z. Majo and T. R. Gross, "Matching memory access patterns and data placement for numa systems," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 2012, pp. 230–241.

[4] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 521–532.

[5] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing indirect memory references with milk," in *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*. IEEE, 2016, pp. 299–312.

[6] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "An efficient temporal data prefetcher for l1 caches," *IEEE Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2017.

[7] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019.

[8] N. Neves, P. Tomás, and N. Roma, "Adaptive in-cache streaming for efficient data management," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. PP, no. 99, pp. 1–14, 2017.

[9] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for high performance data cache prefetch," *Journal of Instruction-Level Parallelism*, vol. 13, pp. 1–24, 2011.

[10] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 469–480.

[11] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 2014, pp. 626–637.

[12] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2015, pp. 285–297.

[13] Y. Guo, P. Narayanan, M. A. Bennaser, S. Chheda, and C. A. Moritz, "Energy-efficient hardware data prefetching," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 19, no. 2, pp. 250–263, 2011.

[14] K.-A. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Clairvoyance: Look-ahead compile-time scheduling," in *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*. IEEE, 2017, pp. 171–184.

[15] I. Hadade, T. M. Jones, F. Wang, and L. di Mare, "Software prefetching for unstructured mesh applications," in *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 2018, pp. 11–19.

[16] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Press, 2017, pp. 305–317.

[17] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 2009, pp. 7–17.

[18] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE, 2009, pp. 79–90.

[19] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 141–152.

[20] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *Software, IEE Proceedings-*, Feb 2004, pp. 96–96.

[21] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 178–190.

[22] S. Somogyi, T. Wenisch, M. Ferdman, and B. Falsafi, "Spatial memory streaming," *Journal of Instruction-Level Parallelism*, vol. 13, pp. 1–26, 2011.

[23] S. Kondguli and M. Huang, "Division of labor: A more effective approach to prefetching," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, June 2018, pp. 83–95.

[24] S. Paiágua, F. Pratas, P. Tomás, N. Roma, and R. Chaves, "Hotstream: Efficient data streaming of complex patterns to multiple accelerating kernels," in *2013 25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2013, pp. 17–24.

[25] Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong, "Spread: A streaming-based partially reconfigurable architecture and programming model," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 12, pp. 2179–2192, Dec 2013.

[26] T. Hussain, O. Palomar, O. Unsal, A. Cristal, E. Ayguadé, and M. Valero, "Advanced Pattern based Memory Controller for FPGA based HPC applications," in *2014 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2014, pp. 287–294.

[27] N. e. a. Binkert, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[28] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in GPUs," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 15–24.

[29] M. Amilkanthwar and S. Balachandran, "CUPL: A compile-time uncoalesced memory access pattern locator for CUDA," in *Proceedings of the 27th ACM International Conference On Supercomputing*. ACM, 2013, pp. 459–460.

[30] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU," in *ACM SIGPLAN Notices*, vol. 48, no. 8. ACM, 2013, pp. 57–68.

[31] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–10.

[32] "Clang. LibTooling," http://http://clang.llvm.org/docs/LibTooling.html, accessed: 30-July-2018.

[33] J. Dongarra, M. A. Heroux, and P. Luszczek, "Hpcg benchmark: a new metric for ranking high performance computing systems," *Knoxville, Tennessee*, 2015.

[34] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2004, p. 75.

[35] I. Corporation, "Intel 64 and ia-32 architectures optimization reference manual," Intel Corporation, Tech. Rep. 248966-040, April 2018.

[36] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.* IEEE, 2009, pp. 44–54.

[37] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

Nuno Neves received his Ph.D. degree (2019) in Electrical and Computer Engineering from Instituto Superior Técnico (IST), University of Lisbon (UL), Lisbon, Portugal. He is currently an Invited Assistant Professor at the Department of Electrical and Computer Engineering of IST and a Junior Researcher of the HPCAS Group of INESC-ID. His main research interests include high-performance and reconfigurable computing, domain-specific architectures, languages, and compilers. He is a member of the IEEE Circuits and Systems Society.

Pedro Tomás received the Ph.D. in Electrical and Computer Engineering (ECE) from Instituto Superior Técnico (IST), Technical University of Lisbon, Portugal, in 2009. He is an assistant professor in the Dept. of ECE, IST, and a senior researcher at Instituto de Engenharia de Sistemas e Computadores RD (INESC-ID). His research activities include computer microarchitectures, specialized computational structures, and high-performance computing. He is also interested in artificial intelligence models and algorithms. He is a member of the IEEE Computer Society and has contributed to more than 60 papers to international peer-reviewed journals and conferences.

Nuno Roma received the Ph.D. degree in electrical and computer engineering from Instituto Superior Técnico (IST), Technical University of Lisbon, Portugal, in 2008. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering of IST and an Integrated Researcher of INESC-ID, working on High-Performance Computing Architectures and Systems (HPCAS). His research interests include specialized computer architectures for digital signal processing, parallel processing, and high-performance computing. He has contributed to more than 100 papers to journals and international conferences. Dr. Roma is a Senior Member of both the IEEE Circuits and Systems Society and ACM.