# DPREF: Decoupling using Dead Code Elimination for Prefetching Irregular Memory Accesses

Konstantinos Sotiropoulos[*,1],
Jonas Skeppstedt[†,2],
Angelos Arelakis[‡,3], Per Stenström[*,1]

[*] *Chalmers University of Technology*
[†] *Lund University*
[‡] *ZeroPoint Technologies*

**ABSTRACT**

**Decoupling refers to the act of slicing a single instruction stream into separate memory accessing and computation streams, and arranging them in a dataflow manner. As these instruction streams gain a certain level of autonomy, decoupling can serve as a means to achieve accurate and timely prefetching.**

**Previous work on compile-time decoupling has yielded promising results for irregular applications. However, the presented slicing methods are limited in terms of their applicability and scalability.**

**This research aims to advance previous works on compile-time decoupling by proposing a flexible and systematic approach to slicing, based on an established optimization technique called Dead Code Elimination (DCE). The ultimate goal is to broaden the applicability to a wider range of irregular applications that existing methods struggle to address.**

KEYWORDS: irregular applications; decoupling; slicing; prefetching; DCE

## 1 Introduction

Irregular memory accesses pose a significant challenge in applications within graph analytics, database management systems, and sparse linear algebra. These applications experience high execution times due to frequent stalls caused by long latency DRAM accesses. Prefetching is a latency avoidance technique that aims to bring useful data closer to the processor before it is requested.

Processors are equipped with dedicated hardware prefetchers that predict future accesses and do so transparently, without requiring any software intervention. These prefetch-

[1]{konstantinos.sotirpoulos,per.stenstrom}@chalmers.se
[2]jonas.skeppstedt@cs.lth.se
[3]angelos.arelakis@zptcorp.com

ers are efficient for applications with regular access patterns, such as iterating over a dense matrix. Nonetheless, they struggle with indirect memory accesses, like those encountered when traversing a sparse graph.

There have been several proposals to improve the predictive capabilities of hardware prefetchers e.g. with the application of machine learning [BKN+21]. However, due to their speculative nature, these techniques can not be accurate, fetching irrelevant alongside useful data, and thus wasting bandwidth, on-chip space and energy.

To enhance accuracy, another class of proposals aims to add some programmability to hardware prefetchers [TMB+21]. Still, this class of prefetchers has limited applicability, as their constrained programming model is only able to cover certain data structures and representations. Moreover, programming these prefetchers necessitates the use of a different programming language from the one used to express the application, creating a learning barrier that hinders the wider adoption of such proposals.

In addition to hardware prefetchers, processor ISAs include specialized instructions to explicitly request data prefetches. Programmers and compilers can leverage their knowledge of the underlying microarchitecture and an application's access patterns to utilize these instructions. This technique, known as software prefetching, has a fundamental limitation: the timing of prefetches, as expressed statically by a look-ahead distance or the location of a prefetch instruction, is hard to reason about. This limitation becomes particularly evident with irregular applications, where software prefetching techniques provide modest performance gains [AJ18].

A recent wave of proposals aims to provide timely and accurate prefetches for irregular applications that use sparse data structures, demonstrating very promising results [MST+21, NS23]. Due to the way sparse data structures are represented, their traversal exhibits a specific type of irregularity, indirection, typically taking the form `A[B[C[i]]]`. The key idea behind these proposals is to decouple such multi-level indirection into distinct pipeline stages, each housing a slice of the application.

These proposals represent examples of hardware-software co-design: the decoupling is enacted by the compiler, and they offer ISA-visible queues for stage-to-stage communication. These queues benefit from low-latency hardware implementation as inter-stage communication occurs every few cycles.

Despite their promising results, these existing proposals face limitations due to their slicing methods. Either they can produce only a restricted number of slices, as in [MST+21], or they can only slice applications that can be arranged as feed-forward pipelines, as in [NS23].

The primary purpose of this study is to extend these proposals through a flexible and systematic compiler slicing technique, utilizing an established optimization technique called Dead Code Elimination (DCE) [CFR+91]. Paired with an appropriate processor architecture, our approach can handle a broader range of applications while scaling with Memory Level Parallelism (MLP) and memory bandwidth.

## 2 DCE for Slicing

With DCE, the compiler partitions a program into live and dead code, and then discards the dead portion. As will be demonstrated, DCE can evolve into a flexible and systematic slicing technique by redefining what is considered as live code.

In standard DCE, live/essential code is formulated on the basis of pre-live statements:

statements that have an effect on observable state, e.g. I/O operations or explicit memory manipulation. Pre-live statements along with their data- and control- dependencies are what constitutes live code. Take, for instance, figure 1: within the function body, the only pre-live statement is located at line 6. This statement is data-dependent on line 2 and control-dependent on line 3.

```
1  void func(int *A, int B) {
2    int C = B % 5;
3    if (C == 0)
4      B = 0;
5    else
6      *A = C;
7  }
```

Figure 1: Live code is highlighted; the remaining code is considered dead/ineffectual and can be discarded by an optimizing compiler

Drawing from the concept of program slicing as introduced by Weiser [Wei84], live code can be viewed as a slice: a reduced version of the pre-optimized program, guaranteed to produce the intended behavior with respect to a slicing criterion, the pre-live statements. To perform any arbitrary slicing with DCE, one simply needs to indicate pre-live statements. This is central to the programmer/compiler interface design, and can be facilitated through the use of annotations, as will be discussed in the upcoming section.

# 3 DPREF on Sparse Matrix-Vector Multiplication

Figure 2 demonstrates the product y = Ax of a sparse matrix A and a dense vector x (SpMV) where A is stored in Compressed Sparse Row (CSR) format. Array `Values` stores all non-zero values of the matrix and array `ColIdx` stores the column index of each non-zero value. The `Offsets` array denotes the start index of each row in the `Values` and `ColIdx` arrays. For instance, the non-zero values of row `n` in the `Values` arrays start at index `Offsets[n]` and end just before index `Offsets[n+1]`.

The `pragma` directive in line (1) explicitly denotes this relationship among the arrays. The compiler leverages this annotation to mark as pre-live the instructions that use `Values` and `X`, while deliberately excluding their indices from the dependency analysis. This marks the beginning of the first slicing phase.

Figure 3a is the slice that corresponds to the dead code. In this slice, a queue is used to push non-zero values of the matrix and their corresponding column indices. The queue is not just a conduit for data transfer, but also a carrier of control flow information. For instance, when processing of a row is completed, a `LoopTerminationToken` is pushed into the queue. Similarly, when the entire matrix has been processed, a `TerminationToken` is pushed, indicating completion of the task.

Figure 3b is the slice that corresponds to the live code. The column loop `J` has been substituted with an infinite loop, where tokens are popped from the queue. These tokens are initially checked for control flow information before being utilized for actual computation. Though not depicted here, this slice can be further sliced on the access to `X[ColIdx]`, reducing its scope to solely performing the computation and storing of the result

```
#pragma dpref indirection(Values[Offsets], X[ColIdx[Offsets]])
void SpMV(const double * __restrict Values, const size_t * __restrict Offsets,
          const size_t * __restrict ColIdx, const double * __restrict X,
          double * __restrict Y) {
  for (size_t I = 0; I < N; I++) {
    Y[I] = 0;
    for (size_t J = Offsets[I]; J < Offsets[I + 1]; J++)
      Y[I] += Values[J] * X[ColIdx[J]];
  }
}
```

Figure 2: SpMV, before slicing. Shaded in green is exclusively live code, shaded in red is exclusively dead code, and code in yellow is common to both.

# 4 Preliminary Conclusion

Our work is currently in progress, exploring the viability of our proposed approach through manual application slicing. Our study extends beyond the SpMV algorithm, including applications like breadth-first search, PageRank, and Hash Join. Three primary challenges have emerged from our findings thus far.

First, for algorithms that are not as straightforwardly decouplable as SpMV, and which encompass feedback loops, it's essential for the compiler to discern and reason about critical system properties, such as deadlock. Modeling the system as a Petri net within the compiler might offer a viable solution to this issue.

Second, when a feedback loop exists between the first and final slice, a distributed termination solution is required. Specifically, the first slice needs to determine when all queues are empty and all slices are inactive so that the termination token can be injected [DFvG83].

Lastly, the microarchitecture of the processor, apart from facilitating quick push/pop operations, must also handle the overhead associated with checking for control flow tokens.

# 5 Acknowledgment

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

# References

[AJ18] Sam Ainsworth and Timothy M. Jones. Software Prefetching for Indirect Memory Accesses: A Microarchitectural Perspective. *ACM Transactions on Computer Systems*, 36(3):1–34, August 2018.

[BKN+21] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1121–1137, Virtual Event Greece, October 2021. ACM.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the con-

```
void firstSlice(const size_t *__restrict Offsets, const size_t *__restrict ColIdx,
                const double *__restrict Values,
                dpref::Queue<tuple<double, size_t>> *Q) {
  for (size_t I = 0; I < N; I++) {
    for (size_t J = Offsets[I]; J < Offsets[I + 1]; J++) {
      Q->push(make_tuple(Values[J], ColIdx[J]));
    }
    Q->pushLoopTerminationToken();
  }
  Q->pushTerminationToken();
}
```

(a) The first slice corresponding to the dead code

```
void secondSlice(const double *__restrict X, double *__restrict Y,
                 dpref::Queue<tuple<double, size_t>> *Q) {
  for (size_t I = 0; I < N; I++) {
    Y[I] = 0;
    for (;;) {
      auto Token = Q->pop();
      if (Token.isTerminationToken()) {
        return;
      }
      if (Token.isLoopTerminationToken()) {
        break;
      }
      auto [MVal, ColIdx] = Token.getData();
      Y[I] += MVal * X[ColIdx];
    }
  }
}
```

(b) The second slice corresponding to the live code

trol dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[DFvG83] Edsger W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, June 1983.

[MST+21] Aninda Manocha, Tyler Sorensen, Esin Tureci, Opeoluwa Matthews, Juan L. Aragón, and Margaret Martonosi. GraphAttack. *ACM Transactions on Architecture and Code Optimization*, 18(4):1–26, September 2021.

[NS23] Quan M. Nguyen and Daniel Sanchez. Phloem: Automatic acceleration of irregular applications with fine-grain pipeline parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, February 2023.

[TMB+21] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor Mudge, and Ronald Dreslinski. Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 654–667, Seoul, Korea (South), February 2021. IEEE.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.