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.

This research aims to advance previous works on compile-time decoupling by proposing a systematic approach to slicing. 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.

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

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

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.

The primary purpose of this study is to extend these proposals through a more systematic compiler slicing technique, utilizing an established optimization technique called Dead Code Elimination (DCE) [CFR$^+$91]. This approach aims to handle a wider range of applications, offering more flexibility in the decoupling process.

2 DCE for Slicing

With DCE, the compiler partitions an application into redundant and essential code, and then discards the redundant portion. The algorithm maintains a workset of live instructions. The workset is initialized with a set of pre-live instructions, such as memory stores, that explicitly modify system state. Live instructions are iteratively removed from the workset for control- and data-dependency examination, subsequently adding more instructions based on outcome of this examination.

With an intermediate representation of a program in Static Single-Assingment (SSA) form, identifying data dependencies is straightforward due to the unambiguous definition of each variable. Control dependencies, on the other hand, necessitate a post-dominance analysis on the CFG. A live instruction is deemed control-dependent on a conditional statement if the basic block containing the live instruction is part of the post-dominance frontier of the basic block that houses the conditional. In other words, if the live instruction is not always on the path from a conditional to the program's exit, then the conditional should be considered live.

DCE can evolve into a flexible slicing technique through redefining what is considered as a pre-live instruction. The redundant portion is not discarded; instead, it undergoes a dependency examination, similar to pre-live code, and subsequently forms one of the two slices, the other being the pre-live instructions and their dependencies. This slicing process can then be systematically and recursively applied.

The identification of pre-live instructions is central to the programmer/compiler interface design, potentially facilitated by annotations, as discussed in the upcoming section, and proves critical in kickstarting the slicing process.

3 DPREF on SpMV

Figure 1 demonstrates the 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 2a is the slice that corresponds to the dead code. Notice how the queue is used to convey tokens that can represent either data or control flow information. Figure 2b is the slice that corresponds to the live code. Notice how the token popped from the queue is first examined for control flow information.

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

```
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

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 Acknowledgement

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

References

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control 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. Aragon, and Margaret Martonosi. GraphAttack: Optimizing Data Supply for Graph Applications on In-Order Multicore Architectures. *ACM Transactions on Architecture and Code Optimization*, 18(4):1–26, December 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.