Compiling Programs to Utilize Cache Optimizations

Samuel Grayson
grayson5@illinois.edu
University of Illinois at Urbana-Champaign

Nirupam K N
nirupam2@illinois.edu
University of Illinois at Urbana-Champaign

ABSTRACT

In most state-of-the-art systems, the cache coherence pattern has been fixed in architecture, which does not know about the general structure of the program it is executing. However, Spandex changes supports the possibility of dynamically selected cache coherence optimizations. We exploit this opportunity by writing a compiler-pass that analyzes the memory-access pattern in parallel tasks and selects a specialized cache coherence pattern.

CCS CONCEPTS

• **computer systems organization** → **Architectures**; • **Software and its engineering** → **Compilers**.

KEYWORDS

cache coherence, heterogeneous systems, compilers

1 INTRODUCTION

Many multi-processor systems use the MESI protocol for cache coherence [13], which provides sequential consistency for any data race-free program. However, the cache coherence protocol might make redundant or indirect requests because it does not know about the program structure.

For example, suppose a program directs each processor to compute one layer of a neural net in a pipelined fashion (see fig. 1). Processor i needs to send its activation's to Processor $i+1$ so that Processor $i+1$ can compute the next layer of activations. In traditional MESI system, Processor i would store data to its cache, in 'exclusive' state. Then Processor $i+1$ would attempt to load the data, miss in its cache, go to the bus, ask Processor i for data, downgrading i's copy to 'shared' state.

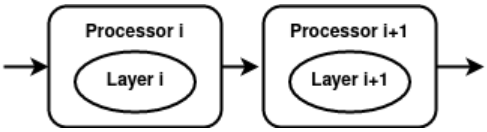

Figure 1: The dataflow graph of pipelined neural net inference

An optimization is available: The compiler could prove that Processor $i+1$ is the consumer of the activations written by Processor i, so it could instruct Processor i to send its data directly to the cache of Processor $i+1$, bypassing the Last-Level Cache (LLC) [5]. Streaming programs spend a substantial portion of their time and energy in the Network on Chip (NoC), so optimizations like this save power and time [2].

Many of these coherence optimizations are only beneficial in specific cases, so the architecture needs to dynamically select the best coherence optimization for the given memory-locations based on the access pattern while maintaining correctness. The architecture only knows about historical access patterns and does not know what the other processors in the system are doing, so it is difficult for the architecture to select a coherence optimization by itself. The compiler can form an expectation of future access patterns and could know what other processors in the system are doing, so it should inform the architecture what coherence optimization to use for specific memory-locations.

Our contribution is a compiler-pass which analyzes the access pattern of the given program to identify certain opportunities for coherence optimization. We hope future work will build on this pass to support more kinds of coherence optimizations and develop a better cost-model. Our implementation leverages the HPVM compiler Intermediate Representation (IR) [9] and emits coherence optimizations for the Spandex Cache Coherence protocol [1].

2 RELATED WORK

Remote Store Programming [8] exploits a similar data-movement optimization, but it requires manual programmer input. The goal of this work is to automate that analysis. While future versions of this compiler could target RSP, targeting Spandex gives us a superset of the optimizations available in RSP.

Prior work on DeNovo cache coherence protocol does include a compiler. Still, it assumes programs are written in a language with annotated memory-effect regions (for each function, a set of memory locations which it could read and another set which it could write) [5]. No popular programming language includes manual annotations; manual annotations are too tedious and error-prone to burden computer programmers. Despite the plethora of work [3, 7, 11] on memory-effect inference, no popular language toolchain includes such an inference pass; they are expensive to perform on an imperative language with polymorphism. Work on dynamic cache-coherence in Spandex [2] presents a high-level algorithm, but it is too high-level for a compiler-developer to implement directly.

3 IMPLEMENTATION

3.1 Coherence Optimizations

Most coherence protocol do not support dynamically selecting coherence optimizations (as discussed in section 2). Spandex [1] is one which does, and dynamic coherence optimizations have already been suggested in prior work [2].

In Spandex, there are four states: *Invalid*, *Valid* (read-only copy expecting self-invalidation), *Shared* (read-only copy expecting writer-invalidation), and *Owned* (writable with no invalidation messages).

To limit the scope of coherence optimizations, we will consider the special case of producer/consumer relationships, where the producer issues and completes all stores before the consumer issues and completes all loads every epoch, and this pattern repeats. This

is a widespread paradigm in pipelined parallelism, such as recurrent neural net training and inference.

MESI: In MESI, initially all lines would be invalid-state. The producer's first store to a line issues an invalidation on the bus and moves the line into modified-state. Subsequent stores hit. The consumer's first load to a line requests this line from the producer, moving both copies into shared-state.

Spandex: Without optimization, the Spandex coherence protocol emulates this, except the L2 is directory-based not bus-based, so the invalidation request looks up any processors which hold the line.

Spandex Producer-Owned: One possible optimization is to have the producer always own the buffer, so all of its stores hit locally. The consumer would have to look the line up in the producer's cache, but the owner can be predicted in hardware (suggested in [2]), so this request goes from L1-to-L1 (bypassing the L2). The compiler would insert a self-invalidation request in the consumer's code at the end of every epoch, to maintain memory consistency. This is optimal when the loads are sparse, because traffic is only generated for each load.

Spandex Consumer-Owned: Another possible optimization is to have the consumer always own the buffer, so all of its loads hit locally, as suggested in [2]. The producer would writethrough with owner prediction, allowing it to send data directly to the consumer's cache. This is optimal when the write are sparse, because traffic is only generated for each write and there is little memory parallelism. No computation of the producer is logically dependent on store, but much of the consumer's computation may be logically dependent on the result of a load, so increasing the latency of stores in exchange for decreasing the latency of loads can be desirable.

3.2 Identifying Producer/Consumer Relationships

In order to identify the memory access pattern among multiple parallel tasks, we need an Intermediate Representation (IR) that takes task-level parallelism into account. Traditional compiler IRs like LLVM [10] and GIMPLE [12] do not support this, for example. HPVM is an extension of LLVM that represents task- and data-parallelism [9]. Currently, HPVM programs are written in C with library calls, which is a cumbersome interace, but there are plans to add a frontend to higher-level languages such as Keras.

In HPVM, programs are represented as a hierarchical dataflow graph (DFG). Every node is either called a 'leaf' or an 'internal' node. For example in fig. 2 C, E, and F are leaves, while A, B and D are internal.

Leaf nodes can contain computation while internal nodes can only string together other nodes. HPVM can execute one input at a time, in which case the parallelism is the width of the DFG, but HPVM also supports 'streaming' (pipelined) execution, where time can be divided into epochs where every node runs once each epoch and the parallelism is the number of nodes in the DFG. In each epoch, every node runs on its own data-item, passing data to its successors to compute on in the next epoch. The data passed through pointers between nodes every epoch is exactly the kind of repeated producer/consumer relationship that we want to optimize.

We want to detect any loads and stores in different nodes that refer to the same memory location. Therefore, we trace each pointer from its definition to its use. Initially, our pass will assign all producer/consumer memory accesses to a given coherence optimization. In the future, our compiler-pass would examine the memory access-pattern on these pointers.

3.3 Access-Pattern Analysis

HPVM gives us a hierarchical DFG graph (see fig. 2). The first step is to flatten the graph, so we see how data is passed between the leaves (see fig. 3). This is done by algorithm 1.

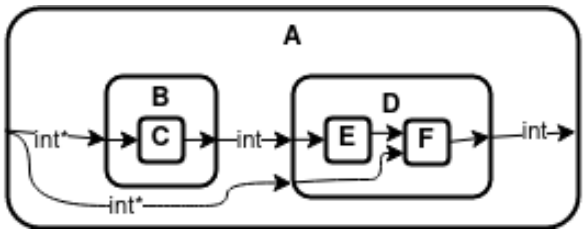

Figure 2: Hierarchical DFG

Algorithm 1: Flattening the HPVM DFG

```
input : a hierarchical DFG, hdfg
output: a flat DFG, fdfg
for source in hdfg nodes do
    if source is leaf then
        source′ := source
    else
        source′ := source.entry, where source.entry is a
          dummy node bearing that name. It will be removed
          later.;
    for destination, where source → destination in hdfg do
        if destination is leaf then
            destination′ := destination;
        else
            destination′ := destination.entry;
        Insert source′ → destination′ into fdfg;
```

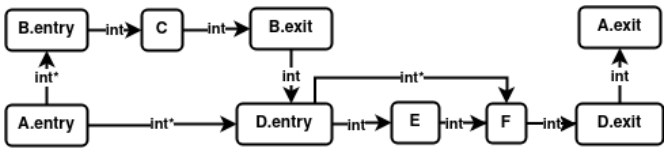

Figure 3: Flattened DFG

Next, we need to expose the leaf-connectivity (see fig. 4), because the leaves are the nodes which contain all computation, including loads and stores. This is done by algorithm 2.

Algorithm 2: Pruning non-leaves from the flat DFG

```
input  : a flat DFG, fdfg
output : a leaf DFG, ldfg
Initialize ldfg to a copy of fdfg;
for node ∈ fdfg do
    if node is not a leaf and not the root entry/exit then
        for predecessor → node, node → successor ∈ ldfg
        do
            Remove predecessor → node from ldfg;
            Remove node → successor from ldfg;
            Insert predecessor → successor into ldfg;
```

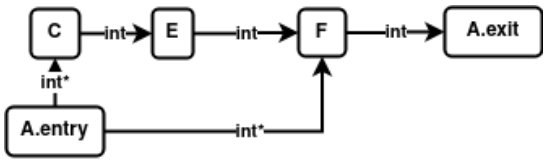

Figure 4: Leaf Dataflow Graph

Algorithm 3: Pruning non-leaves from the flat DFG

```
input  : a leaf DFG, ldfg
output : a memory-communication DFG, mcdfg
Initialize ldfg to a copy of mcdfg;
for node ∈ ldfg do
    if node emits a pointer type then
        for node → A, node → B ∈ ldfg do
            if Pointer is writable in A and A precedes B then
                Insert A → B into mcdfg;
```

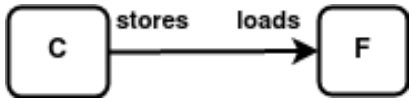

Figure 5: Memory Communication Graph

Finally, we need to look through the leaf DFG and determine where there is implicit memory communication (see the memory communication graph in fig. 5). This is done by algorithm 3.

Finally, we have analyzed the memory access-patterns in the program and identified consumer/producer relationships for cache optimization. Future work would do even more analysis on these relationships.

4 METHODS

Unfortunately, there is only one existing simulator or Spandex protocol, and it is not possible to set up in such a short period of time. There are students working on a gem5 simulator, but that effort was not ready when we were like we had hoped. As such, the experiments in this report will depend on analytical models.

We have annotated each benchmark so that it logs all loads and stores to the buffers I am considering a part of the producer/consumer relationship. Then, we wrote a program that simulates the cache states for those addresses. These are the assumptions our simulator makes:

- If the coherence optimization supports coalescing and a store is to the same line as one of the previous N stores (where N is a parameter defined below), it gets coalesced.
- The program is in steady-state (no cold-effects). We *do* simulate cold-effects which would recur at the beginning of every epoch.
- The working-set fits in the L1 cache. No conflict or capacity misses.

The parameters we used are detailed in table 1. The latency numbers come from a prior publication on dynamic cache coherence [2] and the flit sizes come from the RTL implementation of Spandex, which in turn is dictated by ESP [4].

Table 1: Simulation parameters

Parameter	Value
Avg. hops from L1 to peer L1	1.0 hops
Avg. hops from L1 to L2	1.0 hops
Avg. hops from L1 to bus	1.0 hops
Write coalescing buffer size	32 entries
L1 hit	1 cycle
Avg. latency of L1 to peer L1 and back	60 cycles
Avg. latency of L1 to L2 to peer L1 and back	80 cycles
Avg. latency of L1 to bus to peer L1 and back	70 cycles
Word size	1 flit
Address size	1 flit
Header size	1 flit
Line size	8 words

We annotated these three benchmarks:

- `rw` a microbenchmark with 2 stages, where the producer writes to every byte in a buffer, and the consumer reads all of the writes.
- `sparse_rw` like `rw`, but producer and consumer only access the first word of every line. This is representative of a "sparse" data access pattern, which eliminates the effect of coalescing.
- `ed` an edge-detection pipeline with 5 stages pipeline with 2 parallel paths. This example exhibits pipeline-parallelism and task-parallelism.
- `cava` Harvard Camera image processing pipeline, with 7 stages. This is a good example of a realistic application.

We considered annotating parboil benchmarks, but they do not naturally exhibit pipelined parallelism, as they are more a test of data-parallelism, so it does not fit the producer/consumer model as well.

The cost model we use is described in table 2. Note the last case (Spandex consumer-owned loads) are where write coalescing becomes relevant. All words written to a line can piggy-back off of the same packet.

Table 2: Latency and network cost of each request in terms of the parameters above.

Scenario	Request	State	State Change	Latency	Traffic Routed
MESI	store	S	issuer to M others to I	L1-bus-peerL1-L1	2 × (L1 to L2 hops) × (Inv: header + address) + (L1 to L1 hops) × (InvAck: header + address)
	store	M	M	L1 hit	0
	load	I	all to S	L1-bus-peerL1-L1	2 × (L1 to Bus hops) × (ReqS: header + address) + (L1 to L1 hops) × (RspSdata: header + address + line)
	load	S	S	L1 hit	0
Spandex	store	S	issuer to O others to I	L1-L2-peerL1-L1	2 × (L1 to L2 hops) × (Inv: header + address) + (L1 to L1 hops) × (InvAck: header + address)
	store	O	O	L1 hit	0
	load	I	all to S	L1-L2-peerL1-L1	2 × (L1 to L2 hops) × (ReqS: header + address) + (L1 to L1 hops) × (RspSdata: header + address + line)
	load	S	S	L1 hit	0
Spandex prod.-owned	store	O	O	L1 hit	0
	load	I	V	L1-peerL1-L1	(L1 to L1 hops) × (ReqV: header + address) + (L1 to L1 hops) × (RspData: header + address + line)
	load	V	V	L1 hit	0
Spandex cons.-owned	store	I	I	L1-peerL1-L1	(L1 to L1 hops) × (ReqWTo: header + address + data) + (L1 to L1 hops) × (AckWT: header + address)
	load	O	O	L1 hit	0

5 EXPERIMENTAL RESULTS

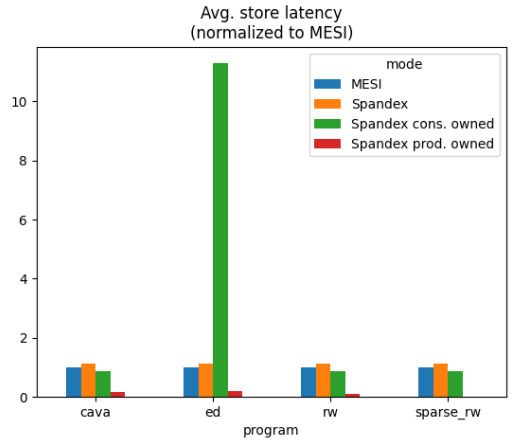

Figure 6: Average Store Latency

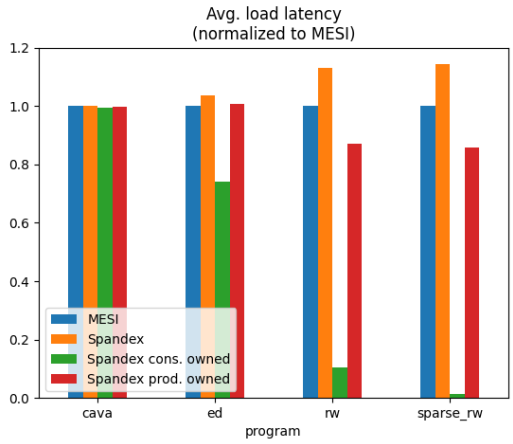

Figure 7: Average Load Latency

Average store latency is least with Spandex producer-owned lines while load latency is least with Spandex consumer-owned lines, both as expected. However, the magnitude of the effect varies widely with the application, by what we believe to be the sparsity of loads relative to the sparsity of stores. NoC usage is least for better for Spandex consumer-owned when the accesses are sparse, and Spandex producer-owned when they are dense.

We believe ed suffers high NoC traffic with Spandex consumer-owned (~10x MESI case) because it has several for-loops that cut 'against the grain', so the stores are unable to be coalesced. However, MESI can still exploit reuse, because it brings the whole line to the writer, so future writes to that line hit *for the duration of the epoch, instead of for the duration of the coalescing window.*

Currently, our compiler pass would select the same kind of request-type mapping for all producer/consumer relationships, so we would select a good all-around cache optimization, like producer-owned for these accesses. This is still dynamic in the sense that accesses outside of the producer/consumer relationship will be regular Spandex requests, while producer/consumer accesses are optimized. However, future work could look at the sparsity of loads and stores

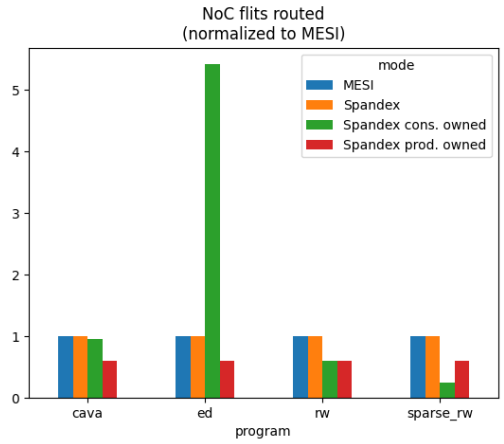

Figure 8: NoC Traffic Routed

and decide for each producer/consumer pair which cache optimization to apply. The benchmarks we ran indicate that when trying to optimize NoC traffic, some cases (`cava`, `ed`, and `rw`) would want producer-owned, but others (`sparse_rw`) would want consumer-owned. NoC traffic routed correlates well with energy utilization, as much of the energy is spent in the interconnect, so this optimization is important for low-power devices [6]. Optimizing for runtime, however, is somewhat more nuanced due to parallelism.

We do not know exactly how the load-latency and store-latency effects the runtime of each benchmark, and we could not simulate the benchmarks for the reasons outlined in section 4. The effect of store-latency is partially isolated by the MSHR buffer. It is probable that `ed` has such a large average store latency that it would fill the MSHR and induce stalls. The impact of load-latency on runtime is also not straightforward because the processor may be able to execute independent instructions out-of-order while waiting for a load. However, many important programs are memory-bound, so these would be sensitive to the load-latency. For example, linked-list traversals would be strongly affected by load-latency.

6 FUTURE WORK

Future work could try to estimate the sparsity of stores compared to the sparsity of loads. Since we have identified the leaf nodes containing the producing and consuming logic, one can use the LLVM analysis tools [10] to ask how many times a static instruction will be run (trip count). Even if this only applied in cases where the code path is statically knowable, a lot of SGEM and image processing code has this property. Then, one could either assume memory accesses are uniform across the buffer or do even more detailed analysis to determine how many accesses are to unique lines in the buffer. Our experiments show there could be extreme benefits in being able to select cache optimization for each access pattern.

We focused entirely on the data-movement between leaves, but future work might optimize the synchronization logic in the HPVM runtime as well. Furthermore, one might also want to look at more kinds of cache optimizations.

Finally, when a simulator becomes available, future work should utilize a proper architectural simulation to produce actual runtime improvement estimates.

7 CONCLUSION

Our experiments show modest gain in certain architectural metrics through compiler-guided cache optimization. More serious gains are within reach, by selecting a different request type mapping for each producer/consumer instead of the same for all.

ACKNOWLEDGMENTS

We would like to acknowledge John Alsop and Sarita Adve. It was invaluable to have feedback and input from the authors of the work we build on [1, 2].

REFERENCES

[1] J. Alsop, M. Sinclair, and S. Adve. 2018. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 261–274. https://doi.org/10.1109/ISCA.2018.00031

[2] Johnathan Alsop, Weon Taek Na, and Sarita V. Adve Matthew D. Sinclair. [n.d.]. Dynamic Coherence Specialization. ([n. d.]).

[3] Gavin M Bierman and Matthew J Parkinson. 2003. Effects and effect inference for a core Java calculus. *Electronic Notes in Theoretical Computer Science* 82, 8 (2003), 82–107.

[4] L. P. Carloni. 2016. Invited: The case for Embedded Scalable Platforms. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.

[5] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. Chou. 2011. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 155–166. https://doi.org/10.1109/PACT.2011.21

[6] Varghese George, Hui Zhang, and Jan Rabaey. 1999. The design of a low energy FPGA. In *Proceedings of the 1999 international symposium on Low power electronics and design*. ACM, New York, NY, USA, 188–193.

[7] Aaron Greenhouse and John Boyland. 1999. An Object-Oriented Effects System. In *ECOOP' 99 — Object-Oriented Programming*, Rachid Guerraoui (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 205–229.

[8] Henry Hoffmann, David Wentzlaff, and Anant Agarwal. 2010. Remote Store Programming. In *High Performance Embedded Architectures and Compilers*, Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–17.

[9] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. HPVM: Heterogeneous Parallel Virtual Machine. *SIGPLAN Not.* 53, 1 (Feb. 2018), 68–80. https://doi.org/10.1145/3200691.3178493

[10] Chris Arthur Lattner. 2002. *LLVM: An infrastructure for multi-stage optimization*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[11] J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564

[12] Jason Merrill. 2003. GENERIC and GIMPLE: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers' Summit*. Citeseer, 171–179.

[13] Daniel J Sorin, Mark D Hill, and David A Wood. 2011. A primer on memory consistency and cache coherence. *Synthesis lectures on computer architecture* 6, 3 (2011), 1–212.