# Optimizations in Stream Programming for Multimedia Applications

by

Eric Wong

S.B., Massachusetts Institute of Technology, 2011

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

August 2012 [September 2012]

Author ........................................................
Department of Electrical Engineering and Computer Science
August 10, 2012

Certified by ........................................................
Saman Amarasinghe
Professor, Thesis Supervisor

Accepted by ........................................................
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

# Optimizations in Stream Programming for Multimedia Applications

by

Eric Wong

Submitted to the Department of Electrical Engineering and Computer Science
on August 10, 2012, in partial fulfillment of the
requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

Multimedia applications are the most dominant workload in desktop and mobile computing. Such applications regularly process continuous sequences of data and can be naturally represented under the stream programming domain to take take advantage of domain-specific optimizations. Exploiting characteristics specific to multimedia programs can provide further significant impact on performance for this class of programs. This thesis identifies many multimedia applications that maintain induction variable state, which directly inhibits data parallelism for the program. We demonstrates it is essential to recognize and parallelize filters with induction variable state to enable scalable parallelization. We eliminate such state by introducing a new language construct that automatically returns the current iteration number of a target filter. This thesis also exploits the fact that multimedia applications are tolerant in the accuracy of the program output. We apply a memoization technique that exploits this tolerance and the repetitive nature of multimedia data. We provide a runtime system that automatically tunes the memoization capabilities for performance and output quality. These optimizations are implemented in the StreamIt programmming language. The necessity of parallelizing induction variable state and performance improvements and quality control of our memoization technique is demonstrated by a case study of the MPEG benchmark.

Thesis Supervisor: Saman Amarasinghe
Title: Professor

# Acknowledgments

This thesis would not have been possible without the guidance and support of Michael Gordon. His extensive knowledge of the streaming domain, direction throughout this project, and patience in guiding me through the steps of conducting research and writing technical papers are what helped this work develop to the state that it is today. I am also thankful to Saman Amarasinghe, for fostering my interest in compilers and performance optimizations of computer systems, and for providing further guidance throughout this project's development.

Finally, I want to thank my family, particularly my parents, grandparents, and little sister. For their unconditional love throughout my entire life and endless support for all of my endeavors, I am forever in their debt.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Multimedia applications represent a large and rich class of programs used across many computer architectures. Such applications, including audio, video, and graphics processing, are prevalent in everyday usage across varying computer systems, ranging from standard laptop and desktop computing systems to handlheld computing devices and wireless cell phones. Multimedia workloads represent a large part of the workload in these computing systems, with estimates of up to 90% of desktop cycles being consumed in multimedia applications [17, 19, 34].

Programmers creating these multimedia applications must consider potential tradeoffs between performance and programming productivity when choosing their target programming domain. Imperative languages and assembly code provide good performance, but may require architecture specific implementations. Functional languages may provide better representability at the cost of performance. If the programmer wants to introduce parallelism to their implementation, such programming solutions would require low-level control and tuning for performance.

Stream programming, on the other hand, naturally represents applications such as audio, video, digital signal processing, and data analysis; applications that are increasingly prevalent as computing moves towards data-centric applications and to the mobile and embedded space. Multimedia programs are often characterized by sequences of distinct processing stages. By virtue of their structure – a graph of independent computational nodes (termed *filters*) with explicit and regular communication – independent processing stages can be mapped to stream programs in a straightforward manner. Furthermore, stream programs are a natural fit for exploiting coarse-grained parallelism suitable for multicore architectures. The interest in streaming applications has spawned a number of streaming languages that target the streaming domain, including StreamIt [48], Brook [15], Cg [37], SPUR [51], Spidle [16], Lime [8], and SPL [27].

This thesis identifies and implements two optimizations specifically aimed for improving performance of multimedia applications implemented under the domain of stream programming. We aim to exploit several characteristics prevalent in streaming implementations of multimedia applications. In particular, we eliminate induction variable state from filters that use them in multimedia applications and implement a tolerant filter memoization scheme that returns very close approximations to actual filter output. These optimizations are implemented in the context of the StreamIt programming language [48]. StreamIt is a high-performance architecture-independent streaming language and compiler system. As a language, it allows programmers to logically connect streaming components into a stream graph. StreamIt relieves programmers of the burden of performing low-level domain-specific optimizations by applying them automatically, including automatically exposing implicit parallelism in stream programs.

### Eliminating Induction Variable State

The first optimization eliminates a specific class of state that filters may maintain, namely *induction variable* state. This state is represented by a transformation on the number of executions of a target filter. This state is used very often in multimedia applications in order to maintain logical counters on the underlying data. For instance, in the MPEG encoder filter shown in Figure 1-1, MotionEstimation uses running counters to maintain positions in a two-dimensional array, representing the underlying blocks of pixels for a picture frame. Induction variable state, and any form of filter state in general, inhibits parallelism in our target filters. This represents a bottleneck for parallelization scalability in the application. We present a keyword

```
int->int filter MotionEstimation(int width,
  int height, int delay,
  int blocks_per_macroblock, int window) {

    ...

  int blockx;
  int blocky;

  init {
    blockx = 0;
    blocky = 0;
  }

  work pop blocks_per_macroblock*64
       push blocks_per_macroblock*64+5 {

        ...

    blockx++;
    if (blockx == (width/16)) {
      blockx = 0;
      blocky = (blocky + 1) % (height/16);
    }
  }
}
```

Figure 1-1: Example of StreamIt filter with induction variable state.

```
int->int filter MotionEstimation(int width,
  int height, int delay,
  int blocks_per_macroblock, int window) {

    ...

  work pop blocks_per_macroblock*64
       push blocks_per_macroblock*64+5 {
    int blockx = iter() % (width/16);
    int blocky = iter() / (width/16)
        % (height/16);

        ...

    blockx++;
    if (blockx == (width/16)) {
      blockx = 0;
      blocky = (blocky + 1) % (height/16);
    }
  }
}
```

Figure 1-2: Example of stateless StreamIt filter using the `iter()`.

solution, namely `iter()`, to effectively represent the number of executions of the respective filter. Accordingly, this would help eliminate induction variable state and expose parallelism in multimedia stream programs. Figure 1-2 shows how such induction variable state would be expressed using the keyword solution. The induction variables in Figure 1-1, `blockx` and `blocky`, are incremented predictably. `blockx` is incremented on each `work()` call and on every `width/16` call. `blocky` is updated once every `width/16` `work()` iteration. Accordingly, each of these induction variables can be expressed in a straightforward manner with the `iter()` keyword.

We extend the StreamIt compiler to be fully aware of the new keyword. Most importantly, we modify StreamIt's source-to-source filter data-parallelization transformation, termed *fission* [24], to correctly parallelize filters that utilize the new keyword. Our formulation correctly calculates the iteration number across data parallel duplicates of the original filter, taking into account the fact that the iterations of the original filter are now split across the duplicates. Furthermore, we show how the modified fission transformation must interact with the steady-state scheduling algorithm of the compiler [33] since it may modify the distribution of iterations between the parallelization duplicates.

The necessity of representing induction variable state and the effectiveness of our parallelization technique is demonstrated via a case study of the StreamIt implementation of the motion estimation stage of MPEG-2 encoding [21]. The motion estimation stage is the most computationally intensive stage of encoding, accounting for at least a third of the total computation in the entire MPEG-2 encoder [20]. A static estimation calculates that 98% of the work of the stage is concentrated in filters that include induction variable state. Originally, these filters cannot be data parallelized because of the presence of state. After modification of these filters to utilize the `iter()` keyword, the entire application is data parallel. The StreamIt compiler, employing our modified fission transformation is able to get significant speedups for the modified version (use of `iter()`) over the unmodified version (with explicit induction state) targeting a commodity multicore SMP processor: 5X for 8 cores, 9X for 16 cores, and 16X for 32 cores.

### Tolerant Memoization for Filter Approximation

The second optimization is aimed to introduce memoization of filters to stream programming. Multimedia applications have a general characteristic in that incoming inputs may often be very repetitive. For instance, videos may render similar images between frames and images may render similar blocks of pixels such as background colors. This repetitive nature lends itself handily to memoization, which allows skipping expensive operations if we have already calculated a corresponding output in a previous execution.

However, as we will show, multimedia applications may not lend itself so handily to the classical function memoization optimization. We instead exploit another common characteristic that many multimedia applications share – that the target consumers are the human senses. Human senses have inherent tolerance to low-level inaccuracies in the underlying data; it is difficult to determine if a specific pixel of an MPEG video frame is exactly the correct color or simply close enough to the correct color. This

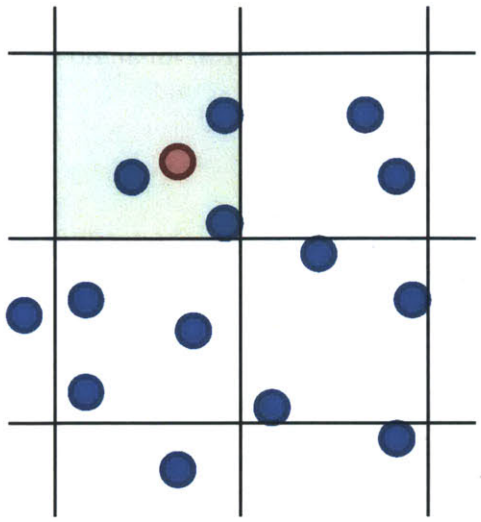


Figure 1-3: Graphical example of a form of Locality-Sensitive Hashing.

characteristic, along with the repetitive nature of the incoming inputs, allows us to memoize certain portions of the stream program, providing potential performance speedups while still generating a "close-enough" output that is indistinguishable to the human senses.

We apply an approximate nearest-neighbor solution for finding "close-enough" memoization matches, called *locality-sensitive hashing* [22, 18, 5]. Figure 1-3 shows a graphical example of a form of locality-sensitive hashing. This algorithm relies on hash functions that hash closer points to the same hash buckets and farther points in different hash buckets. Points in our sample space are assigned to hash buckets; in Figure 1-3 each square grid represents a hash bucket. To find near neighbors, our query point is hashed to identify its hash bucket, thus returning our near neighbors. This technique allows us to find values that can generate results that are virtually indistinguishable to the human senses from the original results.

We extend the StreamIt compiler to inject memoization capabilities into target filters in the stream graph. Such filters will be automatically tuned for quality during runtime. Memoization capabilities are automatically disabled if it is determined the overhead of memoization queries exceeds the benefits of skipping the `work()` . We present the performance improvements and compare the quality of the resulting outputs of various multimedia applications, including a decoder for MPEG-2, a video compression standard and a decoder for MP3, an audio compression standard.

## Contributions

This work expands the domain of algorithms for which the stream programming model, and its associated compiler technologies, is effective at automatically managing parallelism. For automatic parallelization to become mainstream, computer scientists must continue to remove common sensitivities (in this case induction variable state) that inhibit parallelization. Furthermore, the work introduces a potential for performance improvements if the application presents some tolerance for inaccuracies. Such techniques aim to improve runtime performance of multimedia applications in the stream programming domain. In accordance to this, this thesis makes the following contributions:

1. *Directly represent the basis of induction variable state in the language.* We introduce a new expression that is attractive to programmers and precludes the need for a programmer to maintain induction state in a stream program.

2. *Automatic parallelization of induction variable state.* We provide modifications to the foundational parallelization transformation of the StreamIt language so that it can data parallelize filters with induction variable state utilizing the `iter()` expression.

3. *Case study of induction variable state.* We include a motivating case study of applications from the StreamIt benchmark suite that include induction variable state. Furthermore, we demonstrate the performance and scalability benefits of our approach via the motion estimation stage of MPEG2 encoding.

4. *Tolerant memoization for stream programs.* We introduce memoization capabilities to stream programs. We relax the requirement of having exact input matches provided there is a tolerance for slight output inaccuracies.

5. *Performance and quality analysis of memoization for multimedia applications.* We apply these memoization capabilities to various multimedia applications and perform runtime tuning to optimize performance and output quality. For MPEG-2 we found slight performance improvements of 1.14X for 40% hit rates, 1.22X for 55% hit rates, and 1.27X for 60% hit rates. For MP3 we found performance improvements of 1.42X for 40% hit rates and 2.02X for 60% hit rates. For both benchmarks, this optimization yielded outputs that were indistinguishable to the human senses.

The remainder of the thesis is organized as follows. Chapter 2 provides a brief introduction to the StreamIt language and its capabilities. Chapter 3 introduces induction variable state as applied to stream programming and details the implementation and application of the induction variable keyword solution. Chapter 4 introduces the locality-sensitive hashing technique and details the application of this technique to filter memoization.

# Chapter 2

# The StreamIt Programming Language

StreamIt is a high-performance streaming language and compiler system. The compiler is publicly available [1] and include backends for multicore architectures, clusters of workstations, and Tilera architectures.

StreamIt is built on the Synchronous Dataflow model of streaming computations [35]. This model allows programmers to implement independent actors, known in StreamIt as *filters*, which act on data items from input channels and push data to output channels. These filters are composed into a stream graph modeling the desired computation. The synchronous dataflow model requires filters to have static communication rates, thus fixing the number of data items the filter consumes and produces. This restriction allows compilers to perform static analysis of and optimizations to the stream graph and schedule filters for optimal execution.

StreamIt as a programming language aims to provide high-productivity and high-performance for the domain of stream programming. The principle goals of the StreamIt system include:

- Automating streaming-specific optimizations. The model of streaming computations is unique from other models of programming due to the structuring of stream programs. Accordingly, we can take advantage of optimizations such as the implicit parallelism in stream programs.

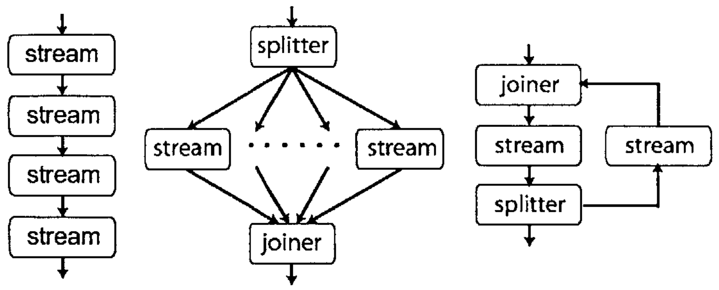
- Improving programmer productivity. StreamIt programs are hierarchical by nature. As such, they are expressive through easy-to-use hierarchical abstractions, with an emphasis on a filter's code reuse. Furthermore, optimizations are performed without the programmer needing to manually tune the program or manually implementing them. The programmer simply needs to express the program in an algorithmically sound manner.

## 2.1 Filters: Basic Stream Units

As described before, the basic unit of a StreamIt program is a *filter*. Each filter represents an actor in the synchronous dataflow model, whereby it reads data items from an input tape, processes them, and then writes processed data items onto an output tape. An example StreamIt filter appears in Figure 1-1.

The filter is defined by its `work()` function. The `work()` function is the steady-state execution step of the filter and is called repeatedly by the StreamIt runtime system. Within the `work()` function, a filter may *peek* at a given element off the input tape, *pop* the first data item off the input tape, or *push* a data item onto the output tape. The rates at which each filter invocation peeks, pops, and pushes are declared as part of the `work()` function. Note the *peek* rate must always be greater than or equal to the *pop* rate. If the *peek* rate exceeds the *pop* rate, the filter represents a sliding window computation where some input elements are accessed across multiple invocations of the filter.

Filters may declare an `init()` function to initialize any internal data structures. Filters may also define a `prework()` function to perform specialized processing of data items off the input stream prior to the steady state. This is necessary for cases where their initial processing has a different input or output rate than the steady-state processing.



(a) **A pipeline.** (b) **A splitjoin.** (c) **A feedbackloop.**

Figure 2-1: Hierarchical stream structures supported by StreamIt

## 2.2 Stream Hierarchical Structures

As depicted in Figure 2-1, StreamIt provides three hierarchical primitives for composing filters into stream graphs, namely *pipeline*, *splitjoin*, and *feedbackloop*. StreamIt requires filters to be connected according to these hierarchical structures. Each hierarchical construct has itself a single input tape and a single output tape, thus allowing for complex nested structures of hierarchical primitives of filters. The term *stream* is used to refer to any of these hierarchical groups or the individual filters themselves.

A *pipeline* represents a sequential composition of streams. The pipeline connects the single output tape of each child stream to the input tape of the subsequent stream. The ordering of the streams is indicated through successive `add` calls. Predictably, the input tape of the pipeline feeds into the input tape of the first child stream and the output tape of the final child stream feeds into the output tape of the pipeline.

A *splitjoin* represents a parallel set of streams. The input tape of the splitjoin feeds into a *splitter*, which splits the single input tape into multiple tapes. StreamIt predefines two types of splitters: *duplicate* splitters copy every item on the single splitjoin input tape to each multiple tape, and *round-robin* splitters send the data items on the input tape to a different tape in a weighted round-robin fashion. The StreamIt syntax for the splitters are, respectively, `split duplicate` and `split roundrobin(`$w_1$`, ...,` $w_n$`)`, indicating the first $w_1$ items get sent to the first child, the next $w_2$ items get sent to the second child, and so on. These multiple tapes correspond to individual children streams of the splitjoin, operating in parallel with one another. The splitjoin child streams are indicated with successive add calls. The output tape of all of these children streams feed into a *joiner*, which converges the output tapes in a weighted round-robin fashion.

Finally, a *feedbackloop* represents a cycle in the stream graph. The feedbackloop contains a body stream, whereby a backwards feedback path is created, a loop stream, which performs computation on the feedback path, a splitter, which distributes data between the feedback path and the output tape, and a joiner, which merges items between the feedback path and the input tape.

## 2.3 Execution Model and Filter Schedules

A filter's *schedule* gives a multiplicity for the filter in a stream graph. The multiplicity indicates how often the filter's `work()` function should be invoked (or `prework()` function on the first iteration of the filter).

The *steady-state schedule* can be calculated such that all filters fire in the schedule, and the schedule can be repeated indefinitely [35]. Steady-state execution of the graph entails repeating the steady-state schedule for as many inputs as is expected. Running a steady-state schedule will leave exactly the same number of data items on all channels before and after the execution. Execution of the stream graph is conceptually wrapped in an outer loop that continuously executes the steady-state schedule. All the multiplicities of the steady-state can be multiplied by the same constant $m$, and the result will still be a valid steady-state. We call this process *increasing* the steady-state of the graph by $m$.

Furthermore, an *initialization* schedule enables the steady-state schedule in the presence of peeking filters. An initialization schedule is required if peeking is present in a graph to enable the calculation and execution of a steady-state schedule [33]. During application execution, the `init()` function is called once for each filter, then the initialization schedule is executed once, followed by an infinite repetition of the steady-state schedule.

The StreamIt program is defined with a top-level stream that defines the hierarchical stream graph. The leaves of the hierarchy correspond to the individual filters interconnected by channels. Once the schedules have been calculated, the filters are translated to fit the schedules accordingly, via increasing the initialization or steady-state multiplicity and the push, pop, and peek rates for the `work()` and `prework()` .

## 2.4 Parallelism in Streaming Languages

The stream graph of a StreamIt program exposes three types of coarse-grained parallelism, task, data, and pipeline parallelism.

*Task parallelism* refers to pairs of filters in a stream graph that are on different parallel branches. This parallelism is exposed directly by the stream graph, through the splitjoin construct. Specifically, the output of each filter will never reach the input of the other. Accordingly, these filters have no dependencies on each other, and can be safely run in parallel.

*Data parallelism* refers to filters that have no dependencies between execution steps. Filters that are data-parallel maintain no state, and accordingly different execution steps working on different inputs can be performed independently. While such stateless filters can provide unlimited data parallelism, it can also potentially increase buffering and latency.

*Pipeline parallelism* refers to filters connected in a pipeline. Filters that produce data items and filters that consume them can be mapped onto different cores. There must be an initialization schedule to prime the inputs, Once enough data is available, the producer and consumer are independent of one another. Extra synchronization and effective load balancing is required to maintain the execution.

Introducing parallelism to the streaming program provides the benefits of performing work that is wholly independent of one another simultaneously. However, in adding parallelism to the program, communication and synchronization costs are introduced to maintain the thc ordering of the data items on the stream and the correctness of the execution. If a particular filter performs too little work, such costs may overtake the benefits of executing in parallel.

## 2.5 Stream Graph Transformations

Accordingly, it is necessary for the StreamIt compiler to perform stream graph transformations in order to adjust the granularity of the stream and its filters for the optimizations that require them. As one of StreamIt's goals is to aid in programmability, the programmer should express the program at a level of granularity that suits their understanding of the program, rather than for what is needed for certain optimizations. Such graph transformations are done automatically without the programmer needing to modify their code in any way.

*Fusion* is the process of merging adjacent filters into a single large filter. Fusion aims to decrease the granularity of the stream graph and merging the work (under the appropriate schedules) of all affected filters. More details of the implementation of the fusion transformations can be found at [24].

Decreasing the granularity of the stream graph through merging `work()` functions of several filters can enable other optimizations to be performed on the program. Filter fusion can shorten the live range of certain variables and allow instructions to be reordered. Furthermore, it can improve load balancing by allowing a larger filter to be split across multiple cores.

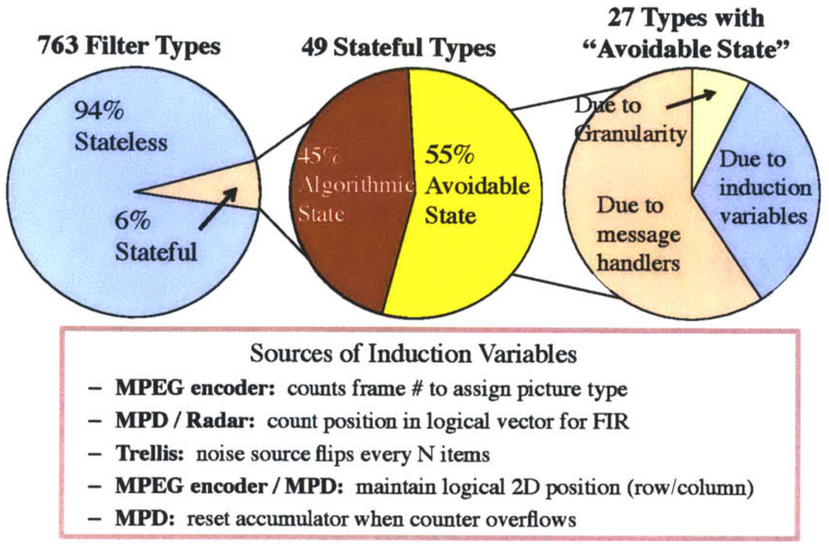
*Fission* is the process of automatically data-parallelizing filters. The transformation duplicates the target filter multiple ways and wraps these duplicated filters, termed *fission products*, in a round-robin splitjoin. Each of these products can be assigned to distinct cores, thus introducing data parallelism. More details of the implementation of the fission transformations can be found in §3.5.1 and [24].

# Chapter 3

# Induction Variable State in Stream Programs

As described in §2.4, the key to effectively parallelizing stream programs is to take advantage of data parallelism present in filters that do not maintain state [25]. Data parallelism provides load-balanced and limitless parallelism (as long as input data is available). As described, however, data parallelism of a filter is restricted by the existence of state. Many real-world streaming applications include filters with explicit state. A prominent example of such state is *induction variable* state, which is derived from the number of executions of the filter. For example, a filter may keep track of how many times it has been invoked, in order to perform a special action every N iterations. An MPEG encoder has a concrete example of such a filter; motion estimation uses running counters to maintain positions in a two-dimensional array, representative of blocks of pixels for a picture frame. The presence of induction variable state inhibits data parallelism, and represents a bottleneck for parallelization scalability of the application.

In this chapter we introduce techniques to represent, capture, and parallelize induction variable state in stream programs. Our research is conducted in the context of the StreamIt programming language [48]. Figure 3-1 summarizes the finding of [47] in regards to induction variable state on the StreamIt benchmark suite. Of the 65 benchmarks included in the suite many of the real world applications include in-



Sources of Induction Variables

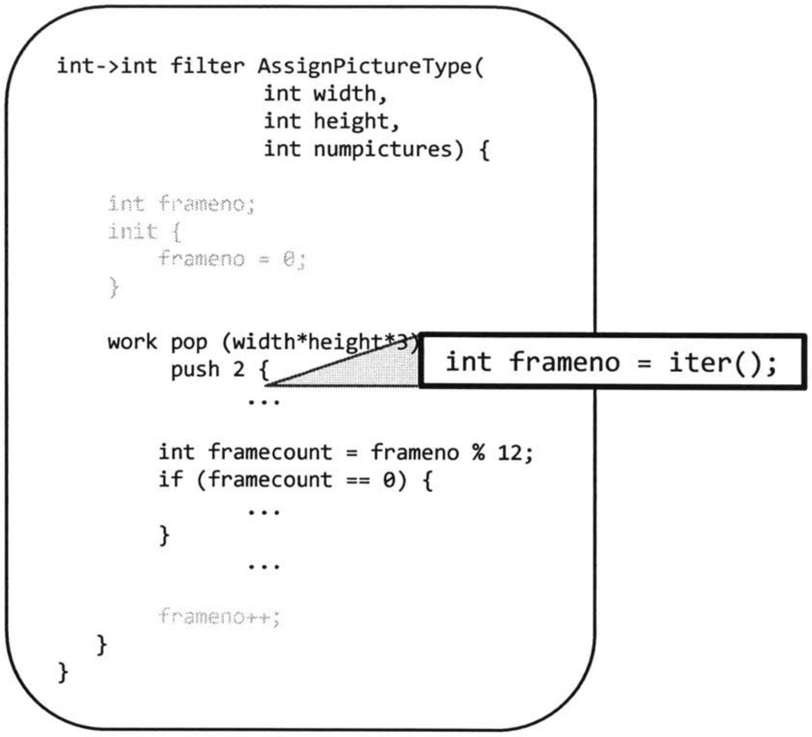
- **MPEG encoder:** counts frame # to assign picture type
- **MPD / Radar:** count position in logical vector for FIR
- **Trellis:** noise source flips every N items
- **MPEG encoder / MPD:** maintain logical 2D position (row/column)
- **MPD:** reset accumulator when counter overflows

Figure 3-1: Summary of the findings of [47]. Of the 763 filters of 65 programs (over 35k lines of code), 55% of stateful filters include avoidable state, and much of that is due to induction variable state in real world applications.

duction variable state, including MPEG encoder, beamforming, Trellis, and Medium Pulse Doppler. It is important to represent induction variable state because it is a common implementation idiom and parallelizing such state removes scalability bottlenecks to drastically improve parallelization performance. As we will show, even a small amount of stateful computation will severely limit parallelization scalability as we quickly transition to the manycore era with hundreds and thousands of cores.

## 3.1 Induction Variable State

Traditional induction variables encapsulate all variables that are increased or decreased by a fixed amount with every iteration of a loop [3]. Induction variable state as applied to stream programming is a class of state that requires keeping count of how often a filter has been invoked. Common usage of induction variable state includes performing some special action after a certain number of iterations and keeping track of array index positions. Some filters, including some used in MPEG2 encoder and MPD, maintain multiple induction variables as well, which may either be dependent



```
int->int filter AssignPictureType(
                  int width,
                  int height,
                  int numpictures) {

    int frameno;
    init {
        frameno = 0;
    }

    work pop (width*height*3)
         push 2 {
                ...

        int framecount = frameno % 12;
        if (framecount == 0) {
                ...
        }
                ...

        frameno++;
    }
}
```

```
int frameno = iter();
```

Figure 3-2: Example StreamIt filter, as used in MPEG Encoder.

or independent of each other.

Many applications in the StreamIt benchmark suite, including MPEG2 encoder, Medium Pulse Doppler, Trellis, and FIRBank, maintain induction variable state by creating a mutable state field in the corresponding filter. This state can be set to the desired starting value. The induction variable is consistently updated at some point during the work call. For many use cases, this induction variable may need to be reset if it reaches a certain threshold.

Figure 3-2 illustrates a common pattern of explicit induction variable state. This filter is based on the `AssignPictureType` filter in the MPEG2 encoder. The implementation of the filter maintains a filter variable `frameno` that is incremented on each call of the `AssignPictureType`'s work function. This variable represents state. Each iteration of the filter uses a value for *frameno* dependent on what the value of *frameno* was in the previous iteration. The `framecount` variable is derived from this state and used in control flow.

As presently constructed, induction variable state forces the corresponding filter to be run in sequential order. In providing a mutable state whose value is dependent on the previous execution step, it is necessary to run a filter execution step and establish the induction variable value before moving on to the next execution step. The tradition fission transformation would not be able to parallelize this filter without understanding how to properly distribute the calculation of the state. As such, it is not possible to create duplicates and run them in a parallel fashion.

With induction variable state, it is possible to make the compiler aware of such state through the keyword solution. Figure 3-2 shows how the same AssignPicture-Type filter can be rewritten to use the keyword. The compiler can generate the value for `iter()` for each iteration of the AssignPictureType filter independent of previous iterations. Duplicates of the filter can be made and data parallelism opportunities can be exploited.

## 3.2 Scalability Implications of Eliminating Induction Variable State

Table 3.1 indicates programs in the StreamIt benchmark suite that use induction variable filters (not including source filters) in the manner described above. The StreamIt compiler provides static estimations of work performed in filters. The above table indicates the percentage of work performed in specifically the induction variable filters.

The majority of programs do not have substantial work performed in filters using induction variables; FIRBank and MPD contain only a few stateful filters whose total work concentration is fairly low. The MPEG-2 motion estimation subset is the only exception because its stream graph is comprised mostly with stateful filters. However, eliminating any form of state will have a large impact on runtime performance even on programs with low work concentration in stateful filters. We model the potential speedups of a particular stateful program in this section. For the purpose of this

|  | Filters with Induction State |  |  |
|---|---|---|---|
| Benchmark | Types | Instances | Work |
| CoarseSerializedBeamFormer | 1 | 1 | 0.32% |
| FHRFeedback | 1 | 1 | 0.03% |
| FIRBankPipeline | 1 | 1 | 3.94% |
| MPD | 4 | 6 | 3.01% |
| MPEGencoder (Motion Estimation) | 1 | 2 | 98.22% |
| MPEGencoder (Picture Preprocessing) | 1 | 1 | 2.69% |

Table 3.1: Benchmarks using induction variable state and estimations on work performed in filters with induction state.

analysis, assume no communication cost between filters. Also assume the compiler exposes no pipeline parallelism. This assumption forces the serialization of stateful filters on the stream graph.

Let $N$ be the number of cores we are planning to parallelize over. Let $\sigma$ be the percentage of work performed in stateful filters that can have its state eliminated, in this case solely filters that use induction variable state.

If $\sigma = 0$, the entire program is stateless. The program can be fused to coarsen the granularity, then fissed and mapped to all of the available cores. Each core would perform $\frac{1}{N}$ of the total work. Thus with no state, the program can exhibit speedups of up to $N$ times the single-core runtime.

For filters that contain stateful work, $1-\sigma$ of the work in the program is considered stateless, and thus can be fissed and assigned to $N$ individual cores. The stateful filters cannot be parallelized, and is sequential to all work in the program. The total serialized work is $\frac{1-\sigma}{N} + \sigma$. Thus the total speedup is the serial work divided by the new parallelized work.

$$\frac{1}{\frac{1-\sigma}{N} + \sigma} = \frac{N}{1 + \sigma(N-1)}$$

We can characterize the amount of speedup between a completely stateless program to an equivalent stateful program with $\sigma$ percentage of stateful work. This is

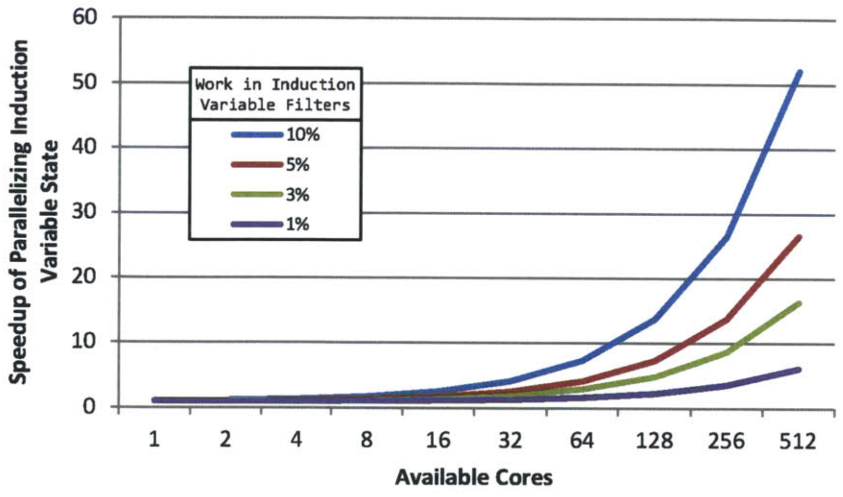


Figure 3-3: Theoretical speedups of a stateless programs over the corresponding $\sigma$% stateful programs.

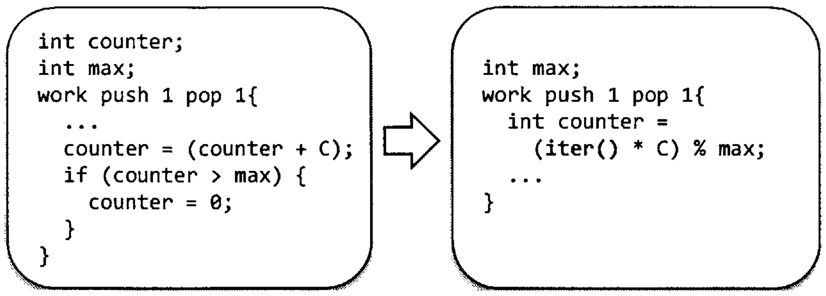
simply:

$$\frac{N}{\frac{N}{1+\sigma(N-1)}} = 1+\sigma(N-1)$$

Figure 3-3 indicates the potential speedups over stateful programs given stateful work percentages and a varying number of cores. Even benchmarks in the suite that exhibit only 3% work in stateful filters can exhibit 8x speedups with 256 cores. Providing a means to remove state from filters that exhibit very small amounts of work relative to the rest of the program can still generate substantial speedups in the near future.

## 3.3 The Iteration Keyword

We remove this potential throughput bottleneck by introducing a new language construct that maintains a value indicating how often the corresponding filter has been invoked. With the introduction of a new language construct, hereafter referred to as `iter()`, the programmer can implement a common idiom much more easily. Furthermore, it allows programmers to circumvent the scalability limitations caused by



```
int counter;
int max;
work push 1 pop 1{
  ...
  counter = (counter + C);
  if (counter > max) {
    counter = 0;
  }
}
```

```
int max;
work push 1 pop 1{
  int counter =
    (iter() * C) % max;
  ...
}
```

Figure 3-4: Translation of a simple filter with induction variable state that resets after a certain value.

stateful filters limited by induction variable state.

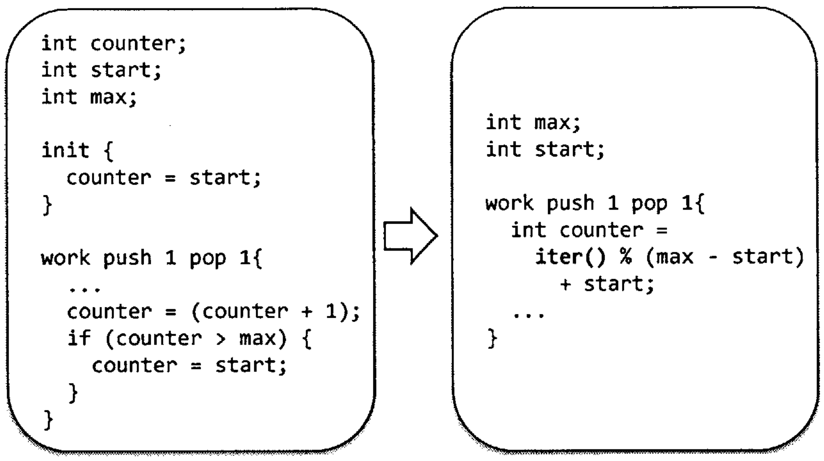
### 3.3.1 Language Construct Approach

`iter()` yields the same value as a field that is incremented on every `prework` and `work` invocation. The value returned by `iter()` indicates how often the filter has been invoked.

Induction variable state requires the user to maintain mutable state, updating such state within the filter invocation, and potentially resetting them when required. The programmer can eliminate much of this code simply by arithmetically manipulating `iter()` usage, as illustrated in Figures 3-4 and 3-5. The implementation using `iter()` describes the induction variable's full behavior much more succintly. It is easy to lose track of where or what causes the induction variable to be updated, but the use of `iter()` allows all of this information to be localized.

Multiple induction variables, as used with the stateful filter in Figure 3-6, can be redefined in terms of `iter()` as well. It is not necessary to maintain multiple separate values for state. Independent and nested induction variable state alike can be defined in terms of `iter()`.

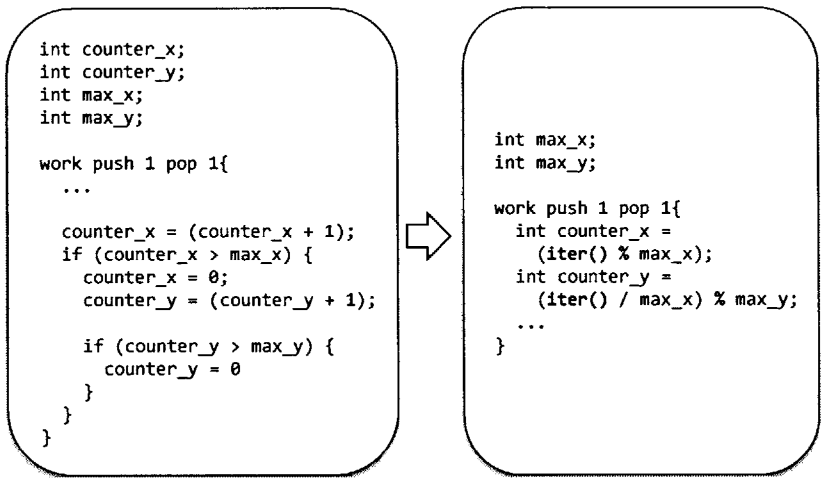
Using `iter()` captures a common idiom in a very simple manner. Programmers can write programs that use induction variable state by simply defining their induction variables in terms of how often the filter has been invoked. Induction variables can all be derived from this value arithmetically. The transformations from filters using induction variable state to using `iter()` are also simple for users to implement.



```
int counter;
int start;
int max;

init {
  counter = start;
}

work push 1 pop 1{
  ...
  counter = (counter + 1);
  if (counter > max) {
    counter = start;
  }
}
```

```
int max;
int start;

work push 1 pop 1{
  int counter =
    iter() % (max - start)
      + start;
  ...
}
```

Figure 3-5: Translation of a simple filter with induction variable state that resets to a special value.



```
int counter_x;
int counter_y;
int max_x;
int max_y;

work push 1 pop 1{
  ...

  counter_x = (counter_x + 1);
  if (counter_x > max_x) {
    counter_x = 0;
    counter_y = (counter_y + 1);

    if (counter_y > max_y) {
      counter_y = 0
    }
  }
}
```

```
int max_x;
int max_y;

work push 1 pop 1{
  int counter_x =
    (iter() % max_x);
  int counter_y =
    (iter() / max_x) % max_y;
  ...
}
```

Figure 3-6: Translation of a filter with nested induction variables.

Some of the common transformations of patterns using induction variables to filters using `iter()` can be summarized below.

- Incrementing the induction variable by $C$ for each invocation scales the running value by $C$, as illustrated in Figure 3-4.

- Placing an upper bound on the induction variable value establishes a range of values it may take. This requires taking the running value modulo the size of this range, as illustrated in Figure 3-4.

- Initializing and resetting the induction variable to a special starting value reduces the total range of values the induction variable can take. The running value is taken modulo the size of this new range to calculate the current excess over the special starting value. We add this value to the special starting value, as illustrated in Figure 3-5.

- A nested induction variable updates only when another induction variable reaches a certain threshold. We scale the running value down according to this threshold to indicate it is incremented only once every time this threshold is reached by the other induction variable, as illustrated in Figure 3-6.

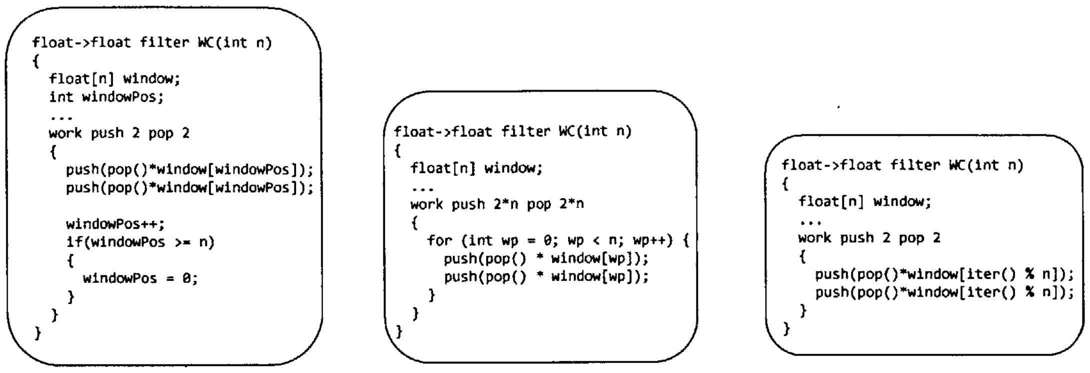
### 3.3.2 Comparison to Automatic Analysis

Another approach to detecting induction variables in stream programmers involves automatically recognizing induction variable usage in filter construction.

The approach of automatic analysis would idiomatically detect a variable modified by a statement similar to `var=var+1`. For programs such as the MPEG2 picture preprocessing subset, as depicted in Figure 3-2, there is only one update to the induction state.

Very few iteration filters, outside of source filters, use induction state in this limited capacity. Consider Figure 3-4, where the induction variable is incremented at each iteration step but resets at a threshold value. This pattern is common in programs that use induction filters; MPD and FIRBank use this technique to iterate across a provided array one element per iteration step. Trellis uses this technique to perform a special action once this threshold is reached. There is more than one location where the induction state can be updated. Furthermore, the induction state may not be updated at every invocation.

A filter may also have multiple dependent induction variables. The stateful filter in Figure 3-6 shows a filter using nested dependent induction variables. Co-induction variables may be constructed to reset the value of other induction variables after it reaches a certain value.



```
float->float filter WC(int n)
{
  float[n] window;
  int windowPos;
  ...
  work push 2 pop 2
  {
    push(pop()*window[windowPos]);
    push(pop()*window[windowPos]);

    windowPos++;
    if(windowPos >= n)
    {
      windowPos = 0;
    }
  }
}
```

(a)

```
float->float filter WC(int n)
{
  float[n] window;
  ...
  work push 2*n pop 2*n
  {
    for (int wp = 0; wp < n; wp++) {
      push(pop() * window[wp]);
      push(pop() * window[wp]);
    }
  }
}
```

(b)

```
float->float filter WC(int n)
{
  float[n] window;
  ...
  work push 2 pop 2
  {
    push(pop()*window[iter() % n]);
    push(pop()*window[iter() % n]);
  }
}
```

(c)

Figure 3-7: Three versions of the weights calculation filter from Medium Pulse Doppler (MPD). (a) is the original filter with explicit induction variable state. (b) is a coarsened version without state. (c) is a version that utilizes the `iter()` keyword to avoid state.

Accordingly, the automatic analysis must be flexible enough to detect the induction variable and potentially unpredictable updating statements. Automatic analysis must be able to detect incrementing statements that may not be called on every invocation. The process of simply detecting and identifying induction variables can potentially branch into many cases each needing to be specially implemented. With the keyword solution, what value the induction variable should take is left to the user to control, without potentially inhibiting data parallelism opportunities.

It may also be difficult to detect how the induction value will be updated. The keyword solution has the added benefit of only maintaining a single value that is predictable in its updates. The value that the keyword returns is simply the number of times the filter has been invoked. This value is always incremented by one at the end of every work call.

### 3.3.3 Comparison to Coarsening

In cases of explicit induction variable state where the induction variable is reset after a certain number of iterations, the filter can be converted into a stateless filter by *coarsening* the filter, increasing the number of input items that are required for the work function execution. Figure 3-7(a) lists the weight calculation filter from the Medium

Pulse Doppler (MPD) benchmark. The filter includes explicit induction variable state as originally implemented by the programmer. This filter can be rewritten without state (and without using the `iter()` keyword) by coarsening the filter such that each work function execution operates on a larger subset of the input. Figure 3-7(b) lists a coarsened implementation that is stateless. In the coarsened implementation the filter requires $2n$ input items.

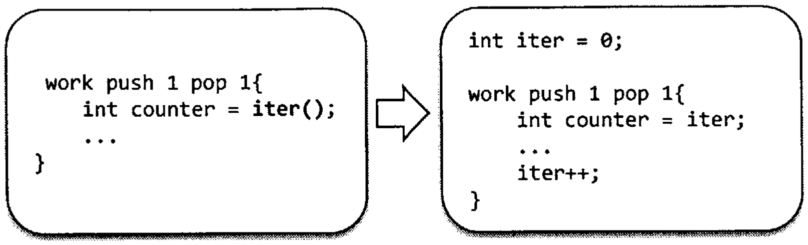
Figure 3-7(c) lists the implementation of the weights calculation filter utilizing the `iter()` keyword. Notice that the filter operates at a finer granularity versus the coarsened version and that it operates at the same granularity as the original filter. Although the `iter()` implementation includes a modulo operation per output, calculation of outputs will be parallelized (see Section 3.5).

The mantra of stream programming is that the programmer should not be burdened with parallelization, granularity, communication or synchronization concerns. Implementing a filter at a fine granularity allows the compiler or runtime to decide on the best granularity for a given architectural target. In practice the use of the `iter()` keyword is preferred over a coarsening conversion because:

- The programmer grasped the algorithm and implemented the application at the fine granularity. A language should constrain the programmer as little as possible for the sake of performance.

- The coarse granularity implementation requires larger input and output buffers to implement because of the larger push and pop rates. Larger buffers occupy more of the cache and could evict filter data or instructions are needed during execution. Thus there could be more accesses to longer latency memory hierarchies [46].

- Larger input and output rates also interact with the steady-state scheduling algorithm. Since the scheduling algorithm is performing many cascading LCMs, a single filter with large input and output rates will increase the multiplicities of all filters of the application, requiring more buffering and increasing latency [33].



```
work push 1 pop 1{
    int counter = iter();
    ...
}
```

```
int iter = 0;

work push 1 pop 1{
    int counter = iter;
    ...
    iter++;
}
```

Figure 3-8: Desugaring a filter using `iter()` keyword.

The use of the `iter()` keyword does not force the programmer to sacrifice latency for parallelization.

## 3.4 Desugaring

A filter is classified as an *iteration filter* when there exists a use of `iter()` in its `prework()` or `work()` function. In this section, we introduce a simple desugaring transformation that will convert an iteration filter into a filter that does not include the `iter()` keyword, but implements the correct semantics of the `iter()` uses. We accomplish this by introducing a state field that records the current iteration of this filter. This state field is updated in a consistent and compiler-defined manner, so later transformations and passes (such as fission, see §3.5) can reason about the state.

The `iter()` expression is replaced with an access to a field holding the value of the iteration count. Again, the filter is given a definition of this field only if it is classified as an iteration filter, this field is not added to non-iteration filters. The `work()` and `prework()` function (if it exists) are appended with incrementing statements that update the iteration value. The name of the state field is compiler defined for easy recognition by later passes, for the remainder of this paper the field is named *iter*.

Figure 3-8 provides a code example of the desugaring process. The left filter in Figure 3-8 shows an iteration filter. All references to the keyword are replaced to field accesses to the *iter* field. Figure 3-8 shows the StreamIt code representation after the iteration keyword is desugared.

## 3.5 Fission of Induction State

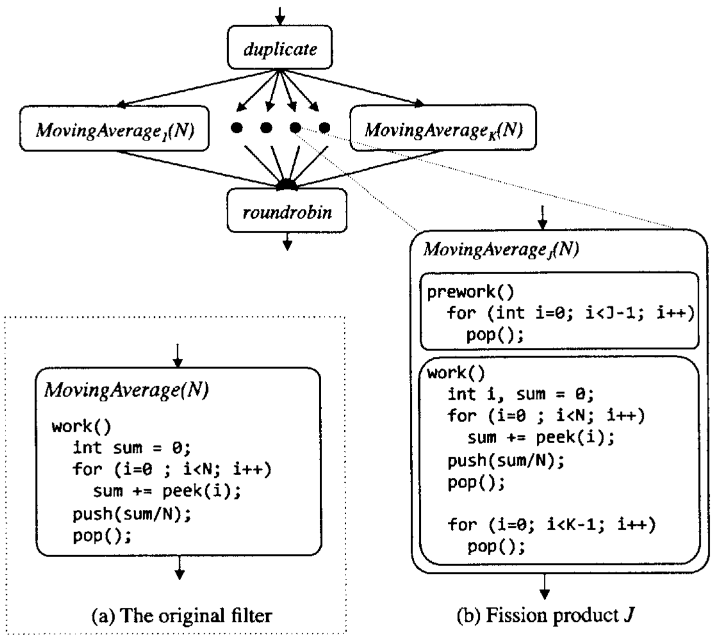
Data parallelism is added to a stream program through a transformation known as *fission*. Fission is the process of duplicating a stateless filter and wrapping these duplicates in a splitjoin so that input data is distributed correctly and output data is collected in correct order. The duplicated filters, known as products, can be assigned to distinct cores, thus introducing data parallelism.

### 3.5.1 The Original Fission Transformation

The unmodified fission transformation is described in [24]. In this section we give some background for the transformation. The original fission transformation is applicable only to a stateless filter, i.e., a filter that does not include a state field to which non-constant values are assigned in the `prework()` or `work()` functions. The simplest case for fission is when it is applied to a filter that does not peek (i.e., the peek rate equals the pop rate). In this case, the filter is duplicated the desired number of ways, and placed in a round-robin splitjoin, where the weights of the splitter are uniform and equal to the pop rate of the orignal filter, and the weights of the joiner are uniform and equal to the push rate of the original filter.

A more complex case is illustrated in Figure 3-9. The base filter, MovingAverage, is a peeking filter with a sliding window (the pop rate is 1 and the peek rate is $N$). In this case the filter is placed in a splitjoin with a duplicate spitter and a round-robin joiner. Each individual MovingAverage$_j$ product has the `work()` function of the base filter with some additional code to implement the sliding window correctly. The duplicate splitter is introduced because of the fact that a single input item is read by multiple filters (due to the overlapping sliding windows between the products). All input items are duplicated to all products, with unneeded items being decimated by popping them at the end of the new `work()` function. The `prework()` function is introduced to decimate input data that is consumed by previous products during before the first iteration.

Providing the complete algorithm for the fission transformation is beyond the



```
MovingAverage(N)
work()
  int sum = 0;
  for (i=0 ; i<N; i++)
    sum += peek(i);
  push(sum/N);
  pop();
```

```
MovingAverage_J(N)
prework()
  for (int i=0; i<J-1; i++)
    pop();

work()
  int i, sum = 0;
  for (i=0 ; i<N; i++)
    sum += peek(i);
  push(sum/N);
  pop();

  for (i=0; i<K-1; i++)
    pop();
```

(a) The original filter (b) Fission product $J$

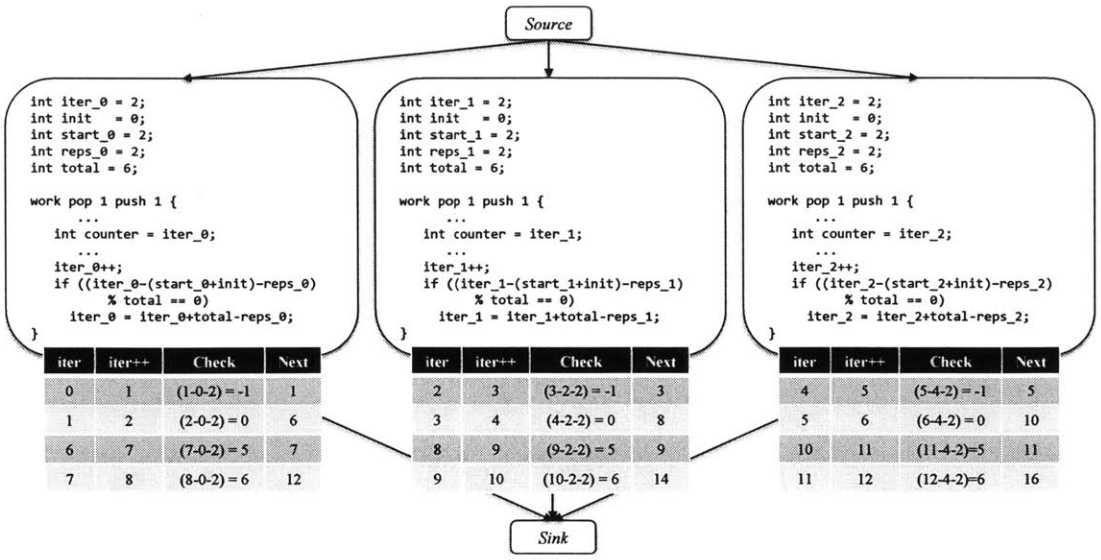
Figure 3-9: Fission of a stateless filter that peeks. (a) The original filter (b) the splitjoin and a detail of fission product $J$.

scope of this paper, but from the example in Figure 3-9, the reader can gleam the necessary background information to understand our modifications to the algorithm in the next section.

### 3.5.2 Modifications to Fission Transformation

Fission is applicable only to stateless filters, as the duplication process does not guarantee consistent behavior if filters contain state that changes between iterations. Because the desugared iteration filters actually use mutable state to keep track of iteration values, the fission process must be modified to handle iteration values. If a filter includes only state in the form of the *iter* field introduced by the desugaring process (see §3.4), then it is considered a candidate for the modified fission process described in this section.

The code additions described below are applied to the fission products after the original fission transformation algorithm introduces the splitjoin structure and (for peeking filters) makes it changes to the `work()` and `prework()` functions.



```
int iter_0 = 2;
int init   = 0;
int start_0 = 2;
int reps_0 = 2;
int total = 6;

work pop 1 push 1 {
     ...
   int counter = iter_0;
     ...
   iter_0++;
   if ((iter_0-(start_0+init)-reps_0)
         % total == 0)
     iter_0 = iter_0+total-reps_0;
}
```

| iter | iter++ | Check | Next |
|---|---|---|---|
| 0 | 1 | (1-0-2) = -1 | 1 |
| 1 | 2 | (2-0-2) = 0 | 6 |
| 6 | 7 | (7-0-2) = 5 | 7 |
| 7 | 8 | (8-0-2) = 6 | 12 |

```
int iter_1 = 2;
int init   = 0;
int start_1 = 2;
int reps_1 = 2;
int total = 6;

work pop 1 push 1 {
     ...
   int counter = iter_1;
     ...
   iter_1++;
   if ((iter_1-(start_1+init)-reps_1)
         % total == 0)
     iter_1 = iter_1+total-reps_1;
}
```

| iter | iter++ | Check | Next |
|---|---|---|---|
| 2 | 3 | (3-2-2) = -1 | 3 |
| 3 | 4 | (4-2-2) = 0 | 8 |
| 8 | 9 | (9-2-2) = 5 | 9 |
| 9 | 10 | (10-2-2) = 6 | 14 |

```
int iter_2 = 2;
int init   = 0;
int start_2 = 2;
int reps_2 = 2;
int total = 6;

work pop 1 push 1 {
     ...
   int counter = iter_2;
     ...
   iter_2++;
   if ((iter_2-(start_2+init)-reps_2)
         % total == 0)
     iter_2 = iter_2+total-reps_2;
}
```

| iter | iter++ | Check | Next |
|---|---|---|---|
| 4 | 5 | (5-4-2) = -1 | 5 |
| 5 | 6 | (6-4-2) = 0 | 10 |
| 10 | 11 | (11-4-2)=5 | 11 |
| 11 | 12 | (12-4-2)=6 | 16 |

Figure 3-10: Example of an iteration filter fissed into three fission products each with multiplicity 2. The chart indicates the values used to determine the next value of the iteration field.

Let $F$ be the stateless filter that is fissed. Assume the fission process yielded $N$ fissed products, namely $F_0$, $F_1$, ... , $F_i$, ... , $F_{N-1}$. The notation $F_i$ represents the $(i+1)$th fissed product of filter $F$.

The fission process now modifies the fission products by adding the following values as fields to the products:

- `init`: the multiplicity of the initialization schedule. This value is determined for $F$ and is constant for all fissed products.

- $\texttt{reps}_i$: how often the `work()` function of the product $F_i$ is invoked between rounds.

- $\texttt{start}_i$: the value of the induction variable each product $F_i$ starts with, less the initialization multiplicity. Alternatively, $\sum_{j=0}^{i-1} \texttt{reps}_j$ of all fission products preceding the current product.

- `total`: the periodic multiplicity of $F$. Alternatively $\sum_{j=0}^{N-1} \texttt{reps}_j$. This value is the same for all fissed products.

As described in §2.3 initialization execution is required for peeking filters to ensure every firing of the periodic steady-state schedule maintains the same number of left-over items on the channel. The fission transformation guarantees that all initializtion multiplicities are executed by the leftmost filter. However, all fission products must take the multiplicity of the initialization schedule into account in their calculations as the multiplicity is adjusted upwards by this value.

Accordingly, the fission product $F_i$ should start each round with iteration values:

$$\mathtt{total} * \mathtt{k} + (\mathtt{start}_i + \mathtt{init})$$

and range up to the value

$$\mathtt{total} * \mathtt{k} + (\mathtt{start}_i + \mathtt{init}) + \mathtt{reps}_i - 1$$

where k is a nonnegative integer indicating how many rounds have been run in the span of the program.

At the end of each fission product's `work()` function, a check must be made to see if it is necessary to increment the induction variable to the next round of values. This will prevent certain fissed products from making calls with duplicate iteration values. This check is done after the field incrementing statement.

$$(\mathtt{iter}_{i,k} - (\mathtt{start}_i + \mathtt{init}) - \mathtt{reps}_i)\%\mathtt{total} \quad == \quad 0$$

This is consistent with the maximum value per round as indicated above. Only when we reach this maximum value does subtracting $\mathtt{start}_i$, init, and $\mathtt{reps}_i$ from $\mathtt{iter}_i$ leave a value divisible by total.

Once the fissed product's iteration value has reached this value, it must be set to:

$$\begin{aligned}
\mathtt{iter}_{i,k+1} &= \mathtt{iter}_{i,k} + (\mathtt{total} - \mathtt{reps}_i) \\
&= \mathtt{total} * \mathtt{k} + (\mathtt{start}_i + \mathtt{init}) + \mathtt{reps}_i \\
&\quad + (\mathtt{total} - \mathtt{reps}_i) \\
&= \mathtt{total} * (\mathtt{k+1}) + (\mathtt{start}_i + \mathtt{init})
\end{aligned}$$

which is the starting iteration value of the next round, as defined.

Figure 3-10 shows the filter from Figure 3-8 fissed into three fission products, each with multiplicity 2. The accompanying charts for each fissed product are as follows:

- `iter` indicates the value of the `iter` field at the start of the filter invocation.
- `iter++` indicates the value of the `iter` field incremented by 1 (which is the value `iter` takes prior to the check.
- `check` indicates the value of $(\texttt{iter}_{i,k} - (\texttt{start}_i + \texttt{init}) - \texttt{reps}_i)$, which must be divisible by `total` in order to advance to the next round.
- `next` indicates the value `iter` will take at the next invocation of the fissed product.

Note that `iter` jumps to the next round of values with multiplicity 2, as expected.

### Accounting for Steady-state Schedule Modification

The code additions described above are dependent on specific values for the initialization and steady-state multiplicities of the fission products. If any multiplicities change, then the values used in the generated code must be updated. The initialization schedule is not modified once it is calculated, but certain passes in the StreamIt compiler may modify the steady-state schedule after fission, increasing the multiplicity of the fission products. In this section we demonstrate how passes that run after fission and seek to modify the steady-state must update the generated fields.

Increasing the steady-state scales the $\texttt{reps}_i$ field for all products. $\texttt{start}_i$ and `total` are dependent on $\texttt{reps}_i$ and must be scaled accordingly. Assume a pass increases the steady-state multiplicity of our filter to $m$. The generated fields for product $F_i$ are updated as follows:

- `init`: unchanged because the initialization schedule in unmodified when the steady-state is increased.
- $\texttt{reps}_i = \texttt{reps}_i * m$

| **Phase** | **Function** |
|---|---|
| Front-end | Parse StreamIt code and create AST. |
| SIR Conversion | Converts the AST to the StreamIt IR (SIR). |
| Graph Expansion | Expands all parameterized structures in the stream graph. |
| Scheduling | Calculates initialization and steady-state execution orderings for filter firings. |
| Partitioning | Performs fission and fusion transformations for coarsening and parallelism. |
| Layout | Assign filters of the partitioned graph to cores |
| Filter optimizations | Apply traditional optimizations such as constant propagation, loop unrolling, scalarization and batching. |
| Code generation | Generates code for synchronization, computation, and communication. |

Table 3.2: Phases of the StreamIt compiler's coarse-grained task, data, and software pipeline parallelism backend [25].

- $\texttt{start}_i = \texttt{start}_i * m$
- $\texttt{total} = \texttt{total} * m$

After we increase the multiplicities, the values of the four fields introduced by the transformation are consistent, i.e., the iteration values will be correctly redistributed to account for the modification to the steady-state multiplicities.

## 3.6 Compiler Infrastructure

We have modified the StreamIt compiler infrastructure [1] to include full support for the iteration keyword and to correctly handle the desugared filters in passes and transformation that are affected by the presense of state. This section provides an overview of the compiler and the changes we implemented.

The overall flow of the StreamIt compiler is given in Table 3.2. Begininning our discussion from the first phase, we made modifications to the parser to correctly parse the new `iter()` keyword. An expression class representing `iter()` was added to the intermediate representation. The type of this expression is always of the base integer type in the language. We added `iter()` to the IR so uses of it can be type-checked by StreamIt's type checker.

The desugaring step (see §3.4) is performed after Graph Expansion and before Scheduling. The modifications to the Fission transformation described in §3.5 are included and called by the partitioning algorithm. Since the Scheduling phase is called before partitioning, the initialization and steady-state multiplicities for each filter is available for use by the modified fission transformation.

The compiler includes a function that determines if a filter is stateful or stateless. Originally, this function looks for a filter field that may be assigned a non-constant value, and the field may also be read. If such a field exists the filter is classified as stateful. We do not want desugared iteration filters to be classified as stateful, so we modified the function to ignore the *iter* field added by the desugaring, thus classifying filters that include state that is only the generated *iter* field as stateless.

The only filter optimization that needed alteration was batching. The batching optimization increases the steady-state multiplicities of all filters to amortize the cost of synchronization at the expense of latency [25]. Since this optimization increases the steady-state multiplicities by a calculated factor $m$, it must now scale the fields of all fissed iteration filters as described in §3.5.2. Since the fields are compiler generated, it is simple to find them by name and scale them appropriately.

The StreamIt compiler infrastructure includes multiple paths and backends. For the evaluation in the next section, we employ the coarse-grained task, data, and software pipelining path as described in [25]. This approach aggressively fuses filters, careful not to introduce state and obscure data parallelism. For our modifications, desugared iteration filters without additional state are classified as stateless and fused with other stateless filters during this coarsening step.

A heuristic algorithm considers each task-parallel slice of the graph, introducing data parallelism when available such that a slice will occupy all cores of the chip. Coarse-grained software pipelining is introduced by adding appropriate buffering and a prolog schedule such that within the steady state, there are no dependences between operators. Finally, a heuristic mapping algorithm first groups data parallel filters together into sets, and then schedules the sets so that each occupies all cores. Stateful filters are greedily bin-packed to take advantage of software pipeline parallelism. The next section gives the experiment results.

## 3.7 Empirical Evaluation

In this section we evaluate the performance and scalability benefits of the `iter()` keyword and its parallelization by providing empirical results for two applications that originally included induction variable state. We modified these applications to remove the explicit induction variable state, and instead employ the `iter()` keyword. We modified the StreamIt compiler as described in the last section. The experimental architecture is composed of 4 octal-core 2.00 GHz Intel Xeon x7550 processors, each with 18 MB L3 caches. The architecture has 128 GB of available memory.

### 3.7.1 MPEG-2 Motion Estimation

This section presents an application of induction variables to the Motion Estimation compression subset of the MPEG-2 encoder. Motion estimation attempts to generate predictions with respect to a set of reference frames obtained from previous or future pictures.

The stream subgraph of the MPEG-2 encoder is illustrated in Figure 3-11 [20]. Each block of pixels will be tested against three types of prediction (no prediction, forward predicted, and backward predicted) to determine which is the best method for motion estimation. The MotionEstimationDecision filter determines which is the best encoding technique for this macroblock.

The filter MotionEstimation is stateful and contains the majority of the work. The MotionEstimation iterates through a two-dimensional array (16x16 macroblocks) along the picture and relies on induction variables to maintain its array position. We can apply the induction variable transformation on this filter to remove the state in the filter.

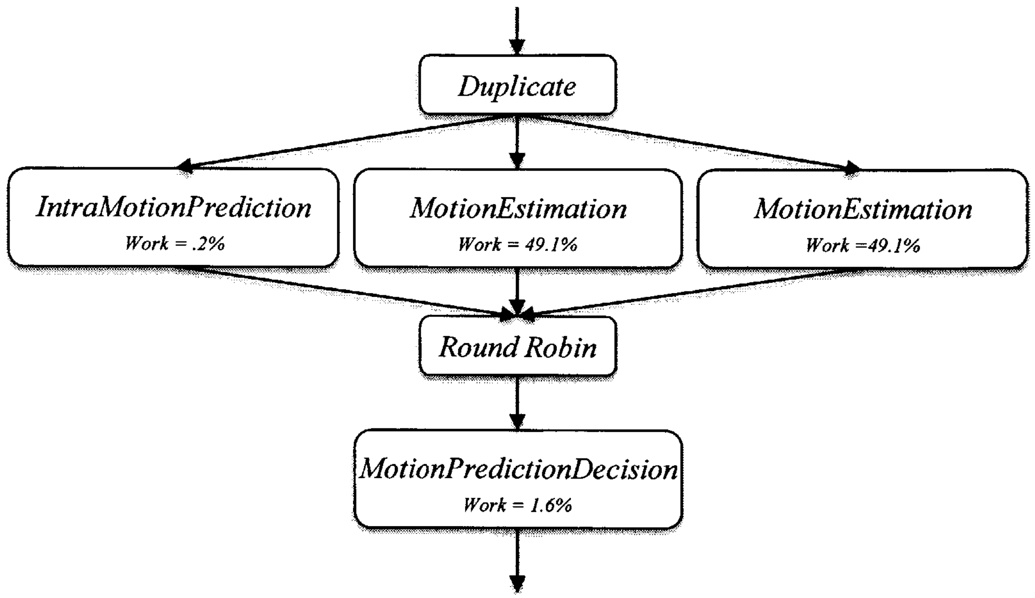


Figure 3-11: MPEG Motion Estimation stream graph.

Reference pictures can be set using upstream messaging from later in the stream graph. Currently the backend does not support the use of upstream messaging, so for the purpose of benchmarking this application, the reference picture is set to a dummy value and is unchanged throughout the program. This does not detract from the data parallelism introduced after removing the induction state. Upstream messaging would simply require that sent messages be duplicated to all fissed filters in the stream graph.

Figure 3-12 shows the speedup figures for the MPEG-2 motion estimation subset. There is a noticeable speedup for 2 cores for both induction state and iteration keyword implementations. This can be attributed to the stream graph, which is composed of two stateful MotionEstimation filters that have task parallelism. The bottleneck in data parallelism is apparent as we increase the number of cores past 2, as the majority of the work cannot be partitioned and load-balanced effectively across multiple cores.

*Theoretical speedup* is calculated using the theoretical speedup value from 3.2 and the base induction state runtime values. The model requires work to be serialized.

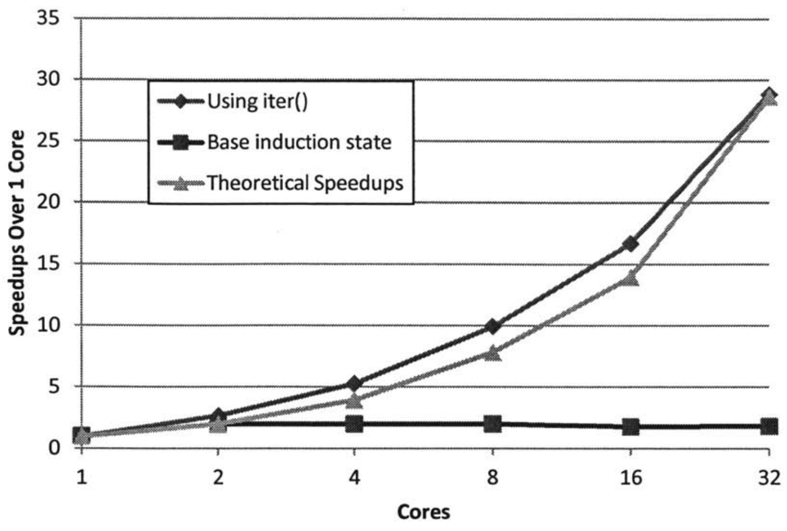


Figure 3-12: Speedups for MPEG-2 Encoder Motion Estimation subset, with and without induction variable state.

Since almost all of the work is embedded in the two filters that are task parallelizable, we can estimate the theoretical speedups by dividing the theoretical speedup value by 2. This is reflected in Figure 3-12.

We can see significant improvements to runtime after making this subset stateless and exposing data parallelism. There is a 4.93X speedup on 8 cores, 9.30X speedup on 16 cores, and 15.62X speedup on 32 cores between the base induction variable and iteration keyword implementations.

### 3.7.2 FIRBank

The FIRBank benchmark contains multiple finite impulse response filters each with a different impulse response coefficient array. This benchmark is used in speech processing applications. FIRBank contains one filter that uses induction variable state with 3.94% of program's work. This filter, Multiply, maintains induction state in an index that traces through the rows of a two dimensional array. Each invocation performs complex multiplication on the stream input values and the array elements of that specified row. The conversion to the `iter()` keyword version removed 5 lines and added 1 line.

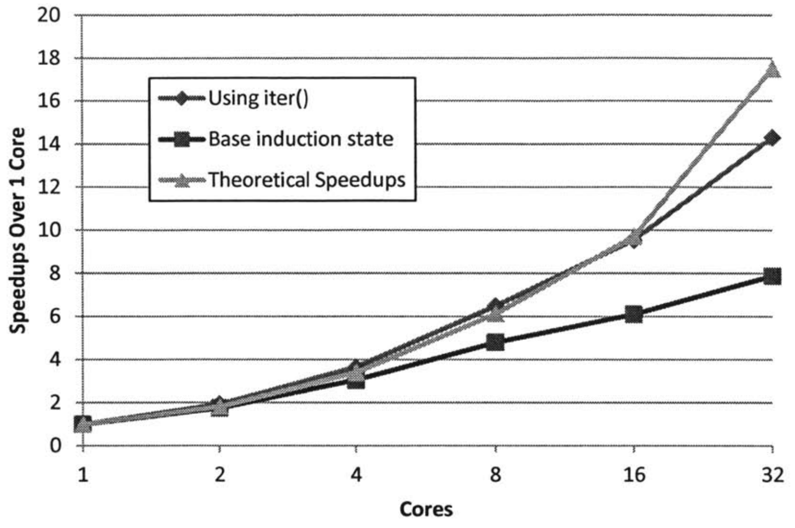


Figure 3-13: Speedups for FIRBank, with and without induction variable state.

The original version with explicit state is partitioned into a 3 filter pipeline: stateless filter, stateful filter (with explicit induction state), and a stateless filter. The stateless filters are data-parallelized, but the first requires data to be collected in a joiner (in a synchronization point) before passing on to the unparallelized stateful filter. Furthermore, output from the stateful filter must be communicated to all cores of the chip as the last stateless filter has fission products on all cores. In the `iter()` version, the entire application is fused to a single filter that can be data parallelized. There is no inter-core communication in the fused and parallelized final version.

Figure 3-13 indicates the speedups over 1 core for both induction state and iteration keyword implementations. We do not see close-to-perfect scaling in this application because I/O costs dominate the computation costs. However data parallelism is exposed. Between the two implementations, there is 1.36X speedup on 8 cores, 1.58X speedup on 16 cores, and 1.83X speedup on 32 cores for the iteration keyword implementation. This abides fairly closely with the model as described in Section 3.2.

# Chapter 4

# Locality-Sensitive Memoization in Streaming Multimedia Programs

Multimedia applications exhibit special characteristics that are well suited for memoization. Many multimedia inputs have redundancy in the underlying data. Images may have very similar blocks of pixels. Audio files may have sequences of similar sounds. Video files render similar images between frames. In fact, multimedia compression techniques exploit such redundancy in order to represent the data in much smaller output sizes.

Such applications also share the quality that the target consumer of the program output is the human senses. Human visual and audio perception presents an inherent tolerance in processing their respective medium. The inherent benefit of such image, video, and audio compression techniques is that it will save on generated output size at the loss of some computing accuracy in its processing. Such loss in accuracy would generate some degradation in output quality. However, this loss in quality is largely indistinguishable by the human senses.

The computer systems that run such multimedia applications include desktop computers, laptops, handheld devices, and mobile phones. Many of these devices are powered by battery, and accordingly must be careful to consume energy efficiently. As multimedia applications are the most dominant component of a typical system's workload, providing optimizations that will save the system from performing expensive computation will help reduce energy consumption.

Accordingly, we construct a memoization technique, applying an approximate nearest-neighbor algorithm to find memoization matches, targeted for applications that can tolerate slight losses in computation accuracy. This technique is implemented in the StreamIt programming language [48]. The StreamIt benchmark suite contains several multimedia applications for which memoization can be applied effectively, including MPEG-2 decoding [30] and MP3 decoding [29].

In this chapter, we describe classical function memoization (§4.1), locality-sensitive hashing (§4.2), our approach to filter memoization for stream programming (§4.3.2), and changes made to the runtime system for filter memoization (§4.4 & §4.5).

## 4.1 Function Memoization

Function *memoization* [40, 41] is a widely used and widely researched optimization technique. The main purpose of memoization is to potentially bypass the execution of an expensive function call that may have been previously executed. Such function calls are identified according to some input that dictates the filter's execution. These functions have the quality that subsequent executions with the same input should yield the same output.

Memoization provides access to such input-output pairs by storing them in a lookup table. This lookup table can be preloaded and can be populated throughout the program execution. Subsequent executions of the function would first query the lookup table with the appropriate inputs. If an input-output pair is found, the output can simply be returned. Accordingly, the technique makes a predictable tradeoff on space used for potential performance gained.

Memoization is a particularly useful technique for programs with inputs that are often repeated throughout a program's execution. This property is often commonly exhibited with multimedia applications. Videos processing often renders similar images between frames. Image processing often needs to render large areas of similar color data. Audio processing often have repeated sounds or large sequences of silence that requires the same function executions.

### 4.1.1 Barriers for Streaming Multimedia Applications

Unfortunately, this simple means of function memoization does not work as well for streaming multimedia applications in practice. Several barriers exist when applying function memoization to streaming multimedia programs.

Many multimedia applications represent the underlying media files using floating-point figures. Even within the same media file, some fuzziness or floating-point precision roundoff errors may cause slight deviations on the floating-point representation. As function memoization operates by matching input parameters, such fuzziness can cause inexact values, thus preventing lookup table matches. Accordingly the use of function memoization really only introduces extra overhead in space usage for the lookup table and program execution in lookup time.

In the context of the streaming space, multimedia programs often have filters that exhibit very large input and output rates. MPEG-2 processes frames a block at a time, which constitutes a set of 8x8 luminance samples. MP3 reorders and adds data across large sliding windows (over 1000 peeks required). Channel vocoder performs a sliding autocorrelation and threshold across a large set of items (by default 100 peeks required). After compiler fusion transformations are applied, such input and output rates can increase. The input spaces for such multimedia programs are generally very large, and as such can cause a drastic increase in the overhead of looking up entries.

Furthermore, we would need to be careful when choosing our target filters for memoization. It may be possible that the overhead of performing memoization queries may exceed the benefits in skipping the `work()` . The amount of work saved via performing lookups to a lookup table may not be enough to compensate for the amount of work required to perform the lookups in the first place. Applying the simple function memoization technique to stream programming for multimedia applications would likely yield little benefit, unless the inputs are deliberately selected to fit the technique.

### 4.1.2 Inexact Memoization for Streaming Applications

Many multimedia algorithms including MPEG-2 and MP3 are examples of *lossy* compression algorithms. Accordingly, corresponding applications present a level of tolerance to the resulting output data. It is possible to introduce speedups to the processing of the application at the cost of some loss in the accuracy of the resulting output. Such deviations to the output are allowed because the final target of the outputs is the human senses. While some visual or audible differences may exist in the resulting output, as long as such differences are inperceptible to the human sense, it may still be accepted by the target user.

As described in §4.1, multimedia applications often presents a level of repetitiveness in the underlying data. However, because of the barrier of inexact matches as described in §4.1.1, it may be necessary to adjust the strictness of our memoization technique in order to exploit the repetitive nature of the data in our multimedia applications.

Such streaming video or audio processing applications exhibit a nice property that a small change to the input results in a corresponding small change in the output [42]. A slight change in the color data (chrominance sample) of an MPEG-2 frame, for instance, would propagate the same slight change in the resulting output frame. Accordingly, we can relax the function memoization requirement of having an exact match to allowing an inexact but "close-enough" match. This makes it possible to exploit the tolerance of an application's target user to achieve the desired speedups via memoization.

The process of finding a "close-enough" input match is analagous to solving the *nearest neighbor* problem. This problem is defined as follows: given a set of points $P$, all of which are in an input space $S$, a requested query point $q \in S$, and a distance metric $D$, find the point $p \in P$ that minimizes $D(q, p)$.

It is straightforward to map the nearest neighbor problem to the problem of matching inputs for memoization. $P$ corresponds to the inputs of the lookup table and the query point $q$ refers to the querying input of the current filter execution. The input space $S$ refers to the input space of all incoming inputs for the target filter. For example, an `int->output` filter with a peek rate of $d$ has an input space in $\mathbb{Z}^d$, and a `float->output` filter with a peek rate of $d$ has an input space in $\mathbb{R}^d$.

The time complexity of many nearest neighbor solutions suffer from exponential scaling with the dimensionality of $S$. As described in §4.1.1, input rates are often very large for multimedia programs. This would in turn significantly scale up dimensionality of the input space and the time complexity of input matching. As memoization is a technique that imposes an ideally slight overhead for finding appropriate input matches on all function calls, it is crucial to find a nearest neighbor solution that can find an appropriate match without imposing significant performance delays.

## 4.2 Locality-Sensitive Hashing

Solutions for the nearest neighbor problem for data points in $d$-dimensional Euclidean space suffer from exponential scaling in either space or runtime. For large enough dimensions, such solutions offer little improvement to a linear search over all points in the table. Such large dimensions are very common for streaming programs, especially after fusion transformations are applied. Often times, however, applications of the nearest neighbor problem do not require the actual nearest neighbor, but rather an approximate point. The approximate nearest neighbor problem relaxes the condition of finding the exact nearest neighbor to the query point to finding a neighbor within a certain range of the query point.

*Locality-sensitive hashing* [22, 18, 5] is a randomized algorithm for solving the approximate nearest neighbor problem in the Euclidean space. Locality-sensitive hashing helps to deal with the previously described *curse of dimensionality*, along with the problem of inexact input values as described in §4, in solving the nearest neighbor problem. The algorithm relies on hash functions that map similar data inputs into the same hash bucket, with high probability. Data inputs are loaded into the hash table accordingly. Querying the hash table is a matter of using the hash functions to identify the corresponding hash bucket and iterating through the appropriate input values in the hash buckets.

### 4.2.1 Locality-Sensitive Hash Families

*Locality-sensitive hash families* [18, 5] define a set of functions that exhibit the property that data inputs close to one another in the Euclidean space will, with high probability, be mapped into the same hash bucket. A hash family is defined below for an input domain $S$, a distance metric $d$, and an approximation factor $c$.

> **Definition 1.** *A hash family $\mathcal{H} = \{h : S \longrightarrow U\}$ is called $(r, cr, p_1, p_2)$-sensitive for $d$ if for any pair of points $p, q \in S$:*
>
> - If $d(p,q) \leq r$ then $Pr_{\mathcal{H}}[h(p) = h(q)] \geq p_1$
> - If $d(p,q) > cr$ then $Pr_{\mathcal{H}}[h(p) = h(q)] \leq p_2$

The locality-sensitive hash family definition specifies that points near one another (i.e. with distance less than $r$) will be hashed to the same bucket with probability at least $p_1$. Furthermore, the probability of points that are far from one another (i.e. with distance greater than $r$) being mapped to the same hash bucket is bounded above by $p_2$. A locality-sensitive hash family defined by above is useful if $p_1 > p_2$ and $c > 1$. By this definition, inputs that are near one another have a higher probability of being hashed into the same bucket than inputs that are far from one another.

Locality-sensitive hash families exist for various distance metrics. Families for Hamming distance for binary vectors [28], the Jaccard similarity coefficient for sample sets[14, 13], and $l_s$ norms for $s \in [0, 2)$ between vectors in some vector space [18] can be applied in the locality-sensitive hashing algorithm.

### 4.2.2 Locality-Sensitive Hashing Algorithm

The general locality-sensitive hashing algorithm consists of a preprocessing step to populate the hash tables and a querying step for finding an appropriate near neighbor. The algorithm also must also take measures to ensure the data points are bucketed and accessed correctly (with high probability).

Given a hash family $\mathcal{G}$, with parameters $(r, cr, p_1, p_2)$, it is possible that the probabilities $p_1$ and $p_2$ are too close to one another. Such can hinder the performance of the system as either points not close in distance are bucketed with comparatively higher frequency, or points that are near in distance are bucketed with comparatively lower frequency. This probability gap can be increased through an amplification process. We define a new hash family $\mathcal{H}$ where each function $h$ is defined by the concatenation of $k$ functions $g_1, g_2, ..., g_k$ in the hash family $\mathcal{G}$.

This new hash family causes reduces the collision probability of objects with large distance between one another $(p_2^k)$, but also reduces the probability of nearby objects $(p_1^k)$. Accordingly, to further amplify the process of locating an appropriate approximate near neighbor, the algorithm constructs $L$ different hash tables, each defined by a hash function $h$ from the newly formed hash family $\mathcal{H}$.

The preprocessing step takes all $n$ points from the input set $S$ and hashes them into each of the $L$ different hash tables. With high probability, loaded points that are close together would be located in the same hash bucket for at least some of the $L$ hash tables.

To query for a nearest neighbor, for each hash table, calculate the appropriate hash bucket for the query point. Each point in each hash bucket is retrieved. The distances between the input point and the retrieved points are calculated. If a distance is less than the requested threshold, the corresponding retrieved point is reported as a near neighbor.

## 4.3 Applying Locality-Sensitive Hashing to Stream Filter Memoization

In applying this approximate nearest-neighbor solution to memoization in stream programming, we must define several characteristics for the target stream programs to ensure locality-sensitive hashing achieves the expected results in filter memoization.

We first define what is considered the "input" for our target memoization tables.

For the original function memoization, the function parameters act as the identifying input. The analagous components in stream programs are the values coming off the input stream, specifically the values we peek. Functionality of the filter comes from all incoming values that will be used in the processing of the filter, regardless of whether or not we pop them off the stream.

### 4.3.1 Memoizable Filters

It is also important to ensure that the target filter yields similar output values for similar input values, as follows:

**Definition 2.** A filter $F : \mathbb{R}^k \rightarrow \mathbb{R}^m$ is *memoizable* if:

Given that $F$ maps input $(p_1, p_2, ..., p_k)$ to output $(q_1, q_2, ..., q_m)$:
For each $i$ such that $1 \leq i \leq k$, there exists some small $\delta, \epsilon_1, \epsilon_2, ..., \epsilon_m \in \mathbb{R}$ such that $F$ maps input $(p_1, p_2, ..., p_i + \delta, ..., p_k)$ to output $(q_1 + \epsilon_1, q_2 + \epsilon_2, ..., q_m + \epsilon_m)$.

This characteristic of the filter ensures that small changes to the filter's inputs will in turn generate comparably small changes to the output. If at any specific input point there are significant jumps in the output, the memoization table may return very different output values for the filter, which would in turn cause large deviations in the overall stream program.

Figure 4-1 shows a `LowPassFilter` as used in the FMRadio application. This filter is an example of a good candidate for memoization. Any $\delta$ change to one of the next `taps` inputs off the input stream would simply affect the resulting `sum` value by a factor of the $\delta$ change. Accordingly, an approximate input match for the LowPassFilter (of `taps` many floats) would still generate a fairly close output `sum` value.

Figure 4-2 shows a `Detect` filter as used in the Frequency-Hopping Radio application. This filter is a trickier to memoize with the locality-sensitive technique due to the step-function nature of the `work()` function. A slight change to the first input

```
float->float filter LowPassFilter(float rate,
      float cutoff, int taps, int decimation) {
  float[taps] coeff;

  init {
    int i;
    float m = taps - 1;
    float w = 2 * pi * cutoff / rate;
    for (i = 0; i < taps; i++) {
      if (i - m/2 == 0)
        coeff[i] = w/pi;
      else
        coeff[i] = sin(w*(i-m/2)) / pi / (i-m/2) *
          (0.54 - 0.46 * cos(2*pi*i/m));
    }
  }

  work pop 1+decimation push 1 peek taps {
    float sum = 0;
    for (int i = 0; i < taps; i++) {
      sum += peek(i) * coeff[i];
    }
    push(sum);

    for (int i=0; i<decimation; i++) {
      pop();
    }
    pop();
  }
}
```

Figure 4-1: Example of a StreamIt filter that can be memoized easily.

```
float->float filter Detect(float start_freq,
    int channels, float channel_bandwidth,
    float hop_threshold, int pos) {

  work pop 1 push 2 {
    float val = pop();
    push(val);
    if (val > hop_threshold) {
        push(pos);
    } else {
        push(0);
    }
  }
}
```

Figure 4-2: Example of a StreamIt filter that cannot be memoized easily.

off the input stream may cause the resulting float `val` to be greater or less than the `hop_threshold` where it would otherwise be the opposite, causing a far more noticeable change to the resulting output. It may be possible that this deviation will be automatically corrected or accounted for further down the stream graph, thereby mitigating the noticeability of this deviation in the final output. However, the memoization technique operates on a filter-by-filter basis without knowledge of the rest of the stream graph. Accordingly, the algorithm attempts to maintain the sensible outputs for the program on the filter level.

### 4.3.2 Stable Distribution-based Hash Families

In establishing the input of our memoization tables as the values we peek off the input stream of our target filter, we must select an appropriate hash family. As we can represent the inputs as points in Euclidean space ($\mathbb{R}^d$), we can apply the locality-sensitive hashing scheme as documented in [18, 5]. The following section provides an overview of the locality-sensitive hash family as applied to $p$-stable distributions and the $l_p$ norm distance metric, in particular the Euclidean norm.

A distribution $D$ is considered $p$-stable [43] if for $n$ independent and identically distributed variables $X_1, \cdots, X_n$ all under the distribution $D$, and $n$ constants $a_1, \cdots, a_n$, the random variable $\sum_i a_i * X_i$ has the same distribution as the random variable $(\sum_i |v_i|^p)^{(1/p)} X$ for some $p > 0$ and $X$ under the distribution $D$. A commonly known stable distribution exists for $p = 2$, namely the Gaussian (normal) distribution.

The definition of $p$-stable distributions provides a nice property that allows us to deal with high dimensionality efficiently for the locality-sensitive hashing scheme. Given an input vector $\boldsymbol{v}$ of dimensions $d$, we generate a random vector $\boldsymbol{x}$ where each entry is chosen from a $p$-stable distribution. The dot product $\boldsymbol{v} \cdot \boldsymbol{x} = \sum_i v_i * x_i$ has the same distribution as the random variable $(\sum_i |v_i|^p)^{(1/p)} X$ where X is a random variable with $p$-stable distribution. This random variable is simply $\|\boldsymbol{v}\|_p X$. For $p = 2$, this norm is the Euclidean norm.

The hashing scheme uses this dot product $\boldsymbol{v} \cdot \boldsymbol{x}$ to assign a hash value for each vector $\boldsymbol{v}$. As the dot product is distributed according to $\|\boldsymbol{v}\|_p X$, it is locality sensitive. Consider such vectors, $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, that are close together, (low $\|\boldsymbol{v}_1 - \boldsymbol{v}_2\|_p$). The hash value returned for both vectors are $\boldsymbol{v}_1 \cdot \boldsymbol{x}$ and $\boldsymbol{v}_2 \cdot \boldsymbol{x}$ respectively. The difference of the hash value, $\boldsymbol{v}_1 \cdot \boldsymbol{x} - \boldsymbol{v}_2 \cdot \boldsymbol{x} = (\boldsymbol{v}_1 - \boldsymbol{v}_2) \cdot \boldsymbol{x}$, is distributed with $\|\boldsymbol{v}_1 - \boldsymbol{v}_2\|_p X$, by definition of $p$-stable distributions. Accordingly, two vectors with lower $\|\boldsymbol{v}_1 - \boldsymbol{v}_2\|_p$ would return hashes that are closer in value than two vectors with higher $\|\boldsymbol{v}_1 - \boldsymbol{v}_2\|_p$. Accordingly, rather than performing exponential searches in high-dimension input spaces, we can simply compute the Euclidean norm to find the appropriate hash bucket.

The hashing scheme we define for the locality-sensitive memoization is defined by a 2-stable distribution-based hash family (Gaussian-based). By performing the dot product $\boldsymbol{v} \cdot \boldsymbol{x}$, we are effectively performing a scalar projection of $\boldsymbol{v}$ onto the real line. This real line will be bucketed into equi-width segments of size $r$. Hash values for each vector are assigned according to the corresponding segment of the real line that the vector has projected onto.

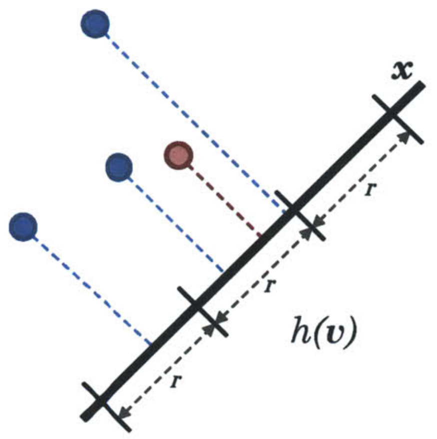


Figure 4-3: Graphical example of Locality-Sensitive Hashing for the $l_2$ hash family.

In particular, our hash function $h : \mathbb{R}^d \rightarrow \mathbb{Z}$ maps incoming inputs of dimensionality $d$ to some integer hash value. Each hash function is defined by a vector $\boldsymbol{x}$, a $d$-dimensional vector where each entry is chosen from a 2-stable (Gaussian) distribution, and a real value $b$, a real number uniformly distributed in the range $[0,\ r]$. The hash function is then defined as $h(\boldsymbol{v}) = \lfloor \frac{\boldsymbol{v}\cdot\boldsymbol{x}+b}{r} \rfloor$. The remainder of the locality-sensitive hashing algorithm follows in line with the process described in §4.2.2.

### 4.3.3 Locality-Sensitive Memoization in the StreamIt Symmetric Multiprocessor Backend

The locality-sensitive memoization scheme has been implemented in the StreamIt backend for symmetric multiprocessor (SMP) architectures. The focus of this algorithm is for single-core performance.

We assign targets for filter memoization after the completion of the fusion transformations in the partitioning phase of the StreamIt compiler as described in Table 3.2. Good targets for filter memoization are the filters that exhibit the highest amount of work, which is identified with internal work estimation calculations. The main purpose of identifying high-work filters is to amortize the cost of performing the search on the input space. The cost of calculating the hash function and performing a search through the inputs in the corresponding hash buckets is not dependent on the amount of work performed in the filter. However, more work for the target filter can be skipped if the memoization process yields an input match.

The memoization table for any particular stream filter maps an integer hash value (i.e. the corresponding hash bucket) to a linked list of input and output pairs. The input vectors, of dimensionality $d_e$ as defined by the peek rate of the filter, and output vectors, of dimensionality $d_u$ as defined by the push rate fo the filter, are all vectors generated from filter execution.

Each memoization table also maintains a unique vector analagous to the vector $\boldsymbol{x}$ in §4.3.2 of dimension $d_x$. $\boldsymbol{x}$ is generated by selecting $d_x$ many random Gaussian values. This value $d_x$ may not necessarily be equal to $d_e$, but will always be at most $d_e$. For performance improvements, we implement a means of sampling only a portion of the inputs off of the input stream. This effective downsampling would require the hash function to perform less calculations while still preserving the locality-sensitive property of the hash function (closer points will still return similar hash values, though the probability for farther points to return similar hash values increases). If downsampling is active in the target filter, the table maintains an additional vector $\boldsymbol{s}$ indicating the indices of the input vectors that we choose to sample.

Each memoization table also maintains a segment size $r$ and the random value $b$ in the range $[0, r)$. This segment size $r$ is largely input dependent and is determined on loading input and output pairs into the table.

Lastly, as we are effectively attempting to solve the *approximate* nearest neighbor problem, we maintain a maximum distance value $m$. This value indicates that for a query input vector $v_i$ and a proposed memoized input vector match $v_j$, it must be true that $D(v_i, v_j) \leq m$, where $D$ defines the Euclidean distance function. This requirement ensures that the proposed match is indeed "close enough" to our query input vector. As a result, the final output vector of the filter execution should be "close enough" as well.

The hash function corresponding to each memoization table is defined similarly to what is specified in §4.3.2. The function takes an input vector of dimensions $d_e$. If the target filter has downsampling active, only the corresponding entries of the input vector (as specified by $\boldsymbol{s}$), are used in the dot product with $\boldsymbol{x}$. The final hash value is specified by:

$$h(\boldsymbol{v}) = \lfloor \frac{\boldsymbol{v}_{samp} \cdot \boldsymbol{x}_{samp} + b}{r} \rfloor$$

## 4.4 Locality-Sensitive Hashing in Stream Filter Execution

As described in §4.3.3, target filters are chosen during compile time using approximate work estimate figures. For filters with large enough work estimates, providing effective amortization of the costs of performing memoization lookups, memoization capabilities are attached to the target filter. This process is performed entirely during compilation. The entirety of the memoization preprocessing and querying is performed during runtime execution of the program. The following section describes the process of preprocessing, loading, and querying the memoization tables during runtime execution.

### 4.4.1 Memoization Profiling Execution

The memoization tables must be populated with meaningful data in order to be effective in execution. As we set out to exploit the repetitive nature of the multimedia data, we choose to profile the program on execution of the program itself.

The user can choose to execute the profiling step of the program for a set number of iterations. In this process, each filter iteration maintains an array of inputs and outputs. Prior to the target filter's `work()` function, inputs off of the input stream are peeked and stored in the input array. Each subsequent `push()` records a single output value, which is stored in the output array. At the completion of the `work()` function, the input and output arrays are added to a profiling set of input-output pairs. At the completion of the profiling execution, all stored input-output pairs are written to disk, for future steady-state execution.

The main purpose of maintaining a separate profiling execution step is to set the required parameters in our memoization tables more effectively. Parameters such as the segment size, $r$, must be adjusted according to the maginitudes of the incoming inputs in order for the memoization tables to be both effective and efficient. Statically establishing $r$ to too large a value may yield too many inputs in a single bucket, thus slowing down the search for a proposed match. Setting $r$ to too small a value may yield too little inputs per bucket, thus bypassing potential matches that would otherwise have been accepted in execution. Furthermore, the segment size $r$ cannot be altered in the duration of the steady-state execution as all inputs are indexed according to the defined hash function, which is in turn defined according to $r$. Altering $r$ in the middle of execution would require reindexing the entire memoization table.

### 4.4.2 Memoization in Steady-State Execution

The profiling execution step must be performed prior to the steady-state execution. Prior to both the initialization and steady-state schedule of the stream program, the preprocessed caches on disk are loaded onto the memoization tables. If such cache files are not found in the expected location, memoization capabilities will be made inactive for the remainder of the steady-state schedule.

The memoization tables are initialized as described in §4.3.3. The unique vectors $x$ and $s$ are generated prior to loading. $r$ is loaded according to the value indicated in the cache file. These values must be assigned prior to loading inputs as the hash value for each input vector is dependent on what they are set to. Adjusting these values would require expensive reindexing operations.

$m$ is originally set to a best-guess static value according to the general magnitude and dimensions of the inputs. This value has no effect on how values are loaded into the table, as it is only used during runtime. It will be automatically adjusted for a desired quality level during steady-state execution.

Inputs and corresponding outputs are read off of the cache files on disk and loaded to the corresponding memoization table. Currently, we only use one table for each target filter, bypassing the amplification process used in locality-sensitive hashing.

There may be potential benefits for speedups in using multiple tables for one filter at the cost of additional memory usage, but in applying this technique to the StreamIt benchmark suite, we find an appropriately tuned $p_2$-stable distribution hash family very accurately assigns hash buckets for our target inputs. Loading is a matter of computing the hash value $\lfloor \frac{\boldsymbol{v}_{samp} \cdot \boldsymbol{x}_{samp} + b}{r} \rfloor$, where $\boldsymbol{v}_{samp}$ corresponds to the downsampled input vector. The input and output pair are added as an entry to the linked list corresponding to this hash value.

During steady-state execution, querying the memoization tables requires first acquiring all entries from the input tape. Prior to any filter execution, the entire input vector (all inputs the filters peek off of the input tape) is acquired. We calculate the target hash value $\lfloor \frac{\boldsymbol{v}_{samp} \cdot \boldsymbol{x}_{samp} + b}{r} \rfloor$, where $\boldsymbol{v}_{samp}$ corresponds to the downsampled input vector. We then iterate over the corresponding linked list of input-output pairs. For each input-output pair we calculate the $l_2$ distance (i.e. Euclidean distance) between the actual peeked input and the proposed input match. This distance value is compared with $m$ to determine if the two input vectors are "close-enough" matches.

If the distance value is less than or equal to $m$, we consider this a "close-enough" match and in turn we can bypass any work performed in the filter. This still requires the appropriate number of `pop()` calls to clear the input tape for the next execution. Lastly, we `push()` each entry of the matched output vector.

If the distance value is greater than $m$, we cannot consider this a "close-enough" match. The next input-output pair entry in the linked list will be checked. If none of the values in this hash bucket can be considered "close-enough", there are several options to we can do to proceed. If there are multiple memoization tables, we can repeat the entire process for this entry with other memoization tables under a different random vector $\boldsymbol{x}$. If there are no potential matches in any memoization table, we must identify this input vector as a cache miss. This would require the filter to be executed. If we choose to allow the memoization tables to learn during steady-state execution (not set by default), the target input-output pair can be added similar to how the loading step is performed.

## 4.5 Adjusting Query Sensitivity for Multimedia Quality and Steady-State Performance

Function memoization is primarily used as a performance optimization. As the technique provides exact results for exact matches, there is little need to consider how the optimization affects the overall output of the program. The locality-sensitive hashing approach to filter memoization provides inexact results that must be "close enough" to the actual output. The notion of two outputs being "close enough", however, must be quantified in some manner to allow the runtime system to adjust the memoization calculations if required.

### 4.5.1 Measuring Quality with the Signal-to-Noise Ratio

A widely-used metric for quantifying the quality of several multimedia processing methods, including video and audio compression programs, is the *Signal-to-Noise Ratio* (SNR). This ratio attempts to measure the level of the desired signal to the level of the background noise.

For the context of the memoization optimization, we attempt to identify the amount of noise introduced by the inexact nature of the filter memoization as compared to the expected execution of the program without the use of memoization. Though the SNR works best with a pure desired signal, this optimization technique attempts to generate an output that matches the expected output as close as possible. Accordingly our desired signal will be the expected output of the program (without memoization enabled). We can thus calculate the SNR as follows:

$$\begin{aligned}
SNR &= \frac{\text{Power of the Desired Signal}}{\text{Power of Corrupting Noise in Output Signal}} \\
&= \frac{RMS(\text{Samples of the Desired Signal})^2}{RMS(\text{Deviations between Desired and Output Signal})^2} \\
&= \frac{\frac{1}{n}\sum {S_{d,i}}^2}{\frac{1}{n}\sum(S_{d,i}^2 - S_{o,i}^2)} = \frac{\sum {S_{d,i}}^2}{\sum(S_{d,i}^2 - S_{o,i}^2)}
\end{aligned}$$

SNR is usually represented in logarithmic decibel format to manage the large ranges in values that can be inputted, defined as follows:

$$SNR_{db} = 10\log_{10}\left(\frac{\text{Power of the Desired Signal}}{\text{Power of Corrupting Noise in Output Signal}}\right)$$

$$SNR_{db} = 10\log_{10}\left(\frac{\sum S_{d,i}{}^2}{\sum(S_{d,i}^2 - S_{o,i}^2)}\right)$$

Good SNRs vary from application to application. Generally, however, having more signal compared to background corrupting noise is better for the output quality. As we are calculating the SNR of our modified output against the actual generated output, a high SNR would mean our modified output is indistinguishable from the actual output.

### 4.5.2 Effects of Various Memoization Table Parameters

§4.3.3 enumerates the parameters that define each memoization table. As every input-output pair is indexed according to these parameters, what they are set to has a very noticeable impact on performance or quality of our steady-state execution.

#### Effects of Segment Size: $r$

Perhaps the most important parameter in the context of the locality-sensitive hashing scheme is the segment size $r$, which largely determines what hash buckets each input vector is hashed to. We briefly described the consequences of not having a tuned $r$ in §4.4.1. If $r$ is set too large, graphically, we have larger segments, thus allowing more inputs to be projected onto this segment. Accordingly, there are more inputs in this hash bucket. Our program execution, as described in §4.4.2, requires a distance function to be calculated on each input in the hash bucket. Too many inputs would thus require many more distance functions to be executed. On the other hand, if $r$ is set too small, we would have smaller segments and less inputs would be projected onto the same segment, even if two inputs are close enough to be considered "close-enough"

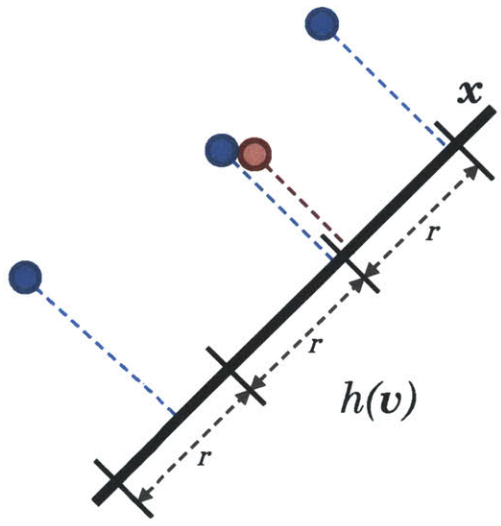


Figure 4-4: Boundary elements can potentially yield hits with matches that are not as well matched as with other elements.

matches. With too many memoization table misses, even though such good matches exist, we are incurring the overhead of memoization lookups while still performing the filter `work()` . Accordingly, the value of $r$ has a large impact on performance of the memoization technique. A happy medium for this parameter would allow for a well-distributed hash table (not too densely or sparsely packed for each hash bucket.) [18].

The value of $r$ has minimal impact on potential quality figures. Quality figures can be slightly affected if a target input is at the edge of a segment, as in Figure 4-4. Accordingly, inputs that are good matches can actually be hashed to different hash buckets. Generally, however, $r$ only sets the boundaries of the hash buckets, and does not have a large impact on if a pair of inputs can be considered good matches.

**Effects of Maximum Vector Distance Thresholds:** $m$

Perhaps the most important parameter in altering output quality is the maximum distance threshold $m$. Ultimately, this value determines if we find a good match for a target input vector. The distance between two input points are considered good matches only if the distance between the two does not exceed $m$. The program execution, as described in §4.4.2, performs the distance value comparisons as the final

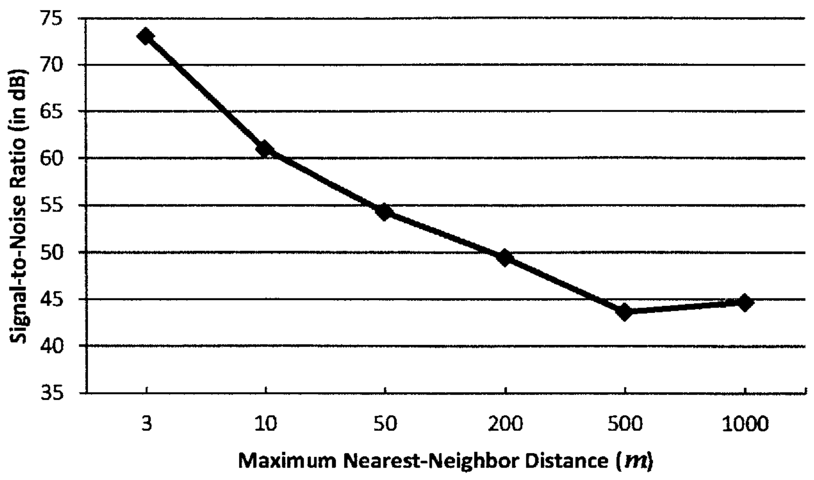


Figure 4-5: Quality comparisons between choices of $m$ for MPEG-2 using locality-sensitive hashing memoization.

step for querying, thus any changes at any step of the program would still meet a certain minimal quality in the final output.

Figure 4-5 shows a comparison between obtained SNR values with the (squared) maximum nearest-neighbor distance. The target memoized `int`→`int` filter has I/O of 64→64 and stores luminance samples for pixels. Quality very clearly degrades as $m$ increases, due to many more input vectors stored in the memoization table being labeled as good matches. Individual MPEG-2 frames have also been generated to show quality degradation visually. Figure A-1 shows the base generated output, with no memoization. With small $m$ distances, as seen in Figures A-4 and A-5, we can see very little deviation from Figure A-1, as is indicated by its high SNR with Figure A-1 as the source output. Figures A-2 and A-3 show distinct deviations in the image generated from poor memoization matches. You can also observe that SNR levels off for $m = 500$ and $m = 1000$. This is likely because the majority of vectors that are considered near neighbors for $m = 1000$ are also near neighbors for $m = 500$, as we only have 64 integers as our input vector.

### Effects of Downsampling

Downsampling, as described in §4.3.3, allows us to only consider a dimensional subset of our inputs when finding the appropriate hash bucket. This factor is largely for performance consideration. Downsampling, in general, can save us from having to calculate too many operations for the required dot-product, which is helpful for large enough dimensionality. However, downsampling can ultimately increase the size of our hash buckets, filling them with inputs that are generally not close at all with the target input.

In our current implementation, downsampling is hand-tuned to the application. The level of downsampling is difficult to tune during runtime, as it directly affects the hash functions. Reassigning the level of downsampling in the middle of execution would require reindexing the entire memoization table.

In general, very high input filters suffer from having to perform many operations to calculate the hash. Certain filters in MP3 decoder have peek rates of greater than 1000. Such filters benefit greatly from downsampling its peeked inputs, as it restricts the buckets without reporting too many false positives.

## 4.5.3 Automatic Tuning for Quality and Performance

As it is difficult to anticipate all incoming input values during compile time, extra runtime analysis is required to finely tune the memoization technique for better performance or quality. The following section presents some additional runtime tuning techniques to achieve better runtime performance or output quality.

### Performance Tuning

As the process of querying the target memoization tables require the overhead of calculating the input's hash and the distance between potential input matches, performance can potentially be worse if the target filter's memoization table does not generate enough hits. This can be a result of the program input, which may not be very repetitive in its underlying values. It may also be the case that the profiling step of the target filter did not generate enough inputs to populate the memoization table. As a result, the work of the filter must be performed in addition to the overhead of performing the lookup query.

To improve runtime performance, we would have to identify which filters can actually benefit from the memoization query. Each filter will thus have its work and hit rates monitored. If the target filter's hit rate is too small, the memoization capabilities would simply be disabled. The ideal hit rate can be generated based on estimates of the average time it takes to complete the target filter's work and the average time it takes to complete the memoization query as follows:

$$R_{work} = R_{query} + (1 - h_{min}) * R_{work}$$

$$h_{min} = \frac{R_{query}}{R_{work}}$$

We would like the filter to perform at least as well using memoization as it would without memoization. $h_{min}$ indicates the break-even hit rate, the point at which the filter using memoization performs, on average, at the same speed as the filter without memoization. The runtimes of both the memoization querying segment and the entire work function can be monitored in aggregate to generate the break-even hit rate. Filters that do not exhibit hits up to this rate will be disabled.

### Quality Tuning

Quality control is largely determined by the value that we set $m$, the maximum vector distance threshold, as described in §4.5.2. This value must be tuned according to the incoming inputs, as it is difficult to ascertain an appropriate value at compile time for a desired level of quality.

We discussed in §4.5.1 how to measure the quality of our memoized outputs. Calculating the SNR requires access to both the desired signal and our target output signal. Accordingly, to tune the memoization technique appropriately, we would have perform both the memoization query and the filter work to generate the target output signal and the desired signal respectively. This presents a direct slowdown in the steady-state execution because much more work may be performed, even if it is not needed.

We must choose the appropriate times to calculate both the desired output and the memoized output. We can perform initial tuning for some fixed amount of iterations at the beginning of the steady-state. During these iterations, we calculate both the output of the filter and the output of our memoization table. We compute the components of the SNR, namely the running squared sum of the desired signal samples and the running squared sum of the differential between the desired signal sample and the output signal sample, in aggregate between iterations. Per iteration, we will be able to generate an estimate of the SNR we will generate for the specified $m$. If it does not exceed the desired SNR threshold, the quality of the output is potentially lower, and we will scale the distance threshold downwards accordingly. The SNR components must be cleared for a new distance threshold $m$.

It may be possible that the beginning of the target input varies greatly from the rest of the input. For instance, an image may be very repetitive in the beginning set of pixels (perhaps it is a flat color background), but the rest of the image may be rich with other details. The autotuning may thus set $m$ to a very high value because the desired signal and memoized output signal are very similar for a repetitive set of inputs. Accordingly, we may also choose to perform randomized sampling during steady-state execution to determine if there is a need to lower the target distance threshold. These random iterations would follow much of the same process in calculating the SNR as described before.

## 4.6 Empirical Evaluation

In this section, we evaluate the performance benefits of the locality-sensitive memoization technique on several multimedia applications, including MPEG-2, MP3, and Channel Vocoder. The performance benefits come with potential quality degradation, and such is documented in our evaluation as well. Modifications are made to the StreamIt compiler to identify targetable filters using static work estimations, as described in §4.3.3. All remaining memoization work is performed at runtime of the program execution. The experimental architecture is composed of 4 octal-core 2.00 GHz Intel Xeon x7550 processors, each with 18 MB L3 caches, and 128 GB of random access memory. We focus our attention on single-core performance and thus work on solely one of the available cores, though the technique is applicable across all cores in parallel.

### 4.6.1 MPEG-2 Decoder Subset

The MPEG-2 decoder can interpret and decompress an MPEG-2 compliant bit stream based on the MPEG-2 specification as presented in [30]. The decoder subset in the StreamIt benchmark suite consists of a block decoder and a motion vector decoder. It omits the bit parser, which parses the incoming bitstream into the blocks data of a macroblock and the differentially coded motion vectors of the same macroblock. Instead, incoming data is hand-parsed and fed into a splitjoin that sends the blocks data to the block decoder, and the differentially coded motion vectors to the motion vector decoder. It also omits the motion compensation subset, which recovers macroblocks that were encoded via MPEG-2 motion prediction after block and motion vector decoding is complete.

This block decoder subset of the MPEG-2 decoder is entirely stateless. The motion vector decoder is stateful, which prevents the entire application to be fused down to a single filter. The block decoder branch of the decoding splitjoin, however, is fused down entirely. Static work estimations place 99% of the work in this decoder subset in the block decoder, making it an ideal candidate for memoization. The resulting fused filter is an `int`→`int` filter with I/O of 64→64.

We run the decoder with the additional runtime parameters tuning active during steady-state execution, as described in §4.5.3, on a set of 352x240 frames, which would take 1320 iterations of this filter to render each frame. The program was given a target SNR of 60 dB, which would make the resulting memoization-generated image return relatively indistinguishable from the original image [23]. At the completion of an

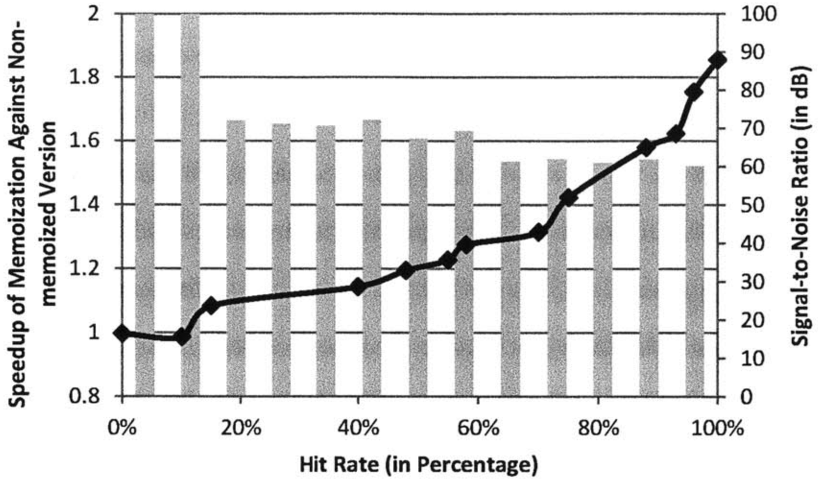


Figure 4-6: Speedups (line graph) and quality (bar graph) comparisons for MPEG-2 decoder using locality-sensitive hashing memoization.

initial 100 iterations the maximum vector distance $m$ varied from 3 to 10, indicating two vectors can be considered matches if their vector distance is at most $m$. It was also found that the ideal hit rate of the filter was 12.7%, calculated as specified in §4.5.3. Accordingly, all program executions with hit rates that ever fall below 12.7% after the first 100 iterations would immediately disable memoization capabilities.

Figure 4-6 shows the speedup and SNR figures for the MPEG-2 block and motion vector decoders subset, with the line and plot indicating performance speedup and the bars indicating SNR figures for the corresponding hit rate. Performance speedups are fairly modest, particularly as most applications will rarely generate high hit rates. There are 1.14X speedups for 40% hit rates, 1.22X speedups for 55% hit rates, and 1.27X speedups for 60% hit rates. We do observe, however, a fairly flat tail from 0% to 15% hit rates. This is generated from having memoization capabilities disabled if we ever go below a 12.7% hit rate, as described before.

As expected, as hit rates increase, generated by populating the profiled cache with more entries, quality generally degrades. More memoization matches indicates the result of our filter execution will not be exact, thus always introducing some nonnegative deviation to our final output. Tuned rendered frames can be seen in Figures A-4 and A-5. Comparing these frames to the source in Figure A-1, they are virtually indistinguishable.

### 4.6.2 MP3 Decoder Subset

The MP3 decoder performs audio decoding of files specified by the MPEG 1/2 Layer 3 audio encoding format [29]. The MP3 decoder in the StreamIt benchmark is a subset of the decoder that performs antialiasing, an inverse DCT, and PCM synthesis. The subset omits the decompression stage, consisting of Huffman decoding and re-quantization. The decoder subset contains a splitjoin of two banks of filters, performing the previously mentioned steps, for input audio files with two channels.

The MP3 decoder stream graph cannot be fused down to a single individual filter. The antialiasing and inverse DCT steps are fused into a single `float`→`float` filter, $f_1$, with I/O `609`→`1152`. Segments of the PCM synthesis step are broken into two single fused filters one `float`→`float` filter, $f_2$, with I/O `2304`→`1152` and one `float`→`int` filter, $f_3$, with I/O `1024`→`32`. By far most of the work performed in the MP3 decoder subset is performed in the PCM synthesis step, particularly the first fused filter which contains a matrix multiplication filter.

We again run the decoder with the additional runtime parameters tuning active during steady-state execution. We run the decoder on a decompressed sound input that would take about 600 iterations to complete. Through experimentation, we determined that, whilst employing the memoization technique, the RAW output audio files have much different SNR thresholds when being considered indistinguishable. An SNR of 80 dB generated fair sounding but slightly distorted samples. An SNR of 100 dB, however, generated very similar sounds to the non-memoized output. Accordingly, with a target SNR of 100 dB, we generated maximum vector distance values of $m \in (1e-5, 5e-5)$ for $f_1$ and $m \in (1e-3, 5e-3)$ for $f_2$. Ideal hit rates were determined to be around 5% for $f_1$, 0.9% for $f_2$, and 66% for $f_3$. Because of the high hit rate requirement for $f_3$, it was difficult to ascertain an accurate $m$ value for it, as it would get disabled very quickly.

Figure 4-7 shows the speedup and SNR figures for the MP3 decoders subset, with the line and plot indicating performance speedup and the bars indicating SNR figures

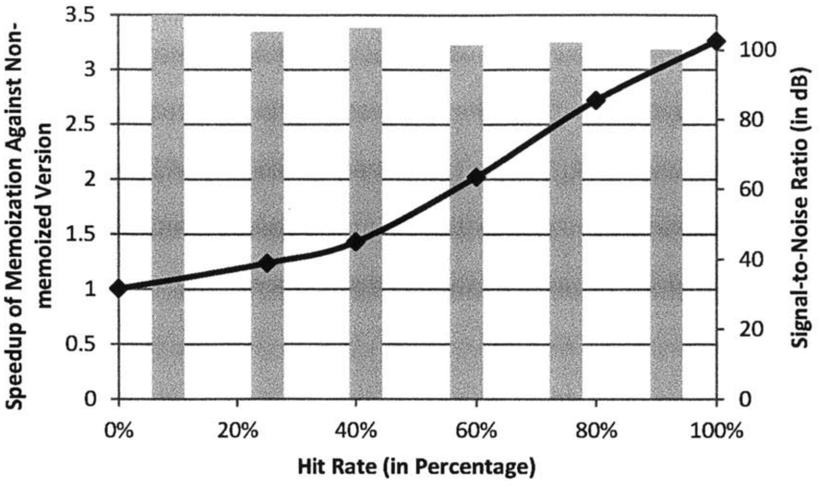


Figure 4-7: Speedups (line graph) and quality (bar graph) comparisons for MP3 decoder using locality-sensitive hashing memoization.

for the corresponding hit rate. Performance speedups are again somewhat modest, with 1.42X speedups for 40% hit rates and 2.02X speedups for 60% hit rates. The input audio file contained many silent periods, as such hit rates between 0% and 30% are hard to generate ("silence" can be very often memoized). Quality figures are also fairly level, but generally trend downward with higher hit rate, as is expected behavior.

# Chapter 5

# Related Work

Data parallelism is an important class of parallelism that many stream programming languages attempt to expose and manage automatically, including Brook [15], StreamC/KernelC [32], SPUR [51], and Cg [37]. Some streaming languages go so far as to disallow stateful computation in all filters (e.g., Brook and StreamC/KernelC) so all filters can be data parallelized. StreamIt's philosophy is to allow state for better expressiveness, but to include language idioms that capture common forms of state. Peeking is an example of representing a common form of state (in the form of a sliding window) in the language and including compiler transformations for its parallelization. Our work adheres to this philosophy of capturing common patterns of state in the language.

The Brook language includes an `indexof` expression that returns the position of the current element within an input or output stream [15]. Given that Brook kernels are executed once per input item index (across all input streams), `indexof` returns the execution iteration of the kernel. In StreamIt, a filter can potentially consume multiple input items, so the simple `indexof` operator does not give the iteration of the filter. Furthermore, StreamIt filters can consume a varying and dynamic number of items per firing. Our `iter()` keyword will correctly count the iterations of a filter with a dynamic input rate. Finally, a study of the importance and prevalence of Brook's `indexof` idiom has not been published. This work motivates the `indexof` keyword in Brook as well as our `iter()` keyword for StreamIt.

The process of eliminating traditional induction variables has been extensively researched [9, 45]. Eliminating derived induction variables in traditional loop structures further motivates the effectiveness of running the loop iterations in parallel. Extensive research has been done in the field of symbolic dependence analysis to determine if iterations in loop structures can be parallelized [38, 11]. These particular optimizations help eliminate instructions by redefining all induction variables in terms of a single induction variable. Induction variable elimination and redefinition has been used largely to remove inter-loop dependencies. Many automatically parallelizing compiler systems, including Rice Fortran D [26], SUIF [49], and Polaris [12] have also implemented recognizers to automatically eliminate such traditional derived induction variables. Such works motivate the transformations that must be performed to redefine induction variables in terms of `iter()`.

Function memoization as an optimization has been discussed for many years and has been extensively researched [2, 40, 41]. In particular memoization has been used extensively to aid in top-down parsing of context-free grammars [44, 31], helping to reduce exponential time and space requirements. This is achieved by maintaining the results of previously calculated inputs, thus preventing repeated descents in the grammar. While such a technique generally requires the explicit maintenance of the lookup-table, automatic memoization has also been researched [44, 39], particularly in the context of functional languages, where functions are first-class objects. Such an approach abstracts the details of memoization away from the programmer, which fits in line with StreamIt principles as discussed in §2. Automatic memoization, however, generally makes no decision regarding which functions to memoize, leaving that wholly to the programmer to decide. Our approach to applying memoization relies on specific characteristics of multimedia applications and static work estimations that allows us to decide whether target filters are good candidates for memoization.

Memoization, as used for functions, provide exact outputs for exact input matches. Our approach to memoization relies on the concept of returning inexact matches for memoizations that are "good-enough" to the end-user. Finding approximations in the place of exact calculations has been researched extensively in the form of soft computing [50]. The approach of applying approximations for multimedia applications is reliant on the fact that the end-user is tolerant to inexact calculation. Li and Yeung [36] investigate the application-level resilience of various programs, including several multimedia benchmarks, to low-level execution errors, finding that many such programs exhibit application-level resiliency to lower-level faults. These accuracy tradeoffs for performance are featured in such programming frameworks as Petabricks [6, 7] and Green [10]. Such programming frameworks allow users to provide controlled approximations for their target programs. Fuzzy, or tolerant, memoization [4] performs a floating point approximation by dropping $N$ least significant bits of incoming operands before performing instruction-level memoization via a look-up table.

# Chapter 6

# Conclusion

Applications in the multimedia space represent a dominant part of the workload of many computing systems in everyday use. Finding effective means of exploiting characteristics specific to multimedia applications can provide effective scaling and speedups for widely used applications. We identify several characteristics regarding multimedia applications that we set out to exploit.

Multimedia applications often requires maintaining induction variable state in its processing, be it maintaining indices of an image or frame numbers of a video. This type of state, if not parallelized, represents a significant bottleneck to scalable parallelization. We introduce a keyword expression that can be used as the basis of all derived induction variable state. The keyword, when executed, returns the current execution iteration of a filter. We present a desugaring of the keyword that enables minimal changes to the existing StreamIt compiler codebase. We present modifications to the fundamental filter fission parallelization transformation that extracts data parallelism from filters that included the new keyword. Our techniques demonstrate a substantial performance improvement on the MPEG2 motion estimation subset when explicit iteration state is replaced by our keyword solution.

Consumers of multimedia application outputs are generally tolerant to inaccuracies in the final output. Furthermore, incoming data in many multimedia programs are very repetitive. We present a means of providing approximations to multimedia outputs using previously executed inputs of the same data source. We present an application of a nearest neighbor solution to approximating filter `work()` executions. Using the locality-sensitive hashing algorithm for Euclidean space, we can find effective matches for our target inputs without being susceptible to the curse of dimensionality. We present automatic runtime tuning of the memoization to achieve the desired output quality and performance improvements based on the incoming inputs. Our technique exhibits performance improvements for a single core on the MPEG2 decoder subset and the MP3 decoder subset while generating indistinguishable quality outputs.

Streaming languages are natural tools for representing multimedia applications. Input data arrives in continuous sequence and specified processing steps can be effectively mapped to filters. Stream programming provides flexibility in representing and implementing multimedia applications. However, unlike other languages that impose steep tradeoffs between flexibility and performance, StreamIt also provides performance and scalability. With StreamIt, programmers need not focus on the low-level details of domain-specific optimizations or parallelization, as it is automated by the compiler system. With the additional optimizations of induction variable state elimination and tolerant filter memoization, we eliminate some barriers to scalability and improve performance of multimedia applications across varying computer architectures.

# Appendix A

# MPEG-2 Rendered Frames

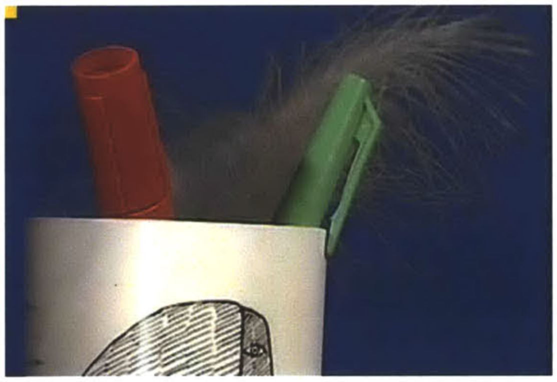


Figure A-1: An MPEG-2 rendered frame using no memoization.

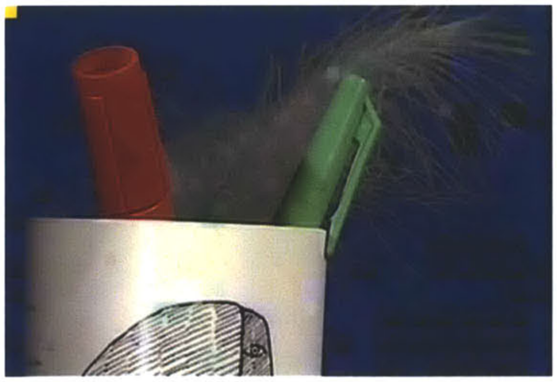


Figure A-2: An MPEG-2 rendered frame using memoization with untuned $m$ distance ($m = 500$). SNR=40db from the source Figure A-1.

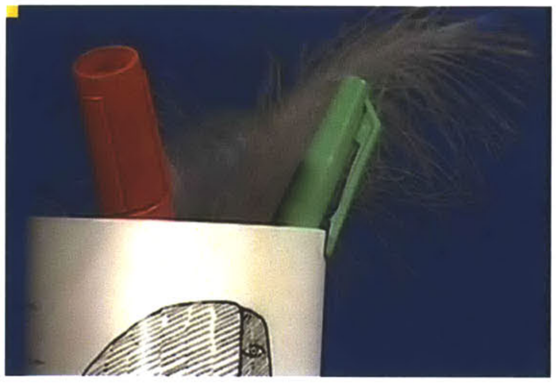


Figure A-3: An MPEG-2 rendered frame using memoization with untuned $m$ distance ($m = 1000$). SNR=45db from the source Figure A-1.

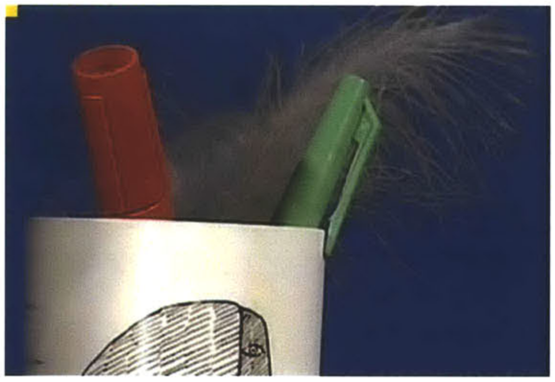


Figure A-4: An MPEG-2 rendered frame using memoization with tuned $m$ distance ($m = 3$). SNR=73db from the source Figure A-1.

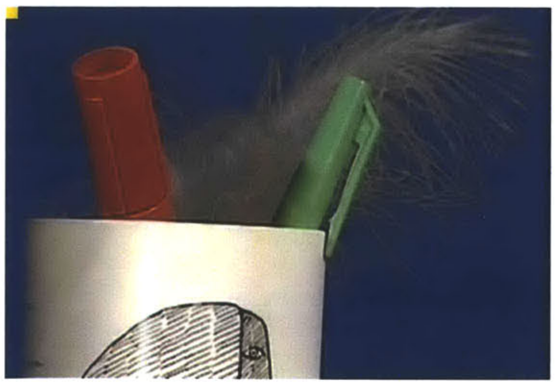


Figure A-5: An MPEG-2 rendered frame using memoization with tuned $m$ distance ($m = 10$). SNR=61db from the source Figure A-1.

# Bibliography

[1] StreamIt. `http://groups.csail.mit.edu/cag/streamit/`.

[2] H. Abelson and G. Sussman. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, 1985.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] Carlos Alvarez, Jesus Corbal, and Mateo Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computers*, 54:922–927, 2005.

[5] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, January 2008.

[6] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

[7] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 85–96, Washington, DC, USA, 2011. IEEE Computer Society.

[8] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: a java-compatible and synthesizable language for heterogeneous architectures. In *OOPSLA '10*, New York, NY, USA, 2010.

[9] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[10] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[11] W. Blume et al. Automatic detection of parallelism: A grand challenge for high performance computing. *Parallel Distributed Technology: Systems Applications, IEEE*, 2(3):37, autumn/fall 1994.

[12] W. Blume et al. Parallel programming with polaris. *Computer*, 29(12):78–82, December 1996.

[13] Andrei Z. Broder. On the resemblance and containment of documents. In *In Compression and Complexity of Sequences (SEQUENCES97*, pages 21–29. IEEE Computer Society, 1997.

[14] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *Selected papers from the sixth international conference on World Wide Web*, pages 1157–1166, Essex, UK, 1997. Elsevier Science Publishers Ltd.

[15] Ian Buck et al. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH*, 2004.

[16] C. Consel, H. Hamdi, L. Rveillre, L. Singaravelu, H. Yu, and C. Pu. Spidle: A DSL Approach to Specifying Streaming Applications. In *Conf. on Generative Prog. and Component Engineering*, 2003.

[17] Thomas M. Conte, Pradeep K. Dubey, Matthew D. Jennings, Ruby B. Lee, Alex Peleg, Salliah Rathnam, Mike Schlansker, Peter Song, and Andrew Wolfe. Challenges to combining general-purpose and multimedia processors. *Computer*, 30(12):33–37, December 1997.

[18] Mayur Datar and Piotr Indyk. Locality-sensitive hashing scheme based on p-stable distributions. In *In SCG 04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM Press, 2004.

[19] Keith Diefendorff and Pradeep K. Dubey. How multimedia workloads will change processor design. *Computer*, 30(9):43–45, September 1997.

[20] Matthew Drake. Stream programming for image and video compression. M.Eng. Thesis, MIT, 2006.

[21] Matthew Drake, Henry Hoffman, Rodric Rabbah, and Saman Amarasinghe. MPEG-2 Decoding in a Stream Programming Language. In *IPDPS*, Rhodes Island, Greece, April 2006.

[22] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[23] E.B. Goldstein. *Sensation and perception.* Brooks/Cole, 4 edition, 1996.

[24] Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *ASPLOS*, 2002.

[25] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, pipeline parallelism in stream programs. In *ASPLOS*, 2006.

[26] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling fortran d for mimd distributed-memory machines. *Commun. ACM*, 35(8):66–80, August 1992.

[27] Martin Hirzel et al. Spl stream processing language specification. Technical report, IBM, 2009.

[28] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[29] ISO/IEC 13818-3: Generic Coding of Moving Pictures and Associated Audio Information, Part 3: Audio, 1994. International Organization for Standardization, 1999.

[30] ISO/IEC 13818: Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s. International Organization for Standardization, 1999.

[31] Mark Johnson. Memoization in top-down parsing. *Comput. Linguist.*, 21(3):405–417, September 1995.

[32] Ujval J. Kapasi et al. Programmable stream processors. *IEEE Computer*, 2003.

[33] Michal Karczmarek, William Thies, and Saman Amarasinghe. Phased scheduling of stream programs. In *LCTES*, 2003.

[34] Christoforos E. Kozyrakis and David A. Patterson. A new direction for computer architecture research. *IEEE Computer*, pages 31–11, 1998.

[35] Edward Ashford Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[36] Xuanhua Li and Donald Yeung. Application-level correctness and its impact on fault tolerance. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 181–192, Washington, DC, USA, 2007. IEEE Computer Society.

[37] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *SIGGRAPH*, 2003.

[38] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. *SIGPLAN Not.*, 26(6):1–14, May 1991.

[39] James Mayfield, Tim Finin, and Marty Hall. Using automatic memoization as a software engineering tool in real-world ai systems, 1995.

[40] Donald Michie. Memo functions: a language feature with rote learning properties. Technical Report MIP-R-29, University of Edinburgh, Scotland, 1967.

[41] Donald Michie. Memo functions and machine learning. *Nature*, 218:1922, 1968.

[42] Mehmet Kivanç Mihçak and Ramarathnam Venkatesan. New iterative geometric methods for robust perceptual image hashing. In *Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, DRM '01, pages 13–21, London, UK, UK, 2002. Springer-Verlag.

[43] J. P. Nolan. *Stable Distributions - Models for Heavy Tailed Data*. Birkhauser, Boston, 2012. In progress, Chapter 1 online at academic2.american.edu/~jpnolan.

[44] Peter Norvig. Techniques for automatic memoization with applications to context-free parsing. *Comput. Linguist.*, 17(1):91–98, March 1991.

[45] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[46] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. In *Conference on Languages, Compilers, Tools for Embedded Systems (LCTES)*, 2005.

[47] William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *PACT*, New York, NY, USA, 2010.

[48] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC*, France, 2002.

[49] Robert P. Wilson et al. Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, December 1994.

[50] Lotfi A. Zadeh. Fuzzy logic, neural networks, and soft computing. *Commun. ACM*, 37(3):77–84, March 1994.

[51] Dan Zhang, Zeng-Zhi Li, Hong Song, and Long Liu. A Programming Model for an Embedded Media Processing Architecture. In *SAMOS*, 2005.