Helium: Lifting High-Performance Stencil Kernels from Stripped x86 Binaries to Halide DSL Code

by

Thirimadura Charith Yasendra Mendis

B.Sc., Electronics and Telecommunication Engineering
University of Moratuwa, Sri Lanka, 2013

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

Author . . . .
Department of Electrical Engineering and Computer Science
May 14, 2015

Certified by . . . .
Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Helium: Lifting High-Performance Stencil Kernels from Stripped x86 Binaries to Halide DSL Code

by

Thirimadura Charith Yasendra Mendis

Submitted to the Department of Electrical Engineering and Computer Science
on May 14, 2015, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Highly optimized programs are prone to bit rot, where performance quickly becomes suboptimal in the face of new hardware and compiler techniques. In this paper we show how to automatically lift performance-critical stencil kernels from a stripped x86 binary and generate the corresponding code in the high-level domain-specific language Halide. Using Halide's state-of-the-art optimizations targeting current hardware, we show that new optimized versions of these kernels can replace the originals to rejuvenate the application for newer hardware.

The original optimized code for kernels in stripped binaries is nearly impossible to analyze statically. Instead, we rely on dynamic traces to regenerate the kernels. We perform buffer structure reconstruction to identify input, intermediate and output buffer shapes. We abstract from a forest of concrete dependency trees which contain absolute memory addresses to symbolic trees suitable for high-level code generation. This is done by canonicalizing trees, clustering them based on structure, inferring higher-dimensional buffer accesses and finally by solving a set of linear equations based on buffer accesses to lift them up to simple, high-level expressions.

Helium can handle highly optimized, complex stencil kernels with input-dependent conditionals. We lift seven kernels from Adobe Photoshop giving a 75% performance improvement, four kernels from IrfanView, leading to 4.97× performance, and one stencil from the miniGMG multigrid benchmark netting a 4.25× improvement in performance. We manually rejuvenated Photoshop by replacing eleven of Photoshop's filters with our lifted implementations, giving 1.12× speedup without affecting the user experience.

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

This thesis includes work published in [20]. I would like to thank all the coauthors and collaborators who worked on the Helium project: Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao and Saman Amarasinghe. I would also like to thank Vladimir Kiriansky, Derek Bruening, and the DynamoRIO user group for invaluable help debugging DynamoRIO clients and Sarah Kong, Chris Cox, Joseph Hsieh, Alan Erickson, and Jeff Chien of the Photoshop team for their helpful input.

Special thanks goes to my thesis advisor Prof. Saman Amarasinghe, who gave guidance and motivated me to pursue this project. His guidance was extremely helpful in solving the intermediate hurdles we had during the initial parts of development of Helium.

Qin Zhao was helpful in bringing me up to pace with the development of DynamoRIO clients. Sylvain Paris gave insights regarding image processing algorithms and drew some of the images that are included in this thesis and in [20]. Jeffrey Bosboom and Kevin Wu were instrumental in patching Photoshop, autotuning and coming up with the final timing results. I also had useful discussions with Fredrik Kjolstad regarding how to present research work.

Prof. Victor Zue, my graduate counsellor, gave general advice regarding MIT's graduate program and kept track of my requirements.

During the first year of my studies, I was supported by a MITEI fellowship.

Contents

List of Figures

Listings

Chapter 1

Introduction

While lowering a high-level algorithm into an optimized binary executable is well understood, going in the reverse direction—lifting an optimized binary into the high-level algorithm it implements—remains nearly impossible. This is not surprising: lowering eliminates information about data types, control structures, and programmer intent. Inverting this process is far more challenging because stripped binaries lack high-level information about the program. Because of the lack of high-level information, lifting is only possible given constraints, such as a specific domain or limited degree of abstraction to be reintroduced. Still, lifting from a binary program can help reverse engineer a program, identify security vulnerabilities, or translate from one binary format to another.

In this thesis, we lift algorithms from existing binaries for the sake of program *rejuvenation*. Highly optimized programs are especially prone to bit rot. While a program binary often executes correctly years after its creation, its performance is likely suboptimal on newer hardware due to changes in hardware and the advancement of compiler technology since its creation. Re-optimizing production-quality kernels by hand is extremely labor-intensive, requiring many engineer-months even for relatively simple parts of the code [22]. Our goal is to take an existing legacy binary, lift the performance-critical components with sufficient accuracy to a high-level representation, re-optimize them with modern tools, and replace the bit-rotted component with the optimized version. To automatically achieve best performance for the algorithm, we lift the program to an even higher level than the original source code, into a high-level domain-specific language (DSL). At this level, we express the original programmer intent instead of obscuring it with performance-related transformations, letting us apply domain knowledge to exploit modern architectural features without sacrificing performance portability.

Though this is an ambitious goal, aspects of the problem make this attainable. Program rejuvenation only requires transforming performance-critical parts of the program, which often apply relatively simple computations repeatedly to large amounts of data. Even though this high-level algorithm may be simple, the generated code is complicated due to compiler and programmer optimizations such as tiling, vectorization, loop specialization, and unrolling. In this thesis, we introduce dynamic, data-flow driven techniques to abstract away optimization complexities and get to the underlying simplicity of the high-level intent.

We focus on stencil kernels, mainly in the domain of image-processing programs. Stencils, prevalent in image processing kernels used in important applications such as Adobe Photoshop, Microsoft PowerPoint and Google Picasa, use enormous amounts of computation and/or memory bandwidth. As these kernels mostly perform simple data-parallel operations on large image data, they can leverage modern hardware capabilities such as vectorization, parallelization, and graphics processing units (GPUs).

Furthermore, recent programming language and compiler breakthroughs have dramatically improved the performance of stencil algorithms [23, 27, 26, 17]; for example, Halide has demonstrated that writing an image processing kernel in a high level DSL and autotuning it to a specific architecture can lead to 2–10$\times$ performance gains compared to hand-tuned code by expert programmers. In addition, only a few image-processing kernels in Photoshop and other applications are hand-optimized for the latest architectures; many are optimized for older architectures, and some have little optimization for any architecture. Reformulating these kernels as Halide programs makes it possible to rejuvenate these applications to continuously provide state-of-the-art performance by using the Halide compiler and autotuner to optimize for current hardware and replacing the old code with the newly-generated optimized implementation.

Helium is able to lift the underlying simple algorithm from highly optimized stencil kernels to a set of tree representations before it generates Halide code. These simple representations of complex, heavily optimized codes enable users as well as developers to easily understand the computation and could possibly be used as a debugging aid.

Overall, we lift seven filters and portions of four more from Photoshop, four filters from IrfanView, and the smooth stencil from the miniGMG [32] high-performance computing benchmark into Halide code. We then autotune the Halide schedules and compare performance against the original implementations, delivering an average speedup of 1.75 on Photoshop, 4.97 on IrfanView, and 4.25 on miniGMG. The entire process of lifting and regeneration is completely automated. We also manually replace Photoshop kernels with our rejuvenated versions, obtaining a speedup of 1.12× even while constrained by optimization decisions (such as tile size) made by the Photoshop developers.

In addition, lifting can provide opportunities for further optimization. For example, power users of image processing applications create pipelines of kernels for batch processing of images. Hand-optimizing kernel pipelines does not scale due to the combinatorial explosion of possible pipelines. We demonstrate how our techniques apply to a pipeline kernels by creating pipelines of lifted Photoshop and IrfanView kernels and generating optimized code in Halide, obtaining 2.91× and 5.17× faster performance than the original unfused pipelines.

Chapter 2

Overview

In this chapter, we give the workflow, a high-level view of the challenges and how we address them in Helium.

2.1 Workflow

Helium lifts stencils from stripped binaries to high-level code. While static analysis is the only sound way to lift a computation, doing so on a stripped x86 binary is extremely difficult, if not impossible. In x86 binaries, code and data are not necessarily separated, and determining separation in stripped binaries is known to be equivalent to the halting problem [16]. Statically, it is difficult to even find which kernels execute as they are located in different dynamic linked libraries (DLLs) loaded at runtime. Therefore, we use a dynamic data flow analysis built on top of DynamoRIO [8], a dynamic binary instrumentation framework.

Helium is fully automated, only prompting the user to perform any GUI interactions required to run the program under analysis with and without the target stencil. In total, the user runs the program five times for each stencil lifted. Figure 2-1 shows the workflow Helium follows to implement the translation. Overall, the flow is divided into two stages: code localization, described in Chapter 3, and expression extraction, covered in Chapter 4. Details of DynamoRIO instrumentation are given in Chapter 5. Figure 2-2 shows the flow through the system for a blur kernel.

Chapter 6 evaluates Helium's performance, while Chapter 7 gives out the extracted trees and Halide code for the Photoshop filters which Helium lift. Chapter 8 discusses the related work.

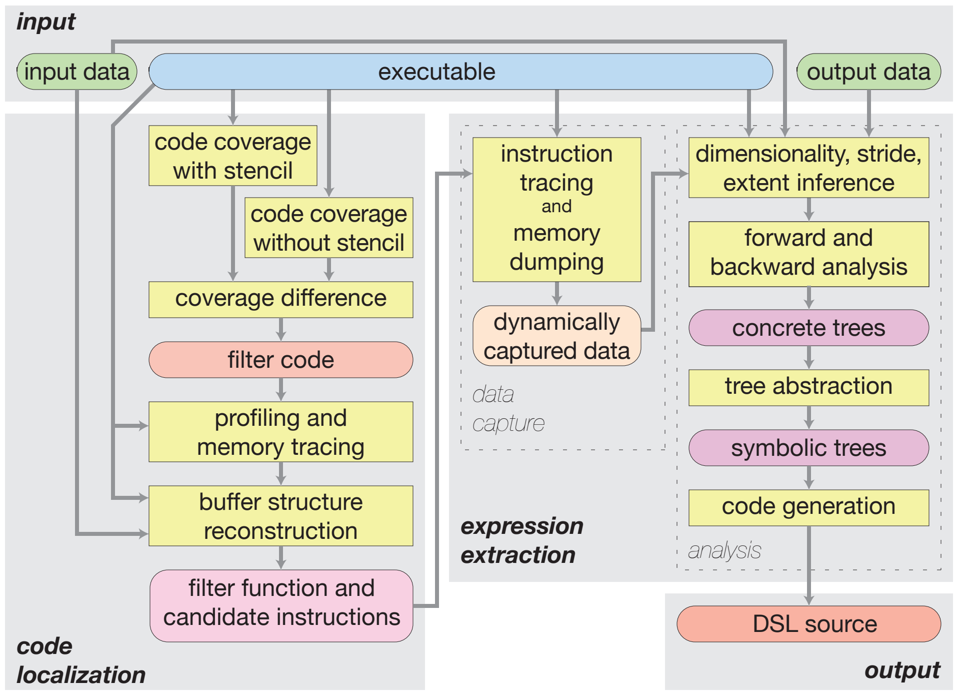
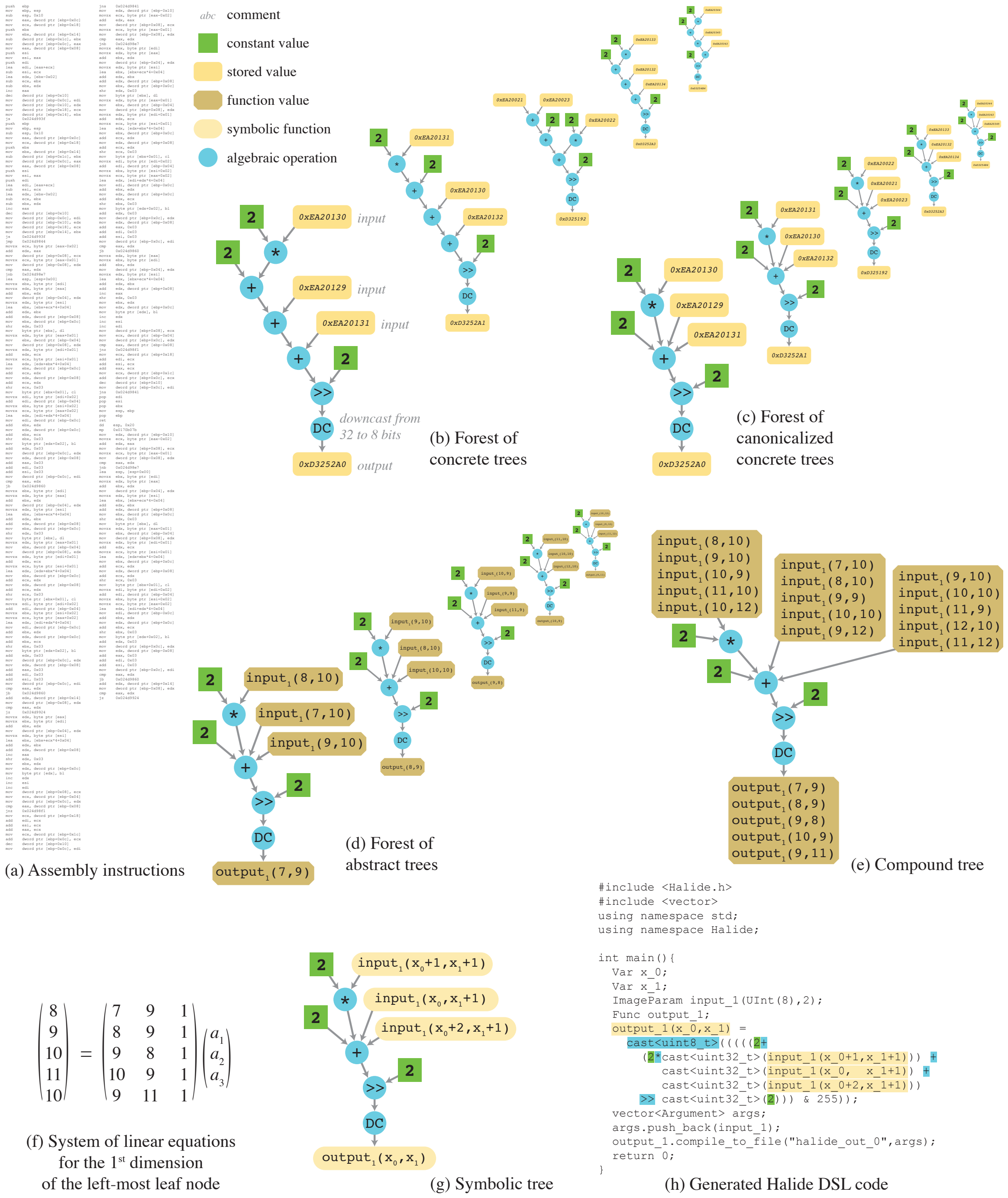

input
input data
executable
output data
code coverage with stencil
code coverage without stencil
coverage difference
filter code
profiling and memory tracing
buffer structure reconstruction
filter function and candidate instructions
code localization
instruction tracing and memory dumping
dynamically captured data
data capture
dimensionality, stride, extent inference
forward and backward analysis
concrete trees
tree abstraction
symbolic trees
code generation
analysis
expression extraction
DSL source
output

Figure 2-1: Helium workflow.

2.2 Challenges & Contributions

Isolating performance critical kernels We find that profiling information alone is unable to identify performance-critical kernels. For example, a highly vectorized kernel of an element-wise operation, such as the invert image filter, may be invoked far fewer iterations than the number of data items. On the other hand, we find that kernels touch all data in input and intermediate buffers to produce a new buffer or the final output. Thus, by using a data-driven approach (described in Section 3.1) and analyzing the extent of memory regions touched by static instructions, we identify kernel code blocks more accurately than through profiling.

Extracting optimized kernels While these stencil kernels may perform logically simple computations, optimized kernel code found in binaries is far from simple. In many applications, programmers expend considerable effort in speeding up performance-critical kernels; such optimizations often interfere with determining the program's purpose. For example, many kernels do not iterate over the image in a simple linear pattern but use smaller tiles for better locality. In fact, Photoshop kernels use a common driver that provides the image as a set of tiles to the kernel. However, we avoid control-flow complexities due to iteration



(a) Assembly instructions

(b) Forest of concrete trees

(c) Forest of canonicalized concrete trees

(d) Forest of abstract trees

(e) Compound tree

$$\begin{pmatrix}8\\9\\10\\11\\10\end{pmatrix} = \begin{pmatrix}7 & 9 & 1\\8 & 9 & 1\\9 & 8 & 1\\10 & 9 & 1\\9 & 11 & 1\end{pmatrix}\begin{pmatrix}a_1\\a_2\\a_3\end{pmatrix}$$

(f) System of linear equations for the 1st dimension of the left-most leaf node

(g) Symbolic tree

```
#include <Halide.h>
#include <vector>
using namespace std;
using namespace Halide;

int main(){
  Var x_0;
  Var x_1;
  ImageParam input_1(UInt(8),2);
  Func output_1;
  output_1(x_0,x_1) =
    cast<uint8_t>(((((2+
      (2*cast<uint32_t>(input_1(x_0+1,x_1+1))) +
         cast<uint32_t>(input_1(x_0,  x_1+1)) +
         cast<uint32_t>(input_1(x_0+2,x_1+1)))
      >> cast<uint32_t>(2))) & 255));
  vector<Argument> args;
  args.push_back(input_1);
  output_1.compile_to_file("halide_out_0",args);
  return 0;
}
```

(h) Generated Halide DSL code

Figure 2-2: Stages of expression extraction for Photoshop's 2D blur filter, reduced to 1D in this figure for brevity. We instrument assembly instructions (a) to recover a forest of concrete trees (b), which we then canonicalize (c). We use buffer structure reconstruction to obtain abstract trees (d). Merging the forest of abstract trees into compound trees (e) gives us linear systems (f) to solve to obtain symbolic trees (g) suitable for generating Halide code (h).

order optimization by only focusing on data flow. For each data item in the output buffer, we compute an expression tree with input and intermediate buffer locations and constants as leaves.

Handling complex control flow A dynamic trace can capture only a single path through the maze of complex control flow in a program. Thus, extracting full control-flow using dynamic analysis is challenging. However, high performance kernels repeatedly execute the same computations on millions of data items. By creating a forest of expression trees, each tree calculating a single output value, we use *expression forest reconstruction* to find a corresponding tree for all the input-dependent control-flow paths. The forest of expression trees shown in Figure 2-2(b) is extracted from execution traces of Photoshop's 2D blur filter code in Figure 2-2(a).

Identifying input-dependent control flow Some computations such as image threshold filters update each pixel differently depending on properties of that pixel. As we create our expression trees by only considering data flow, we will obtain a forest of trees that form multiple clusters without any pattern to identify cluster membership. The complex control flow of these conditional updates is interleaved with the control flow of the iteration ordering, and is thus difficult to disentangle. We solve this problem, as described in Section 4.6, by first doing a forward propagation of input data values to identify instructions that are input-dependent and building expression trees for the input conditions. Then, if a node in our output expression tree has a control flow dependency on the input, we can predicate that tree with the corresponding input condition. During this forward analysis, we also mark address calculations that depend on input values, allowing us to identify lookup tables during backward analysis.

Handling code duplication Many optimized kernels have inner loops unrolled or some iterations peeled off to help optimize the common case. Thus, not all data items are processed by the same assembly instructions. Furthermore, different code paths may compute the same output value using different combinations of operations. We handle this situation by canonicalizing the trees and clustering trees representing the same canonical expression during expression forest reconstruction, as shown in Figure 2-2(c).

Identifying boundary conditions Some stencil kernels perform different calculations at boundaries. Such programs often include loop peeling and complex control flow, making them difficult to handle. In Helium these boundary conditions lead to trees that are different from the rest. By clustering trees (described in Section 4.8), we separate the common stencil operations from the boundary conditions.

Determining buffer dimensions and sizes Accurately extracting stencil computations requires determining dimensionality and the strides of each dimension of the input, intermediate and output buffers. However, at the binary level, multi-dimensional arrays appear to be allocated as one linear block. We introduce *buffer structure reconstruction*, a method which creates multiple levels of coalesced memory regions for inferring dimensions and strides by analyzing data access patterns (Section 3.2). Many stencil computations have ghost regions or padding between dimensions for alignment or graceful handling of boundary conditions. We leverage these regions in our analysis.

Recreating index expressions & generating Halide code Recreating stencil computations requires reconstructing logical index expressions for the multi-dimensional input, intermediate and output buffers. We use access vectors from a randomly selected set of expression trees to create a linear system of equations that can be solved to create the algebraic index expressions, as in Figure 2-2(f). Our method is detailed in Section 4.10. These algebraic index expressions can be directly transformed into a Halide function, as shown in Figure 2-2(g)-(h).

Visualizing and understanding algorithm from optimized code With the inclusion of performance optimizations such as loop unrolling, blocking, tiling, vectorization, parallelization etc. the underlying simple algorithm of stencil kernels become almost indistinguishable from the code which does the performance optimizations. Helium lifts the underlying algorithm of these simple computations from the optimized codes as a set of data dependency trees (Figure 2-2(g)). These trees produced by Helium can be used as a visualization and debugging aid for developers to check whether their optimizations preserve the original programmer intent.

Chapter 3

Code Localization

Helium's first step is to find the code that implements the kernel we want to lift, which we term *code localization*. While the code performing the kernel computation should be frequently executed, Helium cannot simply assume the most frequently executed region of code (which is often just `memcpy`) is the stencil kernel. More detailed profiling is required.

However, performing detailed instrumentation on the entirety of a large application such as Photoshop is impractical, due to both large instrumentation overheads and the sheer volume of the resulting data. Photoshop loads more than 160 binary modules, most of which are unrelated to the filter we wish to extract. Thus the code localization stage consists of a *coverage difference phase* to quickly screen out unrelated code, followed by more invasive profiling to determine the kernel function and the instructions reading and writing the input and output buffers. The kernel function and set of instructions are then used for even more detailed profiling in the expression extraction stage (in Section 4).

3.1 Screening Using Coverage Difference

To obtain a first approximation of the kernel code location, our tool gathers code coverage (at basic block granularity) from two executions of the program that are as similar as possible except that one execution runs the kernel and the other does not. The difference between these executions consists of basic blocks that only execute when the kernel executes. This technique assumes the kernel code is not executed in other parts of the application (e.g., to draw small preview images), and data-reorganization or UI code specific to the kernel will still be captured, but it works well in practice to quickly screen out most of the program code (such as general UI or file parsing code). For Photoshop's blur filter, the coverage difference contains only 3,850 basic blocks out of 500,850 total blocks executed.

Helium then asks the user to run the program again (including the kernel), instrumenting only those basic blocks in the coverage difference. The tool collects basic block execution counts, predecessor blocks and call targets, which will be used to build a dynamic control-flow graph in the next step. Instrumentation is done by the *profiling* DynamoRIO client (see Section 5.3). Helium also collects a dynamic memory trace by instrumenting all memory accesses performed in those basic blocks. Instrumentation is done by the *memory trace* DynamoRIO client (see Section 5.4). The trace contains the instruction address, the absolute memory address, the access width and whether the access is a read or a write. The result of this instrumentation step enables Helium to analyze memory access patterns and detect the filter function.

3.2 Buffer Structure Reconstruction

Helium proceeds by first processing the memory trace to recover the memory layout of the program. Using the memory layout, the tool determines instructions that are likely accessing input and output buffers. Helium then uses the dynamic control-flow graph to select a function containing the highest number of such instructions.

We represent the memory layout as address regions (lists of ranges) annotated with the set of static instructions that access them. For each static instruction, Helium first coalesces any immediately-adjacent memory accesses and removes duplicate addresses, then sorts the resulting regions. The tool then merges regions of different instructions to correctly detect unrolled loops accessing the input data, where a single instruction may only access part of the input data but the loop body as a whole covers the data. Next, Helium links any group of three or more regions separated by a constant stride and of same size to form a single larger region. This proceeds recursively, building larger regions until no regions can be coalesced (see Figure 3-1). Recursive coalescing may occur if e.g. an image filter accesses the R channel of an interleaved RGB image with padding to align each scanline on a 16-byte boundary; the channel stride is 3 and the scanline stride is the image width rounded up to a multiple of 16.

Helium selects all regions of size comparable to or larger than the input and output data sizes and records the associated *candidate instructions* that potentially access the input and output buffers in memory.

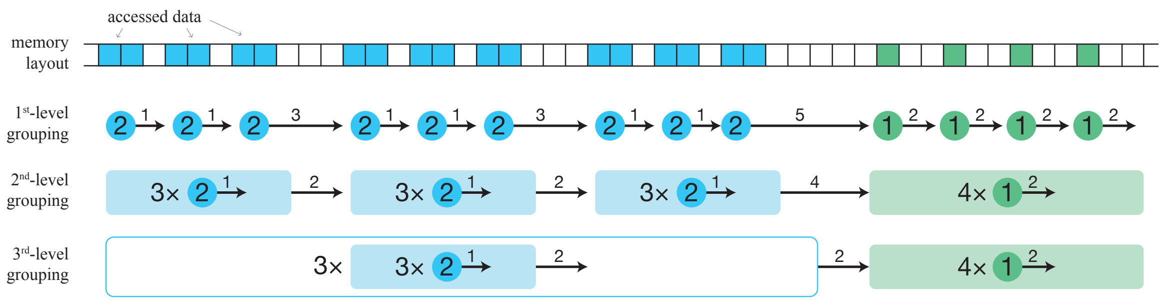


Figure 3-1: During buffer structure reconstruction, Helium groups the absolute addresses from the memory trace into regions, recursively combining regions of the same size separated by constant stride (Linking).

Element Size Helium detects element size based on access width. Some accesses are logically greater than the machine word size, such as a 64-bit addition using an `add/adc` instruction pair. If a buffer is accessed at multiple widths, the tool uses the most common width, allowing it to differentiate between stencil code operating on individual elements and `memcpy`-like code treating the buffer as a block of bits.

Non-rectangular Stencil Kernels When a stencil kernel is non-rectangular, the first and last few regions of contiguous access (before linking) are not of the same size as the middle regions of the buffer. (see Figure 3-2(b),(c)) Hence, these *hanging regions* are not linked with the regions which access the middle part of the buffer during linking.

However, in most cases, the hanging regions are still accessed by a subset of instructions which access the much larger middle region. Therefore, if a particular region is accessed by a subset of instructions annotated to another region of greater size, Helium extends the greater region equal to a multiple of the stride of the smallest sub-region within it (after linking) sufficient to cover the hanging regions (see Figure 3-2(d),(e)). Helium further checks whether the hanging regions are sufficiently close before they are merged to rule out repeatedly used functions like `memcpy`, which access logically different regions which should not be merged. This process helps us to reconstruct buffers correctly when a non-rectangular kernel is used.

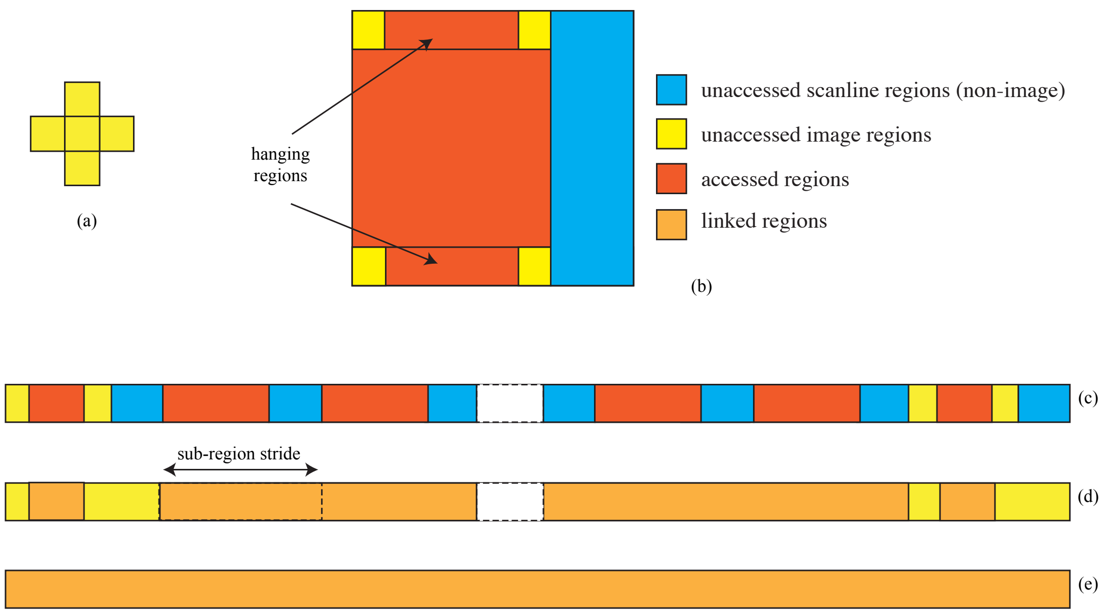


Figure 3-2: Merging hanging regions for non-rectangular stencils (a) 5-point stencil kernel (b) memory layout of the buffer in 2D (c) linearized memory layout (d) after linking (e) after merging hanging regions.

Figure 3-3 depicts the complete process of reconstructing the input buffer of Photoshop's 5-point blur filter (with a non-rectangular stencil) for a 32 $\times$ 32 padded image.

3.3 Filter Function Selection

Helium maps each basic block containing candidate instructions to its containing function using a dynamic control-flow graph built from the profile, predecessor, and call target information collected during screening. The tool considers the function containing the most candidate static instructions to be the kernel. Tail call optimization may fool Helium into selecting a parent function, but this still covers the kernel code; we just instrument more code than necessary.

The chosen function does not always contain the most frequently executed basic block, as one might naïvely assume. For example, Photoshop's invert filter processes four image bytes per loop iteration, so other basic blocks that execute once per pixel execute more often.

Helium selects a filter function for further analysis, rather than a single basic block or a set of functions, as a tradeoff between capturing all the kernel code and limiting the

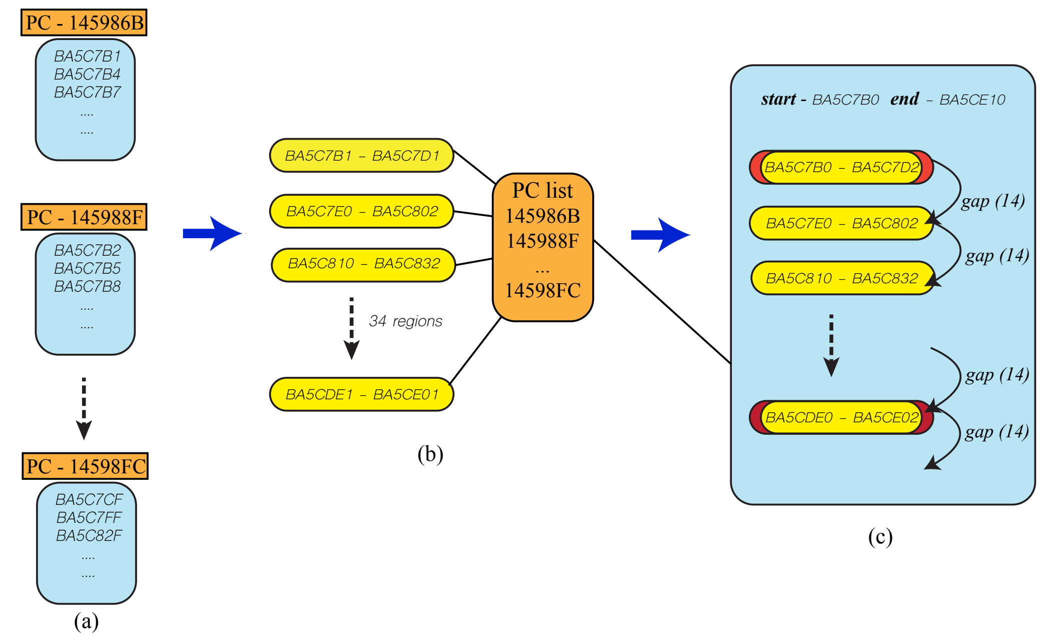


Figure 3-3: Buffer structure reconstruction for Photoshop's 5-point blur filter input buffer for a 32 × 32 padded image (one pixel vertical and horizontal padding) with a scanline stride of 48 (a) initial memory accesses annotated with respective instructions (b) memory regions after coalescing immediately-adjacent memory accesses (c) larger memory region after linking individual memory regions separated by constant stride. The first and last hanging regions are merged. All numbers except the gap values are in hexadecimal.

instrumentation during the expression extraction phase to a manageable amount of code. Instrumenting smaller regions risks not capturing all kernel code, but instrumenting larger regions generates more data that must be analyzed during expression extraction and also increases the likelihood that expression extraction will extract code that does not belong to the kernel (false data dependencies). Empirically, function granularity strikes a good balance. Helium localizes Photoshop's blur to 328 static instructions in 14 basic blocks in the filter function and functions it calls, a manageable number for detailed dynamic instrumentation during expression extraction.

3.4 Localization Results

During localization, we progressively localize the filter code. We increase the level of detail in instrumentation as we progressively narrow the scope of instrumentation. Figure 3-4 shows how code localization phase is able to narrow down the location of the filter code for eleven Photoshop filters.

For these experiments we used a 120 × 144 image for all filters except the sharpen filter. For Photoshop's sharpen filter, we had to use a smaller image (32 × 32), because the code segment for sharpen was used while loading larger images even before the application of the sharpen filter.

Filter	total BB	diff BB	filter func BB
Invert	490663	3401	11
Blur	500850	3850	14
Blur More	499247	2825	16
Sharpen	492433	3027	30
Sharpen More	493608	3054	27
Threshold	491651	2728	60
Box Blur (radius 1)	500297	3306	94
Sharpen Edges	499086	2490	11
Despeckle	499247	2825	16
Equalize	501669	2771	47
Brightness	499292	3012	10

Figure 3-4: Code localization statistics for Photoshop filters, showing the total static basic blocks executed, the static basic blocks surviving screening (Section 3.1), the static basic blocks in the filter function selected at the end of localization (Section 3.3). The filters below the line were not entirely extracted. The total number of basic blocks executed varies due to unknown background code in Photoshop.

In seven out of the eleven filters listed in Figure 3-4, the localized function contains the entire computation of the filter. For other four filters, Helium captures the data intensive parts of the filter. The parts of the filter lying outside the localized function for these four filters are used for computations such as table calculations which do not depend on the input data.

For blur, blur more, invert and despeckle filters the localized function differed from the function that contained the basic block which was executed the most number of times. For Photoshop's brightness filter, we localized two functions having the same number of candidate instructions where one was the same as that for equalize. The reported numbers are for the other function which performs a table lookup.

Chapter 4

Expression Extraction

In this phase, we recover the stencil computation from the filter function found during code localization. Stencils can be represented as relatively simple data-parallel operations with few input-dependent conditionals. Thus, instead of attempting to understand all control flow, we focus on data flow from the input to the output, plus a small set of input-dependent conditionals which affect computation, to extract only the actual computation being performed.

For example, we are able to go from the complex unrolled static disassembly listing in Figure 2-2 (a) for a 1D blur stencil to the simple representation of the filter in Figure 2-2 (g) and finally to DSL code in Figure 2-2 (h).

During expression extraction, Helium performs detailed instrumentation of the filter function, using the captured data for buffer structure reconstruction and dimensionality inference, and then applies expression forest reconstruction to build expression trees suitable for DSL code generation.

4.1 Instruction Trace Capture and Memory Dump

During code localization, Helium determines the entry point of the filter function. The tool now prompts the user to run the program again, applying the kernel to known input data (if available), and collects a trace of all dynamic instructions executed from that function's entry to its exit, along with the absolute addresses of all memory accesses performed by the instructions in the trace. For instructions with indirect memory operands, our tool records the address expression (some or all of $base+scale\times index+disp$). Necessary instrumentation is done by the *instruction trace* DynamoRIO client (see Section 5.5). Helium also collects a page-granularity memory dump of all memory accessed by candidate instructions found in Chapter 3. Read pages are dumped immediately, but written pages are dumped at the filter function's exit to ensure all output has been written before dumping. Instrumentation needed for this part is done by the *memory dump* DynamoRIO client (see Section 5.6). The filter function may execute many times; both the instruction trace and memory dump include all such executions.

4.2 Buffer Structure Reconstruction

Because the user ran the program again during instruction trace capture, we cannot assume buffers have the same location as during code localization. Using the memory addresses recorded as part of the instruction trace, Helium repeats buffer structure reconstruction (Section 3.2) to find memory regions with contiguous memory accesses which are likely the input and output buffers.

4.3 Dimensionality, Stride and Extent Inference

Buffer structure reconstruction finds buffer locations in memory, but to accurately recover the stencil, Helium must infer the buffers' dimensionality, and for each dimension, the stride and extent. For image processing filters (or any domain where the user can provide input and output data), Helium can use the memory dump to recover this information. Otherwise, the tool falls back to generic inference that does not require the input and output data.

Inference using input and output data Helium searches the memory dump for the known input and output data and records the starting and ending locations of the corresponding memory buffers together with any alignment padding. Inputs and outputs for image processing applications are the input and output images, while for other high performance stencil applications which explicitly require input data, the user need to specify to Helium which data was used as input and what was the resultant output after running the program.

Helium next validates that the buffers found through memory dump search are actually accessed by the filter function, by finding whether there are any overlaps between the buffers recovered by performing buffer structure reconstruction and buffers recovered by memory dump search. The memory footprint may include regions which carry the image, but are never accessed by the actual execution of the function and this step ensures that these regions are filtered out. Dimensionality, stride and extents inferred by Helium for the surviving buffers depend on the application's memory layout.

For example, when Photoshop blurs a 32×32 image, it pads each edge by one pixel, then rounds each scanline up to 48 bytes for 16-byte alignment. Photoshop stores the R, G and B planes of a color image separately, so Helium infers three input buffers and three output buffers with two dimensions. All three planes are the same size, so the tool infers each dimension's stride to be 48 (the distance between scanlines) and the extent to be 32. Our other example image processing application, IrfanView, stores the RGB values interleaved, so Helium automatically infers that IrfanView's single input and output buffers have three dimensions.

Generic inference If we do not have input and output data (as in the miniGMG benchmark, which generates simulated input at runtime), or the data cannot be recognized in the memory dump, Helium falls back to generic inference based on buffer structure reconstruction. The dimensionality is equal to the number of levels of recursion needed to coalesce memory regions. Helium can infer buffers of arbitrary dimensionality so long padding exists between dimensions. For the dimension with least stride, the extent is equal to the number of adjacent memory locations accessed in one grouping and the stride is equal to the memory access width of the instructions affecting this region. For all other dimensions, the stride is the difference between the starting addresses of two adjacent memory regions in the same level of coalescing and the extent is equal to the number of independent memory regions present at each level.

For example, consider Figure 3-1. Helium would infer two buffers with dimensions, strides and extents as listed in Figure 4-1.

Buffer	Dimensions	Strides	Extents
Blue buffer	3	1, 3, 11	2, 3, 3
Green buffer	2	1, 3	1, 4

Figure 4-1: Inferred dimensions, strides and extents for example buffer structures presented in Figure 3-1.

If there are no gaps in the reconstructed memory regions, this inference will treat the memory buffer as single-dimensional, regardless of the actual dimensionality.

Inference by forward analysis We have yet to encounter a stencil for which we lack input and output data and for which the generic inference fails, but in that case the application must be handling boundary conditions on its own. In this case, Helium could infer dimensionality and stride by looking at different tree clusters (Section 4.8) and calculating the stride between each tree in a cluster containing the boundary conditions.

When inference is unnecessary If generic inference fails but the application does not handle boundary conditions on its own, the stencil is pointwise (uses only a single input point for each output point). The dimensionality is irrelevant to the computation, so Helium can assume the buffer is linear with a stride of 1 and extent equal to the memory region's size.

4.4 Input/Output Buffer Selection

Helium considers buffers that are read, not written, and not accessed using indices derived from other buffer values to be input buffers. If output data is available, Helium identifies output buffers by locating the output data in the memory dump. Otherwise (or if the output data cannot be found), Helium assumes buffers that are written to with values derived from the input buffers to be output buffers, even if they do not live beyond the function (e.g., temporary buffers).

4.5 Instruction Trace Preprocessing

Before analyzing the instruction trace, Helium preprocesses it by renaming the x87 floating-point register stack using a technique similar to that used in [13]. More specifically, we recreate the floating point stack from the dynamic instruction trace to find the top of the floating point stack, which is necessary to recover non-relative floating-point register locations. Helium also maps registers into memory so the analysis can treat them identically; this is particularly helpful to handle dependencies between partial register reads and writes (e.g., writing to `eax` then reading from `ah`).

Next, Helium transforms each CISC x86 instruction in the instruction trace into a set of RISC like instructions each of which is limited to two sources and one destination. We also make the implicit operands explicit in the reduced form. This allows Helium to treat the transformed instructions in a similar way while preserving the original semantics of the complex instructions. Further, Helium maintains the mapping from x86 instructions to the set of reduced instructions to facilitate any instruction annotations (flagging) that happen in subsequent analysis.

```
                       temp ← eax * r/m32
imul r/m32    →        edx ← high_32(temp)
                       eax ← low_32(temp)
```

Figure 4-2: Reduction of one operand flavor of x86's `imul` instruction.

4.6 Forward Analysis for Input-Dependent Conditionals

While we focus on recovering the stencil computation, we cannot ignore control flow completely because some branches may be part of the computation. Helium must distinguish these *input-dependent conditionals* that affect *what* the stencil computes from the control flow arising from optimized loops controlling *when* the stencil computes.

To capture these conditionals, the tool first identifies which instructions read the input directly using the reconstructed memory layout. Next, Helium does a forward pass through the instruction trace identifying instructions which are affected by the input data, either directly (through data) or through the flags register (control dependencies). The input-dependent conditionals are the input-dependent instructions reading the flag registers (conditional jumps plus a few math instructions such as `adc` and `sbb`). Figure 4-3 and 4-4 shows annotated disassembly for IrfanView's solarize filter and Photoshop's threshold filter after the above analysis.

Then for each static instruction in the filter function, Helium records the sequence of taken/not-taken branches of the input-dependent conditionals required to reach that instruction from the filter function entry point. The result of the forward analysis is a mapping from each static instruction to the input-dependent conditionals (if any) that must be taken or not taken for that instruction to be executed. This mapping is used during backward

```
⋮
jnb 0x1000c2fd
mov cl, byte ptr [eax+esi+0x01]      I
cmp cl, 0x80                         ID, SF
jnb 0x1000c30f                       ID, RF
⋮
```

Figure 4-3: Code snippet from IrfanView's solarize filter; instruction flagging during forward analysis, I - input, SF - sets flags, ID - input dependent, RF - reads flags, O - output.

```
⋮
cmp byte ptr [ecx+eax], dl      I, SF
sbb bl, bl                      ID, RF
and bl, 0x01                    ID
dec bl                          ID
mov byte ptr [esi+eax], bl      ID, O
⋮
```

Figure 4-4: Code snippet from Photoshop's threshold filter; instruction flagging during forward analysis, I - input, SF - sets flags, ID - input dependent, RF - reads flags, O - output

analysis to build predicate trees (see Figure 4-7).

Further, Helium needs to account for dependencies that occur through indirect buffer indexing done using the values of another buffer. To correctly capture these, Helium flags instructions which access buffers using indices derived from other buffers during forward analysis. These flags are used during backward analysis to correctly generate dependencies for indirect buffer accesses.

4.7 Backward Analysis for Data-Dependency Trees

In this step, the tool builds data-dependency trees to capture the exact computation of a given output location. Helium walks backwards through the instruction trace, starting from instructions which write output buffer locations (identified during buffer structure reconstruction). We build a data-dependency tree for each output location by maintaining a frontier of nodes on the leaves of the tree. When the tool finds an instruction that computes the value of a leaf in the frontier, Helium adds the corresponding operation node to the tree, removes the leaf from the frontier and adds the instruction's sources to the frontier if not already present.

We call these *concrete trees* because they contain absolute memory addresses. Figure 2-2 (b) shows a forest of concrete trees for a 1D blur stencil.

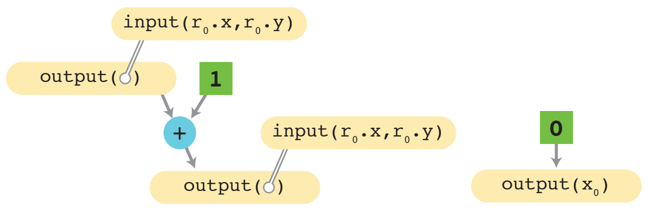
Indirect buffer access Table lookups give rise to indirect buffer accesses, in which a buffer is indexed using values read from another buffer (`buffer_1(input(x,y))`). Figure 4-5 shows an example code snippet.

```
  ⋮
movzx ebx, byte ptr [esi+eax]    I
mov bl, byte ptr [ebx+edx]       ID, IA
dec ecx
mov byte ptr [eax], bl           ID, O
  ⋮
```

Figure 4-5: Code snippet from Photoshop's brightness filter which performs an indirect buffer access, I - input, ID - input dependent, IA - indirect buffer access, O - output. `dec ecx` instruction is not dependent on the input.

If one of the instructions flagged during forward analysis as performing indirect buffer access computes the value of a leaf in the frontier, Helium adds additional operation nodes to the tree describing the address calculation expression (see Figure 4-6). The sources of these additional nodes are added to the frontier along with the other source operands of the instruction to ensure we capture both data and address calculation dependencies.

Recursive trees If Helium adds a node to the data-dependency tree describing a location from the same buffer as the root node, the tree is recursive. To avoid expanding the tree, Helium does not insert that node in the frontier. Instead, the tool builds an additional non-recursive data-dependency tree for the initial write to that output location to capture the base case of the recursion (see Figure 4-6). If all writes to that output location are recursively defined, Helium assumes that the buffer has been initialized outside the function.



(a) Recursive tree (b) Initial update tree

```
Var x_0;
ImageParam input(UInt(8),2);
Func output;
output(x_0) = 0;

RDom r_0(input);
output(input(r_0.x,r_0.y)) =
  cast<uint64_t>(output(input(r_0.x,r_0.y)) + 1);
```

(c) Generated Halide code

Figure 4-6: The trees and lifted Halide code for the histogram computation in Photoshop's histogram equalization filter. The initial update tree (b) initializes the histogram counts to 0. The recursive tree (a) increments the histogram bins using indirect access based on the input image values. The Halide code generated from the recursive tree is highlighted and indirect accesses are in bold.

Known library calls When Helium adds the return value of a call to a known external library function (e.g., `sqrt`, `floor`) to the tree, instead of continuing to expand the tree through that function, it adds an external call node that depends on the call arguments. Handling known calls specially allows Helium to emit corresponding Halide intrinsics instead of presenting the Halide optimizer with the library's optimized implementation (which is often not vectorizable without heroic effort). Helium recognizes these external calls by their symbol, which is present even in stripped binaries because it is required for dynamic linking.

Canonicalization Helium canonicalizes the trees during construction to cope with the vagaries of instruction selection and ordering. For example, if the compiler unrolls a loop, it may commute some but not all of the resulting instructions in the loop body; Helium sorts the operands of commutative operations so it can recognize these trees as similar in the next step. Section 4.12.2 shows specific examples. It also applies simplification rules to these trees to account for effects of fix-up loops inserted by the compiler to handle leftover iterations of the unrolled loop. Figure 2-2 (c) shows the forest of canonicalized concrete trees.

Data types As Helium builds concrete trees, it records the sizes and kinds (signed/unsigned integer or floating-point) of registers and memory to emit the correct operation during Halide code generation (Section 4.11). Narrowing operations are represented as downcast nodes and overlapping dependencies are represented with full or partial overlap nodes. See Section 4.12.1 for more details.

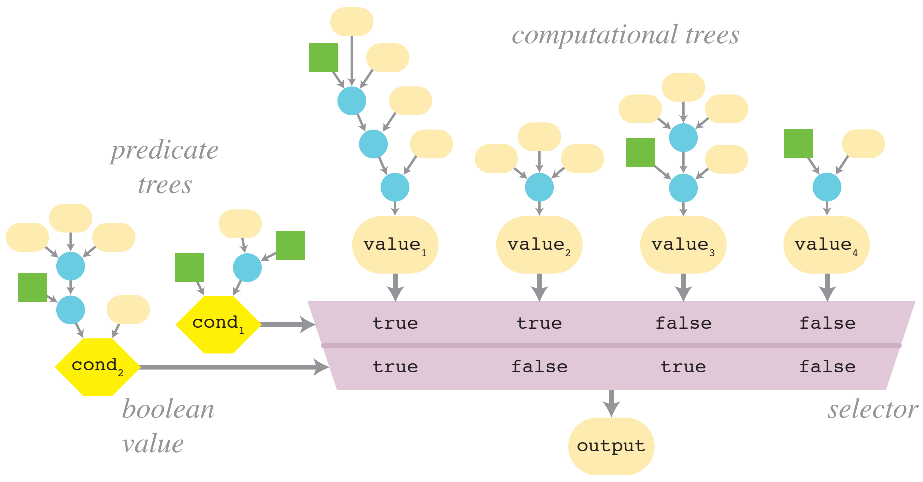


Figure 4-7: Each computational tree (four right trees) has zero or more predicate trees (two left trees) controlling its execution. Code generated for the predicate trees controls the execution of the code generated for the computational trees, like a multiplexer.

Predication Each time Helium adds an instruction to the tree, if that instruction is annotated with one or more input-dependent conditionals identified during the forward analysis, it records the tree as predicated on those conditionals. Once it finishes constructing the tree for the computation of the output location, Helium builds similar concrete trees for the dependencies of the predicates the tree is predicated on (that is, the data dependencies that control whether the branches are taken or not taken). At the end of the backward analysis, Helium has built a concrete *computational tree* for each output location (or two trees if that location is updated recursively), each with zero or more concrete *predicate trees* attached. During code generation, Helium uses predicate trees to generate code that selects which computational tree code to execute (see Figure 4-7).

4.8 Tree Clustering and Buffer Inference

Helium groups the concrete canonicalized computational trees into clusters, where two trees are placed in the same cluster if they are the same, including all predicate trees they depend on, modulo constants and memory addresses in the leaves of the trees. (Recall that registers were mapped to special memory locations during preprocessing.) The number of clusters depends on the control dependency paths taken during execution for each output location. Each control dependency path will have its own cluster of computational trees. Most kernels have very few input-dependent conditionals relative to the input size, so there will usually be a small number of clusters each containing many trees. Figure 4-7 shows an example of clustering (only one computational tree is shown for brevity). For the 1D blur example, there is only one cluster as the computation is uniform across the padded image. For the threshold filter in Photoshop, we get two clusters.

Next, our goal is to abstract these trees. Using the dimensions, strides and extents inferred in 4.3, Helium can convert memory addresses to concrete indices (e.g., memory address 0xD3252A0 to `output_1(7,9)`). We call this *buffer inference*. Algorithm 1 outlines the process followed.

Algorithm 1 Buffer Indices Calculation

```
Input: $M$ - memory location to be abstracted
Output: $x$ - vector of buffer indices (least strided dimension first)
1: Find buffer ($B$) which contains address $M$
2: Let $S$ = start address of $B$
3: Let $D$ = dimensionality of $B$
4: offset = $M - S$
5: for $i = D$ to 1 do
6:     $x_i$ = offset / stride$_{B,i}$ (integer divide)
7:     offset -= $x_i \times$ stride$_{B,i}$
```

At this stage the tool also detects function parameters, assuming that any register or memory location that is not in a buffer is a parameter. After performing buffer inference on the concrete computational trees and attached predicate trees, we obtain a set of *abstract computational and predicate trees* for each cluster. Figure 2-2 (d) shows the forest of abstract trees for the 1D blur stencil. The leaves of these trees are buffers, constants or parameters.

4.9 Reduction Domain Inference

If a cluster contains recursive trees, Helium must infer a *reduction domain* specifying the range in each dimension for which the reduction is to be performed. If the root nodes of the recursive trees are indirectly accessed using the values of another buffer, then the reduction domain is the bounds of that other buffer. If the initial update tree depends on values originating outside the function, Helium assumes the reduction domain is the bounds of that input buffer.

Otherwise, the tool records the minimum and maximum buffer indices observed in trees in the cluster for each dimension as the bounds of the reduction domain. Helium abstracts these concrete indices using the assumption that the bounds are a linear combination of the buffer extents or constants. This heuristic has been sufficient for our applications, but a more precise determination could be made by applying the analysis to multiple sets of input data with varying dimensions and solving the resulting set of linear equations.

4.10 Symbolic Tree Generation

At this stage, the abstract trees contain many relations between different buffer locations (e.g., `output(3,4)` depends on `input(4,4)`, `input(3,3)`, and `input(3,4)`). To convert these dependencies between specific index values into symbolic dependencies between buffer coordinate locations, Helium assumes an affine relationship between indices and solves a linear system. The rest of this section details the procedure that Helium applies to the abstract computational and predicate trees to convert them into *symbolic trees*.

We represent a stencil with the following generic formulation. For the sake of brevity, we use a simple addition as our example and conditionals are omitted.

```
for $x_1 = \dots$
  $\dots$
  for $x_D = \dots$
    output$[x_1]\dots[x_D] =$
      buffer$_1[f_{1,1}(x_1,\dots,x_D)]\dots[f_{1,k}(x_1,\dots,x_D)]+$
      $\dots +$ buffer$_n[f_{n,1}(x_1,\dots,x_D)]\dots[f_{n,k}(x_1,\dots,x_D)]$
```

where `buffer` refers to the buffers that appear in leaf nodes in an abstract tree and `output` is the root of that tree. The functions $f_{1,1},\dots,f_{n,k}$ are *index functions* that describe the relationship between the buffer indices and the output indices. Each index function is specific to a given leaf node and to a given dimension. In our work, we consider only affine index functions, which covers many practical scenarios. We also define the *access vector* $\vec{x} = (x_1, \dots x_D)$.

For a D-dimensional output buffer at the root of the tree with access vector $\vec{x}$, a general affine index function for the leaf node ℓ and dimension d is $f_{\ell,d}(\vec{x}) = [\vec{x}; 1] \cdot \vec{a}$ where $\vec{a}$ is the $(D+1)$-dimensional vector of the affine coefficients that we seek to estimate. For a single abstract tree, this equation is underconstrained but since all the abstract trees in a cluster share the same index functions for each leaf node and dimension, Helium can accumulate constraints and make the problem well-posed. In each cluster, for each leaf node and dimension, Helium formulates a set of linear equations with $\vec{a}$ as unknown.

In practice, for data-intensive applications, there are always at least $D + 1$ trees in each cluster, which guarantees that our tool can solve for $\vec{a}$. To prevent slowdown from a hugely overconstrained problem, Helium randomly selects a few trees to form the system. $D + 1$ trees would be enough to solve the system, but we use more to detect cases where the index function is not affine. Helium checks that the rank of the system is $D + 1$ and generates an error if it is not. In theory, $D + 2$ random trees would be sufficient to detect such cases with some probability; in our experiments, our tool uses $2D + 1$ random trees to increase detection probability.

Helium solves similar sets of linear equations to derive affine relationships between the output buffer indices and constant values in leaf nodes.

As a special case, if a particular cluster's trees have an index in any dimension which does not change for all trees in that cluster, Helium assumes that dimension is fixed to that particular value instead of solving a linear system.

Once the tool selects a random set of abstract trees, a naïve solution to form the systems of equations corresponding to each leaf node and each dimension would be to go through all the trees each time. However, all the trees in each cluster have the same structure. This allows Helium to merge the randomly selected trees into a single *compound tree* with the same structure but with extended leaf and root nodes that contain all relevant buffers (Figure 2-2(e)). With this tree, generating the systems of equations amounts to a single traversal of its leaf nodes.

At the end of this process, for each cluster, we now have a symbolic computational tree possibly associated with symbolic predicate trees as illustrated in Figure 2-2(g). Section 7.1 presents symbolic trees extracted for some Photoshop filters.

4.10.1 Index Function Abstraction Example

For example, consider the compound tree in Figure 2-2(e). The root of the compound tree contains the buffer accesses of the output buffer for each abstract tree used in constructing the compound tree. Therefore, the matrix with access vectors for the output buffer augmented with a column of 1s can be written as in Eq. (4.1).

$$A = \begin{pmatrix} 7 & 8 & 9 & 10 & 9 \\ 9 & 9 & 8 & 9 & 11 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}^T \tag{4.1}$$

Let's consider the left topmost node, the vectors for the index function values corresponding to both the dimensions of the node F_1, F_2 can be given as in Eq. (4.2).

$$F_1 = \begin{pmatrix} 8 & 9 & 10 & 11 & 10 \end{pmatrix}^T \qquad F_2 = \begin{pmatrix} 10 & 10 & 9 & 10 & 12 \end{pmatrix}^T \tag{4.2}$$

We can then write two linear systems and solve for the affine coefficients $\vec{a_1}$ and $\vec{a_2}$ as in Eq. (4.3).

$$F_1 = A \cdot \vec{a_1} \qquad F_2 = A \cdot \vec{a_2} \tag{4.3}$$

The yields in solutions, $\vec{a_1} = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}^T$ and $\vec{a_2} = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix}^T$. Therefore, the final abstraction for the left topmost node of Figure 2-2(e) is `input`$_1$`(x`$_0$`+1, x`$_1$`+1)`. Likewise, for each buffer node residing in the leaves and for each dimension of such nodes, Helium builds up a linear system to find the symbolic relationships between output buffer indexes and the buffer indexes at tree leaves. The complete set of symbolic relationships for Photoshop blur filter in 1D can be referenced in Figure 2-2(g).

4.11 Halide Code Generation

The symbolic trees are a high-level representation of the algorithm. Helium extracts only the necessary set of predicate trees, ignoring control flow arising from loops. Our symbolic trees of data dependencies between buffers match Halide's functional style, so code generation is straightforward.

4.11.1 Declarations

Halide requires input, output buffers, parameters as well as variables controlling iteration order to be defined before initial usage. Helium locates the correct values for these components from the symbolic trees in the following manner.

Output buffers The root nodes of each tree represents an output buffer and Helium emits Halide's `Func` declarations for these output buffers.

Input buffers Any other buffer that is accessed other than the output buffers are considered as pre-populated input buffers and reside as leaf nodes of the tree. If the tree is recursive and there is no non-recursive initial update of the same buffer location, Helium assumes that the output buffer is also pre-populated with values before the function entry point and subsequently creates a new input buffer declaration to handle this case. Input buffers are declared using Halide's `ImageParam` construct. Buffer dimensionality and data type information is preserved from analysis done in Section 4.3.

Parameters Any other non-buffer leaf nodes are considered parameters. Their data types are inferred by the access width and by determining whether they hold floating point or integer values. Helium emits Halide's `Param`s construct in declaring these.

Iteration variables Helium defines iteration variables for each new dimension of the output buffer nodes. As these are reusable by multiple function definitions, Helium defines new iteration variables up to the maximum dimensionality of all buffers. To declare these, Halide's `Var` construct is used.

4.11.2 Definitions

Helium maps the computational tree in each cluster to a Halide function predicated on the predicate trees associated with it. Kernels without data-dependent control flow have just one computational tree, which maps directly to a Halide function. Kernels with data-dependent control flow have multiple computational trees that reference different predicate trees. The predicate trees are lifted to the top of the Halide function and mapped to a chain of `select` expressions (the Halide equivalent of C's `?:` operator) that select the computational tree code to execute.

If a recursive tree's base case is known, it is defined as a Halide function; otherwise, Helium assumes an initialized input buffer will be passed to the generated code. The inferred reduction domain is used to create a Halide `RDom` whose variables are used to define the recursive case of the tree as a second Halide function. Recursive trees can still be predicated as above.

Figure 2-2(h) shows the Halide code generated for the symbolic tree in Figure 2-2(g). Section 7.2 presents Halide code extracted for some Photoshop filters.

4.12 Details on Data-dependency Trees

In this section, we present detailed descriptions of techniques we used to correctly account for full and partial overlap dependencies whilst building data-dependency trees and on techniques used for canonicalizing them.

4.12.1 Partial and Full Overlap Dependencies

Full overlap dependency happens when the application writes to a larger region (a register or memory; remember registers are mapped to special memory locations) and then subsequently reads from a smaller region which is a subset of the larger region and partial overlap dependency happens when the application writes to a region and later reads from a region which partially overlaps with the former region. Partial and full overlap dependencies must be correctly accounted when we are performing backward and forward analysis.

Full Overlap Dependencies

Full overlap dependancies happen quite often when a full register is written and later part of it is read (Figure 4-8). The `shr` instruction updates `edx` and later the partial register `dl` (lower 8 bytes of `edx`) is read, giving rise to a full overlap dependency. Full overlap dependencies can also occur when a larger region of memory is written and subsequently a subregion is read (e.g., write to an `oword` followed by a read from a `dword` starting at a particular memory address).

```
shr edx, 0x03
mov byte ptr [ebx], dl
```

Figure 4-8: Code snippet showing an example full overlap dependency.

We detect such cases and insert a special full dependency node together with the node for the fully overlapped region in the data dependency tree. This is finally translated to `casts` and shifts in Halide code generation to account for data access width differences and

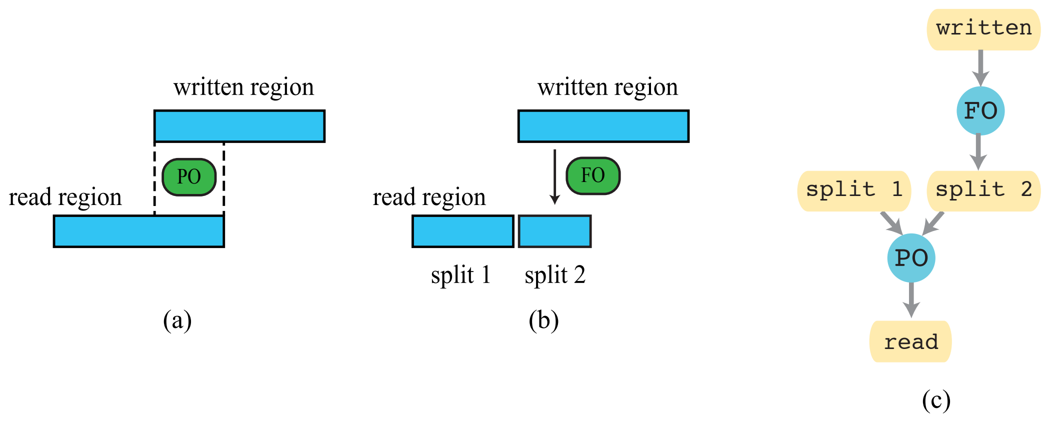


Figure 4-9: Example partial overlap dependency: (a) partial overlap of read and written regions (b) split regions and full overlap dependency with the written region (c) tree representation; FO - full overlap, PO - partial overlap.

relative location differences. If the full overlap is a narrowing operation we represent them as downcast nodes which are translated by Helium as a bit masking operation.

Partial Overlap Dependencies

Partial overlap dependencies arise when the application writes to a certain part of the memory and reads from part of it, where the read region partially overlaps with the written region.

In order to correctly account for dependencies, first the read region is split into disjoint regions such that now only one region is dependent on the written region and the union of the split regions is equal to the original read region. This treatment enables Helium to account for dependencies arising from split regions separately ensuring correctness. The new dependent region (after split) can be a directly dependent or a full overlapped region with the written region.

In case of a partial overlap dependency, a special partial overlap node is added to the dependency tree to which the split regions are connected. Helium then reanalyzes the nodes of the tree to find the dependent region (from the split regions) and updates the dependency. Figure 4-9 shows an example of how partial overlaps are treated in Helium.

In Halide code generation partial overlaps lead to code involving `casts`, shifts and other bitwise operations.

4.12.2 Canonicalization

Importance of Canonicalization

Consider the two uncanonicalized subtrees generated from Photoshop's blur more filter for two distinct output locations (Figure 4-10). Clearly, they are performing the same computation, but the tree structures are different due to instruction reordering in unrolled loops. Without canonicalization, Helium would infer two sets of clusters for the same computation or even worse Helium will have inconsistencies in the linear system of equations solved to produce symbolic trees (see Section 4.10), as there is no guarantee that memory locations in the concrete dependency trees will be in the same order.

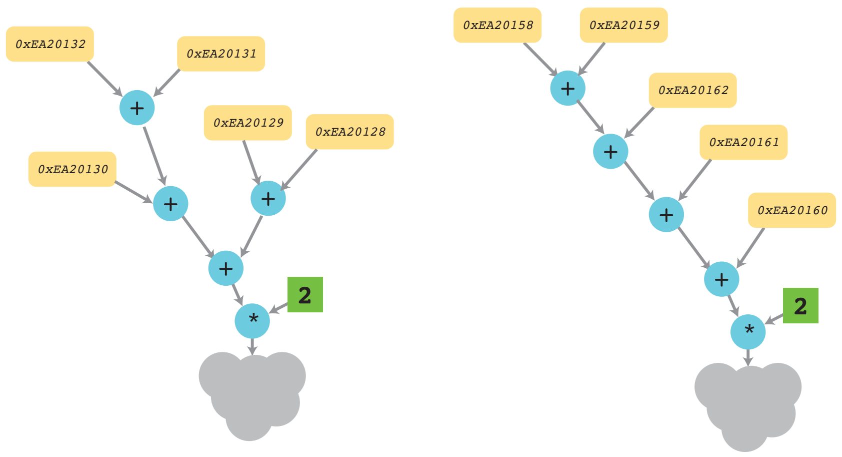


Figure 4-10: Two different tree structures for the same computation in Photoshop blur more filter.

Canonicalization Strategy

Helium canonicalizes trees both during and after construction by applying rewrite rules and sorting. Tree-wide simplifications are performed at the end of tree building, while simplification of identities and data movements happen during tree construction to reduce tree size whilst it is being built.

During Tree Construction

Removal of Assignment Nodes x86 programs frequently move data between registers and memory. These movements add dependencies between nodes of the data dependency trees without actually performing an arithmetic operation. Without any simplification, these are captured as *assignment* nodes. Helium automatically removes these assignment nodes during tree construction and only keeps the most up-to-date source nodes in the tree (Figure 4-11). This reduces the tree size while preserving the computation.

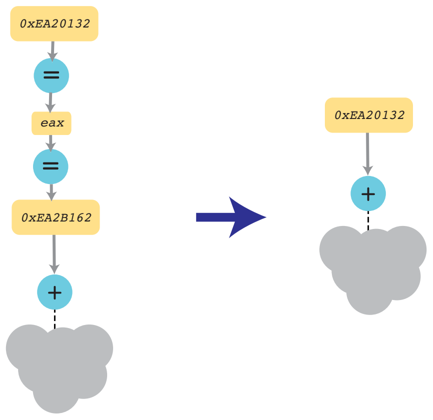


Figure 4-11: Removal of assignment nodes during tree building.

Identities Helium simplifies trees containing additive and multiplicative identities. Further, it simplifies trees with bitwise operations that lead to constants (e.g., `operand | 0xFF = 0xFF` for 8 bit operands; Figure 4-12).

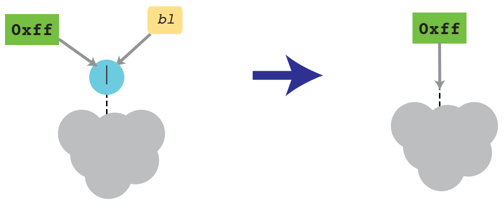


Figure 4-12: Simplification of an identity; occurs during IrfanView's invert filter.

After Tree Construction

We define the operation of *tree ordering* in which the associative operators of the same type are grouped together and for any commutative operators the operands are also sorted (Figure 4-14). While canonicalizing trees, Helium performs number of simplification passes through the trees. After each of these simplification passes, the tree is ordered to ensure canonical structure. A few of Helium's simplification tree rewrite rules are presented below.

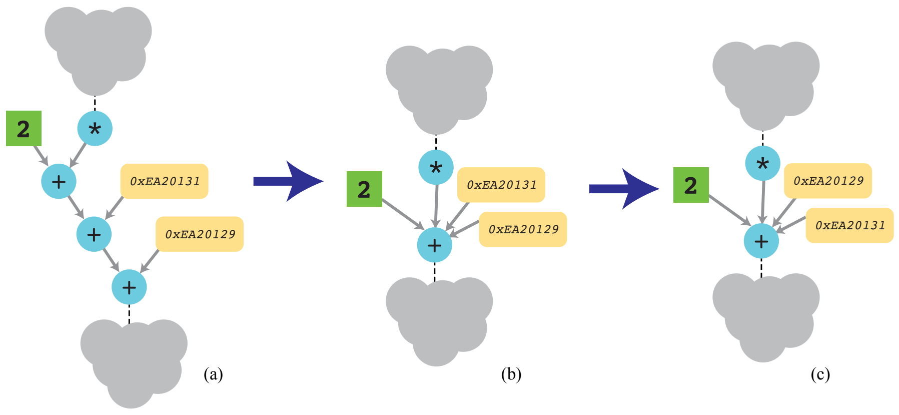


Figure 4-13: Tree ordering ; (a) unordered tree (b) associative operators groups (c) operands of commutative operators sorted.

Constant Folding If there are more than one constant attached to an operator, Helium simplifies them by performing the relevant operation and finally by replacing all of them by the simplified value (e.g., addition of number of constant values). Example occurrence is when a subtree is built using loop carried additions (Photoshop's add noise filter).

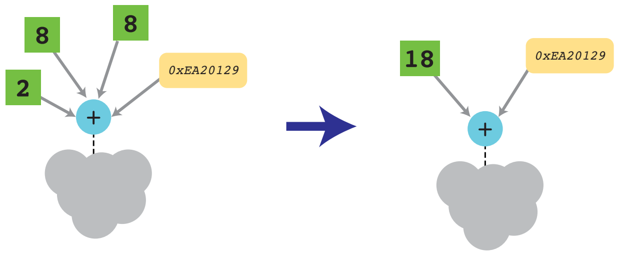


Figure 4-14: Constant folding.

Constant Multiplication Expansion Multiplication of a subtree by a constant n is replaced by an addition of n replicas of that subtree (Figure 4-15). The maximum expansion factor is heuristically chosen to be 10, to keep tree size at a manageable level. This allows the tree ordering routine to sort operands that would otherwise be only visible as a subtree with a multiplication as the root.

Inverse Operation Simplification Helium cancels similar operands which are different in sign attached to an addition node (e.g., $a + (-a) = 0$). As an initial pass, Helium propagates subtractions towards the leaf nodes essentially making them additions of negative values. As additions are associative and commutative, Helium is now able to canonicalize

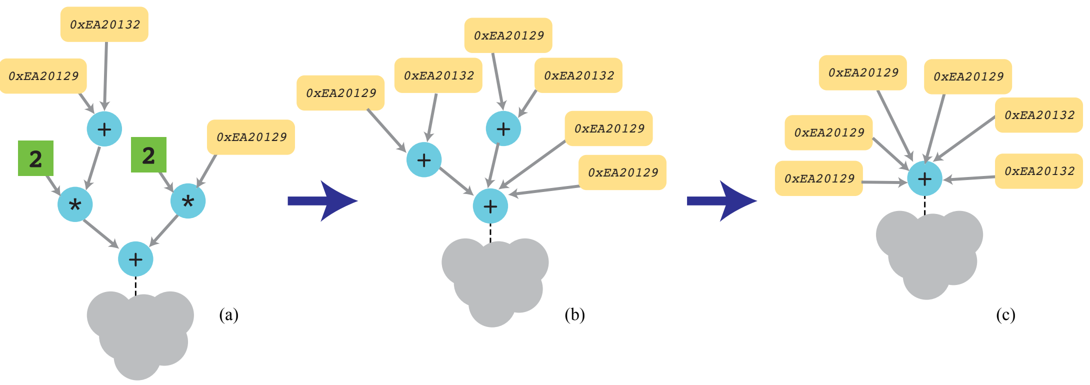


Figure 4-15: Multiplication expansion; (a) before expansion (b) after expansion (c) after tree ordering.

the trees using tree ordering. Finally, it traverses the tree from the root to find cancellation opportunities where the same operands appear connected to an addition node, but with different signs (Figure 4-16). This simplification allows Helium to trim the trees with redundant computations. This simplification is useful in simplifying the structure of Photoshop's box blur filter, which computes a sliding window based local histogram calculation internally leading to addition and subtraction of the same memory location in different parts of the code.

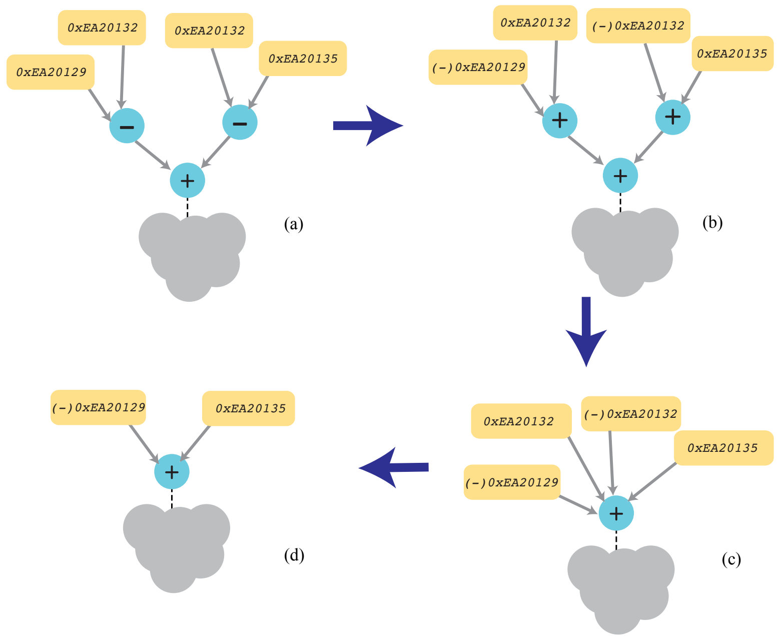


Figure 4-16: Inverse cancellation; (a) before simplification (b) subtraction propagation (c) after tree ordering (d) cancellation of inverses.

Chapter 5

DynamoRIO clients

We used DynamoRIO, a binary instrumentation framework to implement our dynamic analysis tools required to gather information about the application at runtime. DynamoRIO allows users to write their instrumentation routines as *clients*, which register special event callbacks to notify DynamoRIO of the kind of the instrumentation each client wants. In this chapter, we present the details of four clients which gather information required by code localization (Chapter 3) and expression extraction (Chapter 4) stages of our tool. They are the profiling client, which gathers basic block connectivity information, the memory trace client which tracks all memory accesses, the instruction trace client which dumps all the instructions executed with in the instrumented region and the memory dump client which collects a page granularity memory dump from a given region.

5.1 Selective Instrumentation

Binary instrumentation of an entire application may impair its performance and may make it unbearably slow. For an application of the size of Photoshop which loads more than 160 binary modules at runtime, instrumenting each basic block will render the application unresponsive. Therefore, for all DynamoRIO clients we have developed a selective instrumentation library which first checks if a particular basic block needs to be instrumented before it is registered with DynamoRIO. The library allows the clients to selectively instrument a set of instructions, a set of basic blocks, a set of functions, or a set of modules allowing flexibility at each granularity.

The key idea of Helium is to progressively increase the instrumentation granularity while narrowing the scope of instrumentation. This allows our tools to scale to real world commercial applications, yet still capture enough information at the right granularity to facilitate our analysis.

5.2 Information Collection Order

When performing binary instrumentation information can be collected at two instances. Information collected at *instrumentation time* happens only once when a particular basic block is populated in to the code cache. Information collected at *analysis time* originates from the instrumented code and it is executed and updated each time that basic block is run. Our tools collect static information at instrumentation time, while dynamic information is collected during analysis time, to improve scalability by reducing instrumentation overhead.

5.3 Profiling Client

The profiling client collects a profile of the program execution at basic block granularity. For each basic block that is instrumented it collects execution frequency, size of the basic block, whether the final instruction of the basic block is a call instruction or a return instruction and whether the basic block is a call target. Also, for each basic block we collect predecessor and successor basic block information together with addresses of any function callers and callees.

The final output of this instrumentation is a set of files, one for each application thread that executed, containing the profiling information in the following format. All addresses are tracked as offsets from from their respective module start addresses.

```
<start address, size, frequency, is call out, is return, is call target,
{predecessor BBs}, {successor BBs}, {callers}, {callees}>
```

For each of `predecessor BBs, successor BBs, callers and callees`, the client outputs the number of such elements and execution frequencies of each.

An example dump of the profiling information can be found in section A.1. Helium uses this information to build a dynamic control flow graph to aid finding the filter function in code localization.

5.4 Memory Trace Client

The memory trace client traces all memory accesses done in the instrumented basic blocks. The client tracks the absolute memory addresses of memory accesses, whether it is a read or a write, access width and the instruction address. An example dump of memory trace information can be found in section A.2.

Memory traces act as an input to buffer structure reconstruction during code localization phase.

5.5 Instruction Trace Client

The instruction trace client traces all the instructions that are executed within the instrumented region of the application. The instrumented region is the filter function that is localized during the code localization phase. The client outputs a set of files, one per application thread, each containing a linear list of instructions which were executed in program execution order.

Instructions have both static and dynamic properties. The static properties include the opcode, relative location of the instruction from the start of the module and all operands whose location can be determined statically (registers and immediates). The dynamic properties include the absolute memory addresses of memory operands (direct and indirect) and the status of the arithmetic flags before instruction execution.

The static properties are populated at instrumentation time to an array and are referenced when the instruction trace is written to files. In order to recover the dynamic properties, the application is instrumented such that a per-thread data structure is populated with the dynamic properties of each instruction as and when they are executed. The per-thread data structure also carries a pointer to the array with static properties.

The instructions are written to output files (one file per thread) in the following format in the program execution order.

```
<opcode, #dsts, {dst info}, #srcs, {src info}, eflags, instr addr>
```

5.5.1 Operand Information

For each source and destination operand, we collect the operand type, access width and the operand location. Operand type can be one of register, heap memory access, stack memory access, integer immediate or floating point immediate. For all memory accesses, we further collect how the memory addresses are calculated. For example, if the memory operand is a base-displacement memory operand, its memory address is calculated as, $base + scale \times index + disp$, where we collect the registers relating to base and index and immediates relating to scale and displacement.

The `dst info` and `src info` fields in the instruction trace are populated using the collected operand information. The order in which the operand information is written to the files is as follows.

```
<type, access width, location, {base, scale, index, displacement}>
```

We separately dump a disassembly trace of the instructions that are executed with in the instrumented region. Helium uses instruction traces during the expression extraction stage to generate Halide code for the filter function.

5.6 Memory Dump Client

The memory dump client dumps each page accessed by a given set of instructions. In Helium, memory dump is used to dump the regions which are accessed by candidate instructions discovered during code localization.

The memory dump client accepts a set of instruction addresses given as relative offsets from the start of a module and tracks memory regions accessed by these instructions. If a particular instruction reads from memory, it dumps the page immediately and notes the memory region. If again an instruction reads from the same page, the client will ignore such accesses. For any memory writes, it accumulates the residing page locations throughout the filter function's execution. Theses pages are dumped at the filter function's exit to ensure all output has been written before dumping.

Chapter 6

Evaluation

In this chapter, we present the extent to which our strategy of expression extraction was successful in extracting stencil kernels from commercial applications, Photoshop and IrfanView and a multigrid benchmark, miniGMG. We also present performance comparisons between execution of native code and the execution of auto-tuned versions of filters lifted by Helium. We also manually replaced the application code with our lifted implementations in order to provide identical user experience and report the performance gains as experienced by users. Finally, we present a case study on Helium's ability to lift the algorithm from complex stencil binaries, oblivious to various optimizations that are already present in them.

6.1 Extraction Results

We used Helium to lift seven filters and portions of four more from Photoshop CS 6 Extended, four filters from IrfanView 4.38, and the smooth stencil from the miniGMG high-performance computing benchmark into Halide code. We do not have access to Photoshop or IrfanView source code. We only have access to their stripped binaries with no symbol information. miniGMG is open source.

For code localization phase all steps were performed using a 120 $\times$ 144 image, for both IrfanView and Photoshop, except for Photoshop's sharpen filter (see Section 3.4). For expression extraction stage a handcrafted image of size 32 $\times$ 32, resembling an arithmetic progression of pixel values was used.

Photoshop We lifted Photoshop's blur, blur more, sharpen, sharpen more, invert, threshold and box blur (for radius 1 only) filters. The blur and sharpen filters are 5-point stencils; blur more, sharpen more and box blur are 9-point stencils. Invert is a pointwise operation that simply flips all the pixel bits. Threshold is a pointwise operation containing an input-dependent conditional: if the input pixel's brightness (a weighted sum of its R, G and B values) is greater than the threshold, the output pixel is set to white, and otherwise it is set to black. See Chapter 7 for symbolic trees and lifted Halide code for Photoshop filters.

We lifted portions of Photoshop's sharpen edges, despeckle, histogram equalization and brightness filters. Sharpen edges alternates between repeatedly updating an image-sized side buffer and updating the image; we lifted the side buffer computation. Despeckle is a composition of blur more and sharpen edges. When run on despeckle, Helium extracts the blur more portion. From histogram equalization, we lifted the histogram calculation, but cannot track the histogram through the equalization stage because equalization does not depend on the input or output images. Brightness builds a 256-entry lookup table from its parameter, which we cannot capture because it does not depend on the images, but we do lift the application of the filter to the input image.

Photoshop contains multiple variants of its filters optimized for different x86 instruction sets (SSE, AVX etc.). Our instrumentation tools intercept the `cpuid` instruction (which tests CPU capabilities) and report to Photoshop that no vector instruction sets are supported; Photoshop falls back to general-purpose x86 instructions.We do this for engineering reasons, to reduce the number of opcodes our backward analysis must understand; this is not a fundamental limitation. The performance comparisons later in this section do not intercept `cpuid` and thus use optimized code paths in Photoshop.

Figure 6-1 shows statistics for code localization, demonstrating that our progressive narrowing strategy allows our dynamic analysis to scale to large applications.

All but one of our lifted filters give bit-identical results to Photoshop's filters on a suite of photographic images, each consisting of 100 megapixels. The lifted implementation of box blur, the only filter we lifted from Photoshop that uses floating-point, differs in the low-order bits of some pixel values due to reassociation.

IrfanView We lifted the blur, sharpen, invert and solarize filters from IrfanView, a batch image converter. IrfanView's blur and sharpen are 9-point stencils. Unlike Photoshop, IrfanView loads the image data into floating-point registers, computes the stencil in floating-point, and rounds the result back to integer. IrfanView has been compiled for maximal processor compatibility, which results in unusual code making heavy use of partial register reads and writes.

Our lifted filters produce visually identical results to IrfanView's filters. The minor differences in the low-order bits are because we assume floating-point addition and multiplication are associative and commutative when canonicalizing trees.

miniGMG To demonstrate the applicability of our tool beyond image processing, we lifted the Jacobi smooth stencil from the miniGMG high-performance computing benchmark. We added a command-line option to skip running the stencil to enable coverage differencing during code localization. Because we do not have input and output image data for this benchmark, we manually specified an estimate of the data size for finding candidate instructions during code localization and we used the generic inference described in section 4.3 during expression extraction. We set `OMP_NUM_THREADS=1` to limit miniGMG to one thread during analysis, but run using full parallelism during evaluation.

Because miniGMG is open source, we were able to check that our lifted stencil is equivalent to the original code using the SymPy[1] symbolic algebra system. We also checked output values for small data sizes.

6.2 Experimental Methodology

We ran our image filter experiments on an Intel Core i7 990X with 6 cores (hyperthreading disabled) running at 3.47GHz with 8 GB RAM and running 64-bit Windows 7. We used the Halide release built from git commit `80015c`. Instrumentation tools for Helium was built using DynamoRIO revision `2773`.

Helium We compiled our lifted Halide code into standalone executables that load an image, time repeated applications of the filter, and save the image for verification. We ran 10 warmup iterations followed by 30 timing iterations. We tuned the schedules for our generated Halide code for six hours each using the OpenTuner-based Halide tuner [4]. We

[1]`http://sympy.org`

Filter	total BB	diff BB	filter func BB	static ins. count	mem dump	dynamic ins. count	tree size
Invert	490663	3401	11	70	32 MB	5520	3
Blur	500850	3850	14	328	32 MB	64644	13
Blur More	499247	2825	16	189	38 MB	111664	62
Sharpen	492433	3027	30	351	36 MB	79369	31
Sharpen More	493608	3054	27	426	37 MB	105374	55
Threshold	491651	2728	60	363	36 MB	45861	8/6/19
Box Blur (radius 1)	500297	3306	94	534	28 MB	125254	253
Sharpen Edges	499086	2490	11	63	46 MB	80628	33
Despeckle	499247	2825	16	189	38 MB	111664	62
Equalize	501669	2771	47	198	8 MB	38243	6
Brightness	499292	3012	10	54	32 MB	21645	3

Figure 6-1: Code localization and extraction statistics for Photoshop filters, showing the total static basic blocks executed, the static basic blocks surviving screening (Section 3.1), the static basic blocks in the filter function selected at the end of localization (Section 3.3), the number of static instructions in the filter function, the memory dump size, the number of dynamic instructions captured in the instruction trace (Section 4.1), and the number of nodes per concrete tree. Threshold has two computational trees with 8 and 6 nodes and one predicate tree with 19 nodes. The filters below the line were not entirely extracted; the extracted portion of despeckle is the same as blur more. The total number of basic blocks executed varies due to unknown background code in Photoshop.

tuned using a 11267 by 8813 24-bit truecolor image and evaluated with a 11959 by 8135 24-bit image.

We cannot usefully compare the performance of the four Photoshop filters that we did not entirely lift this way; we describe their evaluation separately in Section 6.5.

Photoshop We timed Photoshop using the ExtendScript API[2] to programmatically start Photoshop, load the image and invoke filters. While we extracted filters from non-optimized fallback code for old processors, times reported in this section are using Photoshop's choice of code path. In Photoshop's performance preferences, we set the tile size to 1028K (the largest), history states to 1, and cache tiles to 1. This amounts to optimizing Photoshop for batch processing, improving performance by up to 48% over default settings while dramatically reducing measurement variance. We ran 10 warmup iterations and 30 evaluation iterations.

[2]`https://www.adobe.com/devnet/photoshop/scripting.html`

Filter	Photoshop	Helium	speedup
Invert	102.23 ± 1.65	58.74 ± .52	1.74x
Blur	245.87 ± 5.30	93.74 ± .78	2.62x
Blur More	317.97 ± 2.76	283.92 ± 2.52	1.12x
Sharpen	270.40 ± 5.80	110.07 ± .69	2.46x
Sharpen More	305.50 ± 4.13	147.01 ± 2.29	2.08x
Threshold	169.83 ± 1.37	119.34 ± 8.06	1.42x
Box Blur	273.87 ± 2.42	343.02 ± .59	.80x

Filter	IrfanView	Helium	speedup
Invert	215.23 ± 37.98	105.94 ± .78	2.03x
Solarize	220.51 ± 46.96	102.21 ± .55	2.16x
Blur	3129.68 ± 17.39	359.84 ± 3.96	8.70x
Sharpen	3419.67 ± 52.56	489.84 ± 7.78	6.98x

Figure 6-2: Timing comparison (in milliseconds) between Photoshop and IrfanView filters and our lifted Halide-implemented filters on a 11959 by 8135 24-bit truecolor image.

IrfanView IrfanView does not have a scripting interface allowing for timing, so we timed IrfanView running from the command line using PowerShell's `Measure-Command`. We timed 30 executions of IrfanView running each filter and another 30 executions that read and wrote the image without operating on it, taking the difference as the filter execution time.

miniGMG miniGMG is open source, so we compared unmodified miniGMG performance against a version with the loop in the `smooth` stencil function from the OpenMP-based Jacobi smoother replaced with a call to our Halide-compiled lifted stencil. We used a generic Halide schedule template that parallelizes the outer dimension and, when possible, vectorizes the inner dimension. We compared performance against miniGMG using OpenMP on a 2.4GHz Intel Xeon E5-2695v2 machine running Linux with two sockets, 12 cores per socket and 128GB RAM.

6.3 Lifted Filter Performance Results

Photoshop Figure 6-2 compares Photoshop's filters against our standalone executable running our lifted Halide code. We obtain an average speedup of 1.75 on the individual filters (1.90 excluding box blur).

Profiling using Intel VTune shows that Photoshop's blur filter is not vectorized. Photoshop does parallelize across all the machine's hardware threads, but each thread only achieves 10-30% utilization. Our lifted filter provides better performance by blocking and vectorizing in addition to parallelizing.

Photoshop implements box blur with a sliding window, adding one pixel entering the window and subtracting the pixel leaving the window. Our lifted implementation of box blur is slower because Helium cancels these additions and subtractions when canonicalizing the tree, undoing the sliding window optimization.

IrfanView Figure 6-2 compares IrfanView's filters against our Halide applications. We obtain an average speedup of 4.97.

miniGMG For miniGMG, we measure the total time spent in the stencil we translate across all iterations of the multigrid invocation. Unmodified miniGMG spends 28.5 seconds in the kernel, while miniGMG modified to use our lifted Halide smooth stencil finishes in 6.7 seconds for a speedup of 4.25.

6.4 Performance of Filter Pipelines

Professional users of Photoshop normally apply a set of filters one after the other in a feed forward pipeline to process images in a batch. Here, we compare filter pipelines to demonstrate how lifting to a very high-level representation enables additional performance improvements through stencil composition. In general, Halide is able to fuse computation of the multiple stages in the pipeline to achieve better locality of data to improve overall performance.

For Photoshop, our pipeline consists of blur, invert and sharpen more applied consecutively, while for IrfanView we ran a pipeline of sharpen, solarize and blur.

We obtain a speedup of 2.91 for the Photoshop pipeline. Photoshop blurs the entire image, then inverts it, then sharpens more, which has poor locality. Halide inlines blur and invert inside the loops for sharpen more, improving locality while maintaining vectorization and parallelism.

We obtain a speedup of 5.17 for the IrfanView pipeline. IrfanView improves when running the filters as a pipeline, apparently by amortizing the cost of a one-time preparation step, but our Halide code improves further by fusing the actual filters (Figure 6-3).

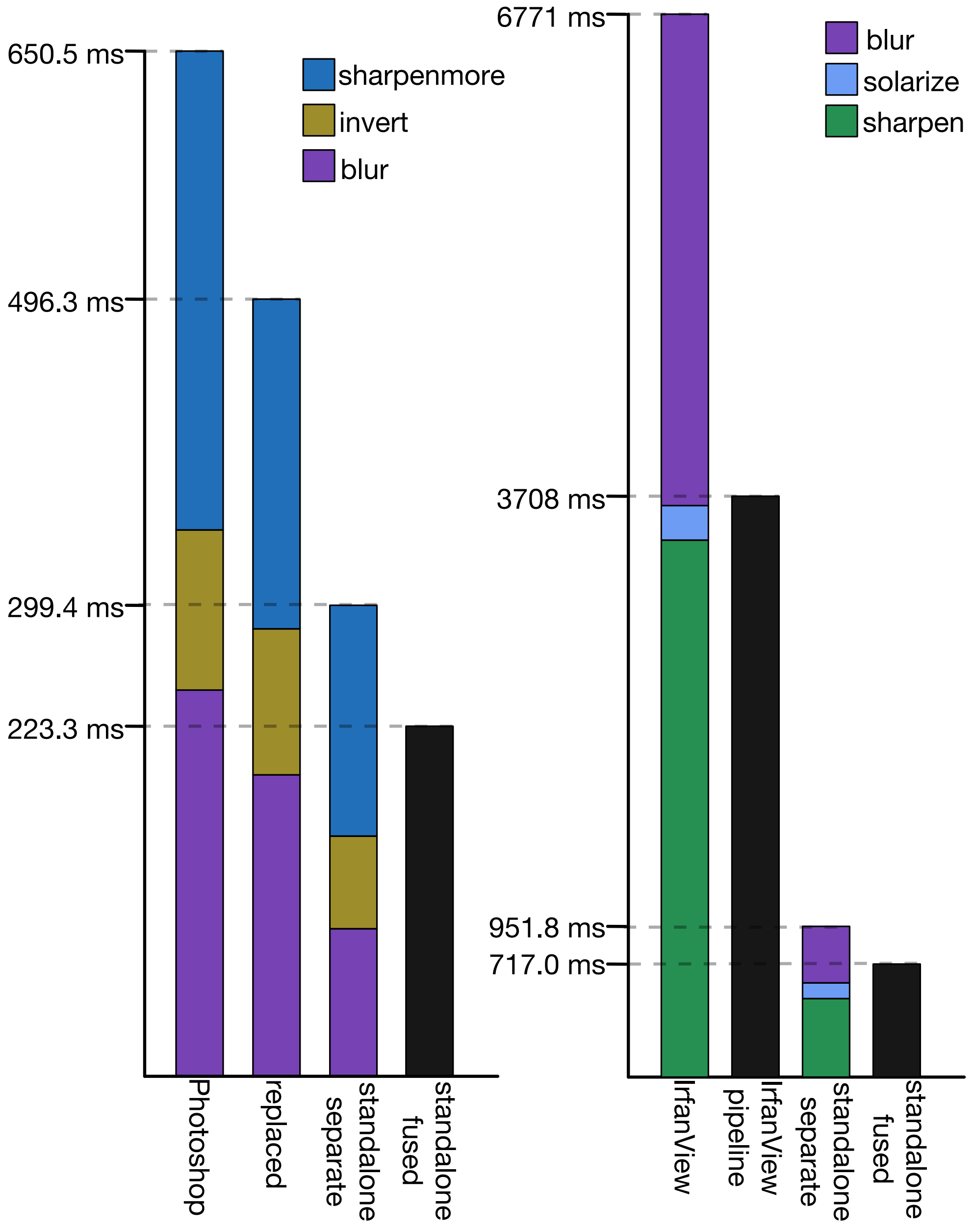


Figure 6-3: Performance comparison of Photoshop and IrfanView pipelines. Left-to-right, the left graph shows Photoshop running the filters in sequence, Photoshop hosting our lifted implementations (Section 6.5), our standalone Halide executable running the filters in sequence, and our Halide executable running the fused pipeline. The right graph shows IrfanView running the filters in sequence, IrfanView running the filters as a pipeline (in one IrfanView instance), our Halide executable running the filters in sequence, and our Halide executable running the fused pipeline.

6.5 In Situ Replacement Photoshop Performance

To evaluate the performance impact of the filters we partially extracted from Photoshop, we replaced Photoshop's implementation with our automatically-generated Halide code using manually-implemented binary patches. We compiled all our Halide code into a DLL that patches specific addresses in Photoshop's code with calls to our Halide code. Manual intervention was needed to figure out the Application Binary Interface that Photoshop was using to pass in function arguments. We had to manually match up the arguments passed in by Photoshop for a particular function with argument's in our Halide replacement to find input image pointers, output image pointers, image buffer extents, strides and other parameters specific to a given filter.

Other than improved performance, these patches are entirely transparent to the user. The disadvantage of this approach is that the patched kernels are constrained by optimization decisions made in Photoshop, such as the granularity of tiling, which restricts our ability to fully optimize the kernels.

When timing the replacements for filters we entirely lift, we disabled Photoshop's parallelism by removing `MultiProcessor Support.8BX` from the Photoshop installation, allowing our Halide code to control parallelism subject to the granularity limit imposed by Photoshop's tile size. When timing the filters we only partially lift, we removed parallelism from the Halide schedule and allow Photoshop to parallelize around our Halide code. While we would prefer to control parallelism ourselves, enough Photoshop code is executing outside the regions we replaced to make disabling Photoshop's parallelism a large performance hit.

Figure 6-4 compares unmodified Photoshop (same numbers as in the previous section) with Photoshop after in situ replacement. For the fully-lifted filters, we are still able to improve performance even while not fully in control of the environment. Our replacement for box blur is still slower for the reason described in Section 6.3. The portions of histogram equalization and brightness we lift are too simple to improve, but the replaced sharpen edges is slightly faster, demonstrating that even when Helium cannot lift the entire computation, it can still lift a performance-relevant portion.

Filter	Photoshop	replacement	speedup
Invert	102.23 ± 1.65	93.20 ± .71	1.10x
Blur	245.87 ± 5.30	191.83 ± 1.12	1.28x
Blur More	317.97 ± 2.76	310.70 ± .88	1.02x
Sharpen	270.40 ± 5.80	194.80 ± .66	1.39x
Sharpen More	305.50 ± 4.13	210.20 ± .71	1.45x
Threshold	169.83 ± 1.37	124.10 ± .76	1.37x
Box Blur	273.87 ± 2.42	395.40 ± .72	.69x
Sharpen Edges	798.43 ± 1.45	728.63 ± 1.85	1.10x
Despeckle	763.87 ± 1.59	756.40 ± 1.59	1.01x
Equalize	405.50 ± 1.45	433.87 ± .90	.93x
Brightness	498.00 ± 1.31	503.47 ± 1.17	.99x

Figure 6-4: Timing comparison (in milliseconds) between Photoshop filters and in situ replacement with our lifted Halide-implemented filters on a 11959 by 8135 24-bit truecolor image.

6.6 Lifting Halide-generated Binaries Back to Halide Source

Helium is oblivious to how the code is scheduled and is able to recover the stencil computation from optimized x86 binaries which are clobbered with scheduling information. In order to test Helium's resilience to arbitrary scheduling, we split, unrolled, tiled, fused, parallelized computational loops using Halide's scheduling language for a 9-point blur filter written in Halide (Code 6.1) and tested whether Helium was able to recover the underlying simplistic algorithm.

```
Var x("x"), y("y"), xi("xi"), yi("yi");
Var x_outer, y_outer, x_inner, y_inner, tile_index;
ImageParam input_1(UInt(8),2);
Func blur_x("blur_x"), blur_y("blur_y");

blur_x(x, y) = (cast<uint16_t>(input(x, y)) + cast<uint16_t>(input(x + 1, y))
                + cast<uint16_t>(input(x + 2, y))) / 3;
blur_y(x, y) = cast<uint8_t>((blur_x(x, y) + blur_x(x, y + 1)
                 + blur_x(x, y + 2)) / 3);
```

Listing 6.1: Original 9-point blur stencil in Halide.

The disassembly generated for each of the schedules are different (see Section B) and statically it is hard to get rid of the scheduling information and recover the underlying computation. However, Helium, which uses dynamic data flow driven approaches is able to ignore instructions performing scheduling and is able to recover the underlying simplistic algorithm of the blur filter.

Figure 6-5 shows the schedules we tested and the amount of static assembly instructions in the filter function localized by Helium. Complex scheduling has increased the amount of static instructions in the filter function, apart from the last parallelized schedule, where Helium was able to find an inner function which performs the computation after parallelization. Under each schedule, Helium recovered the exact same symbolic tree (Figure 6-6) and Halide code (Code 6.2).

Schedule	Filter function static ins.
no schedule	218
`blur_y.unroll(x, 2)`	327

`blur_y.tile(x, y, x_outer,`
` y_outer, x_inner, y_inner, 2, 2)`
	513

`blur_y.tile(x, y, x_outer,`
`        y_outer, x_inner, y_inner, 2, 2)`
`blur_y.fuse`
`        (x_outer, y_outer, tile_index)`
`blur_y.parallel(tile_index)`
	284

Figure 6-5: Number of static instructions in the filter function for various schedules

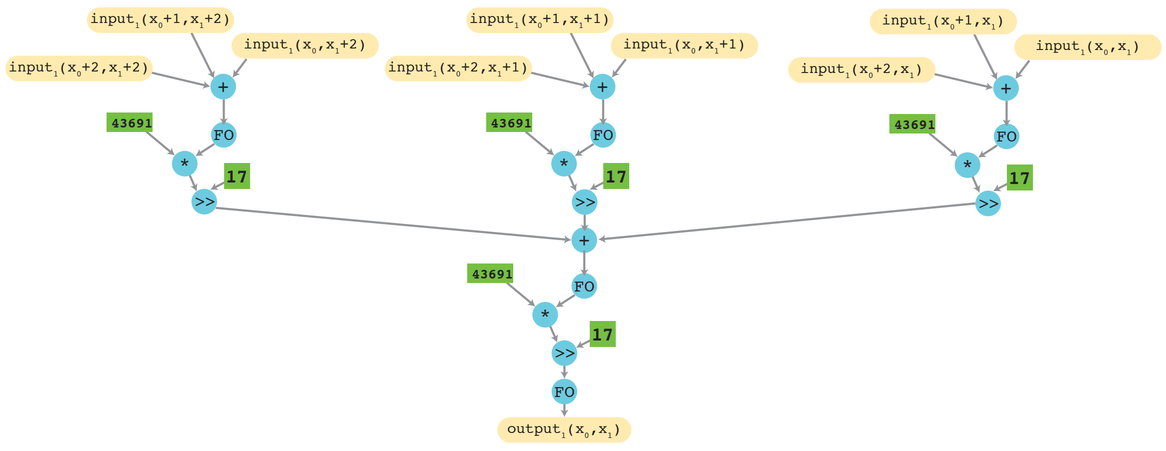


Figure 6-6: Symbolic tree for the 9-point blur filter.

```
Var x_0, x_1, x_2, x_3;
ImageParam input_1(UInt(8),2);
Func output_1;
Expr output_1_p__0 =  ((43691  * cast<uint32_t>((

     ((43691  * cast<uint32_t>((cast<uint32_t>(input_1(x_0,x_1))
                       + cast<uint32_t>(input_1(x_0+1,x_1))
                       + cast<uint32_t>(input_1(x_0+2,x_1)))
                       & 65535)) >> cast<uint32_t>(17))

    + ((43691  * cast<uint32_t>((cast<uint32_t>(input_1(x_0,x_1+1))
                       + cast<uint32_t>(input_1(x_0+1,x_1+1))
                       + cast<uint32_t>(input_1(x_0+2,x_1+1)))
                       & 65535)) >> cast<uint32_t>(17))

    + ((43691  * cast<uint32_t>((cast<uint32_t>(input_1(x_0,x_1+2))
                       + cast<uint32_t>(input_1(x_0+1,x_1+2))
                       + cast<uint32_t>(input_1(x_0+2,x_1+2)))
                       & 65535)) >> cast<uint32_t>(17))
            ) & 65535)) >> cast<uint32_t>(17))  & 255;


output_1(x_0,x_1) = cast<uint8_t>(clamp(output_1_p__0,0,255));
```

Listing 6.2: Halide code lifted by Helium

Helium lifted Halide code differs from the original version in two aspects. First, due to the compiler optimization of replacing constant divisions by reciprocal multiplications has introduced multiplications and bit shifts instead of divisions. Secondly, Helium does not breakup the computation into two buffers. This is a disadvantage for aggressive scheduling and will be added as a feature in future Helium releases.

Chapter 7

Understanding Original Algorithm by Lifting Optimized Code

Simple stencil kernels are normally scheduled in number of different ways in order to increase their cache locality, CPU usage etc. For example, commercial applications such as Photoshop optimize their image processing kernels by doing loop unrolling, loop fusion, vectorization, parallelization etc. Even though, the underlying algorithm is simple, these optimizations obscure *what* the kernel is computing. The complex control flow arising from blocking, vectorization, parallelization, unrolling etc. makes even the source code difficult to analyze.

However, Helium allows the users to visualize the underlying algorithm from heavily optimized stencil code with access to only its binaries without symbols. It allows the users to decouple *what* is being computed form *when* it is being computed. In this chapter, we present symbolic trees and the respective Halide source code for some of the optimized filters we lift from Photoshop using Helium. These simple graphical representations of complex, heavily optimized codes enable users as well as developers to easily understand the computation and could possibly be used as a debugging aid.

7.1 Symbolic Trees for Photoshop filters

In this section, we present symbolic trees recovered by Helium for six Photoshop filters. They are sharpen, sharpen more, blur more, invert, threshold and box blur (radius 1). These trees help understand what the underlying computation is for these filters, without obscuring it with scheduling information.

Sharpen is a 5-point stencil (Figure 7-1) which subtracts pixels in all four directions of the middle pixel from 8 times the value of the middle pixel. Sharpen more and blur more are 9-point stencils. Sharpen more subtracts the values of all 8 neighboring pixels from 12 times the value of the middle pixel (Figure 7-2). Blur more exhibits a slightly complicated kernel (Figure 7-3). Invert is a point-wise kernel which flips the bits of the input image (Figure 7-4).

Threshold filter has an input dependent conditional, where it first checks whether the luminance of a pixel is above a certain threshold (Figure 7-5). If it is above, it makes the output pixel's value equal to 255, else it makes the output 0.

Box blur for radius of 1, adds up the neighboring 8 points and the middle pixel it self and then does a division by reciprocal multiplication on their summation (Figure 7-6). The Halide lifted box blur filter does a brute force summation of the pixel values, but Photoshop's implementation uses a sliding window algorithm which is asymptotically faster.

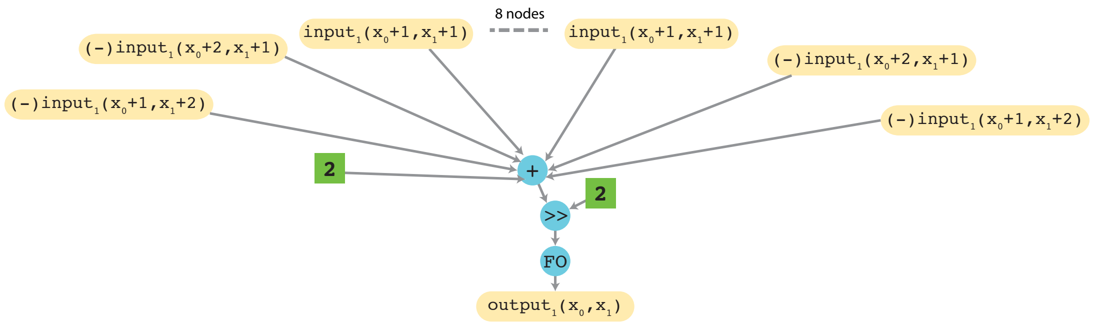
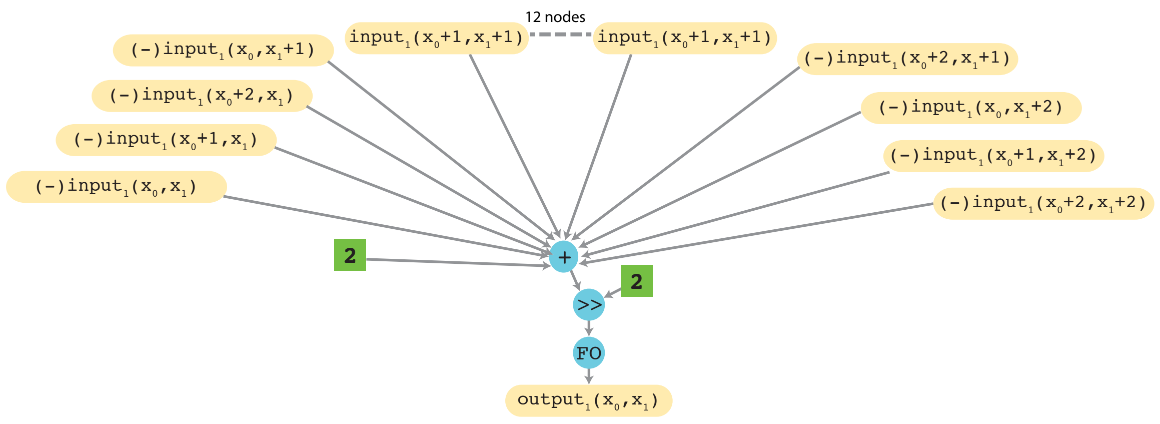

8 nodes
(-)input$_1$(x$_0$+2,x$_1$+1)
input$_1$(x$_0$+1,x$_1$+1)
input$_1$(x$_0$+1,x$_1$+1)
(-)input$_1$(x$_0$+2,x$_1$+1)
(-)input$_1$(x$_0$+1,x$_1$+2)
(-)input$_1$(x$_0$+1,x$_1$+2)
2
+
2
>>
FO
output$_1$(x$_0$,x$_1$)

Figure 7-1: Symbolic tree for Photoshop sharpen filter.



12 nodes
(−)input$_1$(x$_0$,x$_1$+1)
input$_1$(x$_0$+1,x$_1$+1)
input$_1$(x$_0$+1,x$_1$+1)
(−)input$_1$(x$_0$+2,x$_1$+1)
(−)input$_1$(x$_0$+2,x$_1$)
(−)input$_1$(x$_0$,x$_1$+2)
(−)input$_1$(x$_0$+1,x$_1$)
(−)input$_1$(x$_0$+1,x$_1$+2)
(−)input$_1$(x$_0$,x$_1$)
(−)input$_1$(x$_0$+2,x$_1$+2)
2
+
2
>>
FO
output$_1$(x$_0$,x$_1$)

Figure 7-2: Symbolic tree for Photoshop sharpen more filter.

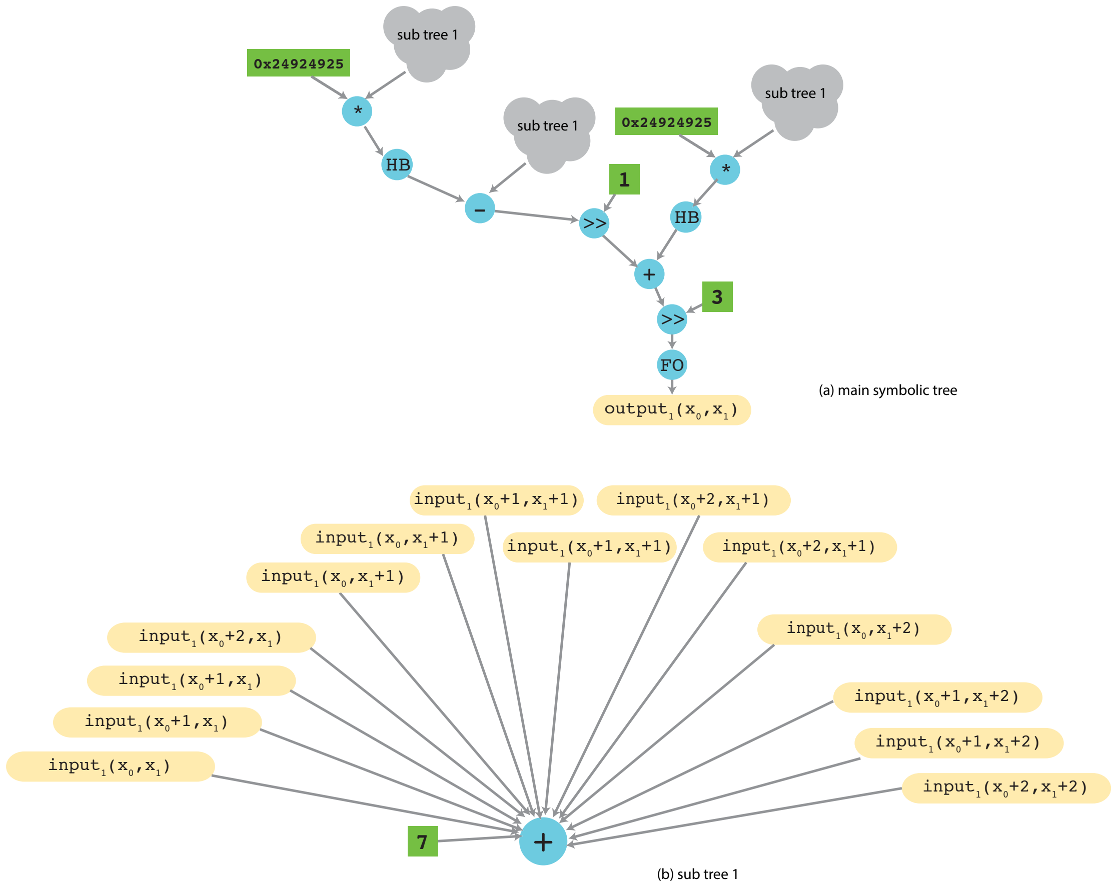


Figure 7-3: Symbolic tree for Photoshop blur more filter.

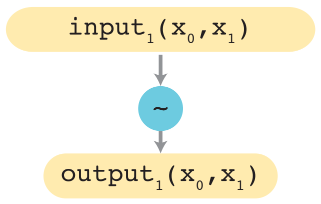


Figure 7-4: Symbolic tree for Photoshop invert filter.

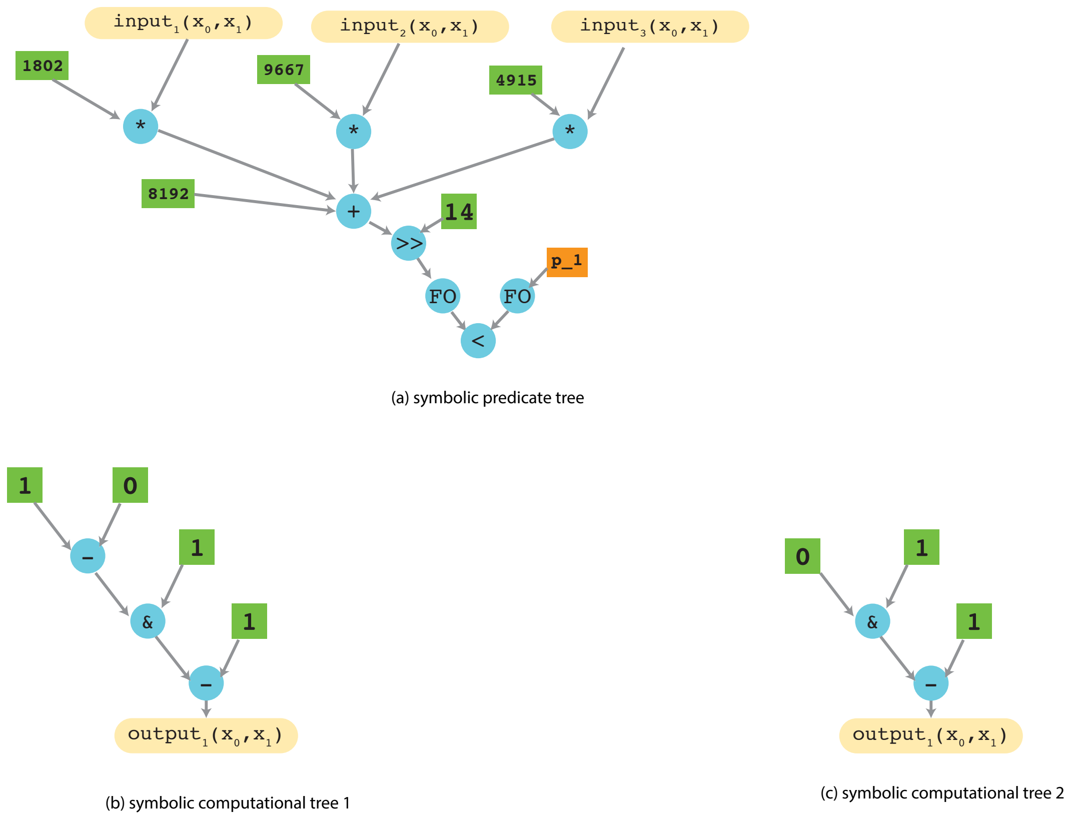


Figure 7-5: Symbolic trees for Photoshop threshold filter.

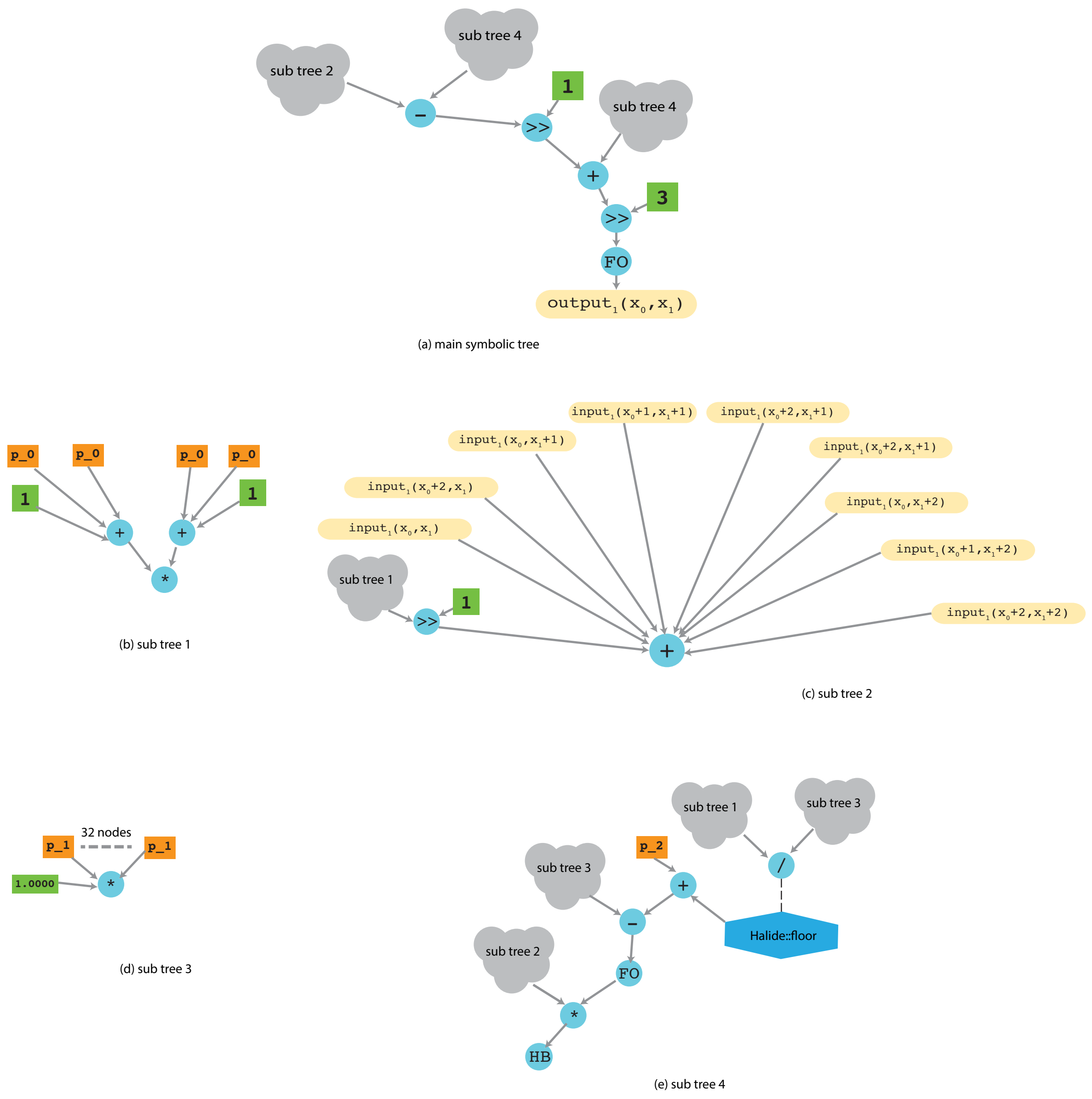


Figure 7-6: Symbolic tree for Photoshop box blur (radius 1) filter. The subtree abstractions are given only for clarity, Helium does not perform analysis to break up the tree into common subtrees.

7.2 Generated Halide Code for Photoshop Filters

Halide code for the six filters from Section 7.1 is given here. The code is refactored from the originally lifted code to improve its readability.

```
#include <Halide.h>
#include <vector>
using namespace std;
using namespace Halide;

int main(){

    Var x_0;
    Var x_1;
    ImageParam input_1(UInt(8),2);
    Func output_1;

    Expr output_1_p__0 = (((((((
        (cast<uint32_t>( input_1(x_0+1,x_1+1) )
        + cast<uint32_t>( input_1(x_0+1,x_1+1) )
        + cast<uint32_t>( input_1(x_0+1,x_1+1) )
        + cast<uint32_t>( input_1(x_0+1,x_1+1) )
        + cast<uint32_t>( input_1(x_0+1,x_1+1) )
        + cast<uint32_t>( input_1(x_0+1,x_1+1) )
        + cast<uint32_t>( input_1(x_0+1,x_1+1) )
        + cast<uint32_t>( input_1(x_0+1,x_1+1) ))
        - cast<int32_t>( input_1(x_0+1,x_1+2) ))
        - cast<int32_t>( input_1(x_0+1,x_1) ))
        - cast<int32_t>( input_1(x_0,x_1+1) ))
        - cast<int32_t>( input_1(x_0+2,x_1+1) ))
        + cast<uint32_t>( 2 ))
                >> cast<uint32_t>( 2 )))  ;

    output_1(x_0,x_1) = cast<uint8_t>(clamp(output_1_p__0,0,255));

    vector<Argument> args;
    args.push_back(input_1);
    output_1.compile_to_file("halide_out_0",args);
    return 0;
}
```

Listing 7.1: Halide code for Photoshop sharpen filter

```
#include <Halide.h>
#include <vector>
using namespace std;
using namespace Halide;

int main(){

    Var x_0;
    Var x_1;
    ImageParam input_1(UInt(8),2);
    Func output_1;
    Expr output_1_p__0  = ((((cast<uint32_t>(2 )
                    - cast<int32_t>((input_1(x_0,x_1)))
                    - cast<int32_t>((input_1(x_0+1,x_1) ))
                    - cast<int32_t>((input_1(x_0+2,x_1) ))
                    - cast<int32_t>((input_1(x_0,x_1+1) )) +
                    cast<uint32_t>(input_1(x_0+1,x_1+1) ) +
                    cast<uint32_t>(input_1(x_0+1,x_1+1) ) +
                    cast<uint32_t>(input_1(x_0+1,x_1+1) ) +
                    cast<uint32_t>(input_1(x_0+1,x_1+1) ) +
                    cast<uint32_t>(input_1(x_0+1,x_1+1) ) +
                    cast<uint32_t>(input_1(x_0+1,x_1+1) ) +
                    cast<uint32_t>(input_1(x_0+1,x_1+1) ) +
                    cast<uint32_t>(input_1(x_0+1,x_1+1) ) +
                    cast<uint32_t>(input_1(x_0+1,x_1+1) ) +
                    cast<uint32_t>(input_1(x_0+1,x_1+1) ) +
                    cast<uint32_t>(input_1(x_0+1,x_1+1) ) +
                    cast<uint32_t>(input_1(x_0+1,x_1+1) )
                    - cast<int32_t>((input_1(x_0+2,x_1+1) ))
                    - cast<int32_t>((input_1(x_0,x_1+2) ))
                    - cast<int32_t>((input_1(x_0+1,x_1+2) ))
                    - cast<int32_t>((input_1(x_0+2,x_1+2) )))
                                  >> cast<uint32_t>(2 ))) & 255 ) ;

    output_1(x_0,x_1) = cast<uint8_t>(clamp(output_1_p__0,0,255));
    vector<Argument> args;
    args.push_back(input_1);
    output_1.compile_to_file("halide_out_0",arguments);
    return 0;

}
```

Listing 7.2: Halide code for Photoshop sharpen more filter

```
#include <Halide.h>
#include <vector>
using namespace std;
using namespace Halide;

int main(){

    Var x_0;
    Var x_1;
    Var x_2;
    Var x_3;
    ImageParam input_1(UInt(8),2);
    Func output_3;
    Expr output_3_p__0 =  ((((((
    (7  + cast<uint32_t>(input_1(x_0,x_1) )
        + cast<uint32_t>(input_1(x_0+1,x_1) )
        + cast<uint32_t>(input_1(x_0+1,x_1) )
        + cast<uint32_t>(input_1(x_0+2,x_1) )
        + cast<uint32_t>(input_1(x_0,x_1+1) )
        + cast<uint32_t>(input_1(x_0,x_1+1) )
        + cast<uint32_t>(input_1(x_0+1,x_1+1) )
        + cast<uint32_t>(input_1(x_0+1,x_1+1) )
        + cast<uint32_t>(input_1(x_0+2,x_1+1) )
        + cast<uint32_t>(input_1(x_0+2,x_1+1) )
        + cast<uint32_t>(input_1(x_0,x_1+2) )
        + cast<uint32_t>(input_1(x_0+1,x_1+2) )
        + cast<uint32_t>(input_1(x_0+1,x_1+2) )
        + cast<uint32_t>(input_1(x_0+2,x_1+2) ))

    -  (( (cast<uint64_t>(613566757 ) * cast<uint64_t>(
    (7  + cast<uint32_t>(input_1(x_0,x_1) )
        + cast<uint32_t>(input_1(x_0+1,x_1) )
        + cast<uint32_t>(input_1(x_0+1,x_1) )
        + cast<uint32_t>(input_1(x_0+2,x_1) )
        + cast<uint32_t>(input_1(x_0,x_1+1) )
        + cast<uint32_t>(input_1(x_0,x_1+1) )
        + cast<uint32_t>(input_1(x_0+1,x_1+1) )
        + cast<uint32_t>(input_1(x_0+1,x_1+1) )
        + cast<uint32_t>(input_1(x_0+2,x_1+1) )
        + cast<uint32_t>(input_1(x_0+2,x_1+1) )
        + cast<uint32_t>(input_1(x_0,x_1+2) )
        + cast<uint32_t>(input_1(x_0+1,x_1+2) )
        + cast<uint32_t>(input_1(x_0+1,x_1+2) )
     + cast<uint32_t>(input_1(x_0+2,x_1+2) )))) ) >> ( 32))

    ) >> cast<uint32_t>(1 ))

    +  (( (cast<uint64_t>(613566757 ) * cast<uint64_t>(
    (7   + cast<uint32_t>(input_1(x_0,x_1) )
        + cast<uint32_t>(input_1(x_0+1,x_1) )
        + cast<uint32_t>(input_1(x_0+1,x_1) )
        + cast<uint32_t>(input_1(x_0+2,x_1) )
        + cast<uint32_t>(input_1(x_0,x_1+1) )
        + cast<uint32_t>(input_1(x_0,x_1+1) )
        + cast<uint32_t>(input_1(x_0+1,x_1+1) )
        + cast<uint32_t>(input_1(x_0+1,x_1+1) )
        + cast<uint32_t>(input_1(x_0+2,x_1+1) )
        + cast<uint32_t>(input_1(x_0+2,x_1+1) )
        + cast<uint32_t>(input_1(x_0,x_1+2) )
        + cast<uint32_t>(input_1(x_0+1,x_1+2) )
        + cast<uint32_t>(input_1(x_0+1,x_1+2) )
        + cast<uint32_t>(input_1(x_0+2,x_1+2) )))) ) >> ( 32)))
                             >> cast<uint32_t>(3 )) ) & 255 ) ;

    output_3(x_0,x_1) = cast<uint8_t>( clamp(output_3_p__0,0,255) );
    vector<Argument> arguments;
    arguments.push_back(input_1);
    output_3.compile_to_file("halide_out_0",arguments);
    return 0;
}
```

Listing 7.3: Halide code for Photoshop blur more filter

```
#include <Halide.h>
#include <vector>
using namespace std;
using namespace Halide;
int main(){

    Var x_0;
    Var x_1;
    Var x_2;
    Var x_3;
    ImageParam input_1(UInt(32),2);
    Func output_1;
    Expr output_1_p__0 =  ~ input_1(x_0,x_1) ;
    output_1(x_0,x_1) = cast<uint32_t>( clamp(output_1_p__0,0,65535) );

    vector<Argument> arguments;
    arguments.push_back(input_1);;
    output_1.compile_to_file("halide_out_0",arguments);
    return 0;
}
```

Listing 7.4: Halide code for Photoshop invert filter

```
#include <Halide.h>
#include <vector>
using namespace std;
using namespace Halide;
int main(){

    Var x_0;
    Var x_1;
    Var x_2;
    Var x_3;
    ImageParam input_2(UInt(8),2);
    ImageParam input_1(UInt(8),2);
    ImageParam input_3(UInt(8),2);
    Param<uint8_t> p_1;
    Func inter_2;
    Expr inter_2_p__1 = ((0  & 1 ) - 1 );
    Expr inter_2_p__0 = select((cast<uint32_t>((((
                (8192  +
                 (4915  * cast<uint32_t>(input_2(x_0,x_1) )) +
                 (9667  * cast<uint32_t>(input_1(x_0,x_1) )) +
                 (1802  * cast<uint32_t>(input_3(x_0,x_1) )))
                        >> cast<uint32_t>(14 ))) & 255 ) )
        < cast<uint32_t>((( p_1) & 255 ) )),(((0  - 1 ) & 1 ) - 1 )
                                            ,inter_2_p__1);
    inter_2(x_0,x_1) = cast<uint8_t>(clamp(inter_2_p__0,0,255));

    vector<Argument> arguments;
    arguments.push_back(input_2);
    arguments.push_back(input_1);
    arguments.push_back(input_3);
    arguments.push_back(p_1);
    inter_2.compile_to_file("halide_out_0",arguments);
    return 0;
}
```

Listing 7.5: Halide code for Photoshop threshold filter

```
#include <Halide.h>
#include <vector>
using namespace std;
using namespace Halide;


int main(){

    Var x_0;
    Var x_1;
    Var x_2;
    Var x_3;
    ImageParam input_1(UInt(8),2);
    Param<uint32_t> p_0("p_0");
    Param<double> p_1("p_1");
    Param<double> p_2("p_2");
    Func output_1;
    Expr output_1_p__0 =
    (((((cast<uint64_t>(cast<uint32_t>(((
        Halide::floor(((cast<double>(1.000000 )
            * cast<double>(p_1 )
            * cast<double>(p_1 )
            * cast<double>(p_1 )
            ......   //32 multiplications
            ......
            ......
            * cast<double>(p_1 )
            * cast<double>(p_1 )
            * cast<double>(p_1 )
            * cast<double>(p_1 )
            * cast<double>(p_1 )) /
        cast<double>(((1  + p_0  + p_0 ) * (1  + p_0  + p_0 ))))) +

        cast<double>(p_2) ) -

    (cast<double>(1.000000 )
     * cast<double>(p_1 )
     * cast<double>(p_1 )
     * cast<double>(p_1 )
     ......     //32 multiplications
     ......
     ......
     * cast<double>(p_1 )
     * cast<double>(p_1 )
     * cast<double>(p_1 )
     * cast<double>(p_1 )
     * cast<double>(p_1 )
     * cast<double>(p_1 )
```

```
  * cast<double>(p_1 )))))
  * cast<uint64_t>((((((1  + p_0  + p_0 ) * (1  + p_0  + p_0 ))
                    >> cast<uint32_t>(1 ))

 + cast<uint32_t>(input_1(x_0,x_1))
 + cast<uint32_t>(input_1(x_0+1,x_1))
 + cast<uint32_t>(input_1(x_0+2,x_1))
 + cast<uint32_t>(input_1(x_0,x_1+1))
 + cast<uint32_t>(input_1(x_0+1,x_1+1))
 + cast<uint32_t>(input_1(x_0+2,x_1+1))
 + cast<uint32_t>(input_1(x_0,x_1+2))
 + cast<uint32_t>(input_1(x_0+1,x_1+2))
 + cast<uint32_t>(input_1(x_0+2,x_1+2))))) >> cast<uint32_t>(32))
+
(( - ((cast<uint64_t>(cast<uint32_t>(((
 Halide::floor(((cast<double>(1.000000 )
 * cast<double>(p_1 )
 * cast<double>(p_1 )
 * cast<double>(p_1 )
 * cast<double>(p_1 )
 .....  //32 multiplications
 .....
 .....
 * cast<double>(p_1 )
 * cast<double>(p_1 )
 * cast<double>(p_1 )
 * cast<double>(p_1 )
 * cast<double>(p_1 )) /
 cast<double>(((1  + p_0  + p_0 ) * (1  + p_0  + p_0 ))))) + 1 ) -
 (cast<double>(1.000000 )
 * cast<double>(p_1 )
 * cast<double>(p_1 )
 * cast<double>(p_1 )
 * cast<double>(p_1 )
 * cast<double>(p_1 )
 ......  //32 multiplications
 ......
 ......
 * cast<double>(p_1 )
 * cast<double>(p_1 )
 * cast<double>(p_1 )
 * cast<double>(p_1 )
 * cast<double>(p_1 )))))
   * cast<uint64_t>((((((1  + p_0  + p_0 ) * (1  + p_0  + p_0 ))
                    >> cast<uint32_t>(1 ))

 + cast<uint32_t>(input_1(x_0,x_1) )
 + cast<uint32_t>(input_1(x_0+1,x_1) )
```

```
    + cast<uint32_t>(input_1(x_0+2,x_1) )
    + cast<uint32_t>(input_1(x_0,x_1+1) )
    + cast<uint32_t>(input_1(x_0+1,x_1+1) )
    + cast<uint32_t>(input_1(x_0+2,x_1+1) )
    + cast<uint32_t>(input_1(x_0,x_1+2) )
    + cast<uint32_t>(input_1(x_0+1,x_1+2) )
    + cast<uint32_t>(input_1(x_0+2,x_1+2) )))) >> cast<uint32_t>(32))
+

  ((((((1  + p_0  + p_0 ) * (1  + p_0  + p_0 )) >> cast<uint32_t>(1 ))

    + cast<uint32_t>(input_1(x_0,x_1) )
    + cast<uint32_t>(input_1(x_0+1,x_1) )
    + cast<uint32_t>(input_1(x_0+2,x_1) )
    + cast<uint32_t>(input_1(x_0,x_1+1) )
    + cast<uint32_t>(input_1(x_0+1,x_1+1) )
    + cast<uint32_t>(input_1(x_0+2,x_1+1) )
    + cast<uint32_t>(input_1(x_0,x_1+2) )
    + cast<uint32_t>(input_1(x_0+1,x_1+2) )
    + cast<uint32_t>(input_1(x_0+2,x_1+2) )))))

 >> cast<uint32_t>(1))) >> cast<uint32_t>(-1  + 32  - 28 )) & 255);

    output_1(x_0,x_1) = cast<uint8_t>( clamp(output_1_p__0,0,255) );
    vector<Argument> arguments;
    arguments.push_back(p_0);
    arguments.push_back(p_1);
    arguments.push_back(p_2);
    arguments.push_back(input_1);
    output_1.compile_to_file("halide_out_0",arguments);
    return 0;

}
```

Listing 7.6: Halide code for Photoshop box blur filter

Chapter 8

Related Work

Binary static analysis Phoenix [24], BitBlaze [25], BAP [9], and other tools construct their own low-level IR (*e.g.*, register transfer language (RTL)) from binaries. These low-level IRs allow only limited analysis or low-level transformations.

Other static analysis aims for high-level representations of binaries. Value set analysis [6] is a static analysis that tracks the possible values of pointers and indices to analyze memory access in stripped binaries; instead of a complicated static analysis, Helium recovers buffer structure from actual program behavior captured in traces. SecondWrite [3], [13] decompiles x86 binaries to LLVM IR; while the resulting IR can be optimized and recompiled, this IR is too low-level to get more than minor speedup over existing optimized binaries. [18] uses SecondWrite for automatic parallelization of affine loops, but must analyze existing loop structure and resolve aliasing, while we lift to Halide code expressing only the algorithm. McSema [2] also decompiles x86 to LLVM IR for analysis. The Hex-Rays decompiler [1] decompiles to a C-like pseudocode which cannot be recompiled to binaries. SmartDec [14] is a binary to C++ decompiler that can extract class hierarchies and try/catch blocks; we extract high-level algorithms independent of particular language constructs.

Dynamic translation and instrumentation Binary translation systems like QEMU [7] translate machine code between architectures using RISC-like IR; RevNIC [11] and S2E [12] translate programs from x86 to LLVM IR by running them in QEMU. Dynamic instrumentation systems like Valgrind [21] present a similar RISC-like IR for analysis and instrumentation, then generate machine code for execution. These IRs retain details of the original binary and do not provide enough abstraction for high-level transformations.

Microarchitecture-level dynamic binary optimization Some systems improve the performance of existing binaries through microarchitecture-level optimizations that do not require building IR. Dynamo [5] improves code locality and applies simple optimizations on frequently-executed code. Ubiquitous memory introspection [34] detects frequently-stalling loads and adds prefetch instructions. [19] translates x86 binaries to x86-64, using the additional registers to promote stack variables. We perform much higher-level optimizations on our lifted stencils.

Automatic parallelization Many automatic parallelization systems use dynamic analysis to track data flow to analyze communication to detect parallelization opportunities, but these systems require source code access (often with manual annotations). [28] uses dynamic analysis to track communication across programmer-annotated pipeline boundaries to extract coarse-grained pipeline parallelism. Paralax [29] performs semi-automatic parallelization, using dynamic dependency tracking to suggest programmer annotations (e.g., that a variable is killed). HELIX [10] uses a dynamic loop nesting graph to select a set of loops to parallelize. [30] uses dynamic analysis of control and data dependencies as input to a trained predictor to autoparallelize loops, relying on the user to check correctness.

Pointer and shape analysis Pointer analyses have been written for assembly programs [15]. Shape analyses [31] analyze programs statically to determine properties of heap structures. [33] uses dynamic analysis to identify pointer-chasing that sometimes exhibits strides to aid in placing prefetch instructions. Because we analyze concrete memory traces for stencils, our buffer structure reconstruction and stride inference is indifferent to aliasing and finds regular access patterns.

Chapter 9

Conclusion

9.1 Limitations

Lifting stencils with Helium is not a sound transformation. In practice, Helium's lifted stencils can be compared against the original program on a test suite – validation equivalent to release criteria commonly used in software development. Even if Helium were sound, most stripped binary programs do not come with proofs of correctness, so testing would still be required.

Some of Helium's simplifying assumptions cannot always hold. The current system can only lift stencil computations with few input-dependent conditionals, table lookups and simple repeated updates. Helium cannot lift filters with non-stencil or more complex computation patterns. Because high performance kernels repeatedly apply the same computation to large amounts of data, Helium assumes the program input will exercise both branches of all input-dependent conditionals. For those few stencils with complex input-dependent control flow, the user must craft an input to cover all branches for Helium to successfully lift the stencil.

Helium is only able to find symbolic trees for stencils whose tree shape is constant. For trees whose shape varies based on a parameter (for example, box blur), Helium can extract code for individual values of the parameter, but the resulting code is not generic across parameters.

Helium assumes all index functions are affine, so kernels with more complex access functions such as radial indexing cannot be recognized by Helium.

By design, Helium only captures computations derived from the input data. Some stencils compute weights or lookup tables from parameters; Helium will capture the application of those tables to the input, but will not capture table computation.

9.2 Future Work

Helium could be extended to support non-linear index accesses with the aid of program synthesis techniques. Using recipes written by a domain expert for common non-linearities that can be seen in common stencil computations, we could define a grammar of such computations from which we will synthesize non-linear expressions (e.g., coordinate transformations). We will use input-output examples to aid the synthesis task. Then, Helium would be able to capture filters such as warp, pointillize, radial blur etc. which involve non-linear coordinate transformations.

We would like to extend Helium to capture parts of filter computations that do not directly rely on the input data. This could be done by extending Helium to localize more than a single function and by establishing the dependency order of such localized functions.

We also hope to automate the manual replacement of Photoshop filters (Section 6.5) by observing the internal calling convention of used by Photoshop. Also, we hope to apply Helium beyond image processing filters to other applications which exhibit stencil computations.

9.3 Conclusion

Most legacy high-performance applications exhibit bit rot during the useful lifetime of the application. We can no longer rely on Moore's Law to provide transparent performance improvements from clock speed scaling, but at the same time modern hardware provides ample opportunities to substantially improve performance of legacy programs. To rejuvenate these programs, we need high-level, easily-optimizable representations of their algorithms. However, high-performance kernels in these applications have been heavily optimized for a bygone era, resulting in complex source code and executables, even though the underlying algorithms are mostly very simple. Current state-of-the-art techniques are not capable of extracting the simple algorithms from these highly optimized programs. We believe that fully dynamic techniques, introduced in Helium, are a promising direction for lifting important computations into higher-level representations and rejuvenating legacy applications.

Helium source code is available at `http://projects.csail.mit.edu/helium`.

Appendix A

Sample DynamoRIO Client Outputs

A.1 Profiling Client Sample Output

The output format for profiling information dumped by the client is as follows.

`<start address, size, frequency, is call out, is return, is call target,`

`{predecessor BBs}, {successor BBs}, {callers}, {callees}>`

For each of `predecessor BBs, successor BBs, callers` and `callees`, the client outputs the number of such elements and execution frequencies of each. Note that `start address` is relative to the starting point of the module in which the basic block resides.

Following is an excerpt from the profile client output for the Photoshop blur filter.

```
1436660,7,2,0,0,1,2,67ce50,1,1436660,1,0,1,67ce50,1,0,
1437e30,7,2,0,1,1,2,67d573,1,1437e30,1,0,1,67d573,1,0,
1459800,63,8,0,0,1,3,dd2f69,3,1459800,4,dea79d,1,0,1,dd2f69,3,0,
145983f,2,4,0,0,0,1,1459800,4,0,0,0,
1459841,27,952,0,0,0,2,1459924,476,1459841,476,0,0,0,
1459844,24,8,0,0,0,2,145983f,4,1459844,4,0,0,0,
145985c,139,960,0,0,0,3,1459844,4,145985c,480,1459841,476,0,0,0,
1459860,135,45120,0,0,0,2,145985c,480,1459860,44640,0,0,0,
14598e7,10,960,0,0,0,2,1459860,480,14598e7,480,0,0,0,
1459924,27,960,0,0,0,2,14598e7,480,1459924,480,0,0,0,
145993f,7,8,0,1,0,2,1459924,4,145993f,4,0,0,0,
149faf0,9,2,0,0,1,2,abf20d,1,149faf0,1,0,1,abf20d,1,0,
```

A.2 Memory Trace Client Sample Output

The output format for memory trace client is as follows. For each memory access following is recorded.

```
<instr address, read(0)/write(1), access width, memory address>
```

Following is an excerpt from the memory trace client output for the Photoshop blur filter.

```
e19f7,0,4,0x0302d000
e19f7,0,4,0x0302d000
e19f7,0,4,0x0302d000
126b70,0,4,0x08755ac2
126b74,1,4,0x08755acc
126b78,0,4,0x08755abe
126b7c,1,4,0x08755ac8
126b80,0,4,0x08755aba
126b84,1,4,0x08755ac4
126b93,0,4,0x5e826ba4
4b487c,0,4,0x0bccf6b0
4b4880,0,4,0x01ff2bdc
4b487c,0,4,0x0bccf6b0
4b4880,0,4,0x01ff2bdc
```

Appendix B

Assembly Code for Various Halide Schedules

In this section, we present excerpts from portion of assembly performing the filter computation for the 9-point blur filter written in Halide (Code 6.1). The first excerpt is when Halide code is compiled without any schedule and the second is when the filter is tiled. This shows how the code becomes clobbered with scheduling information with the introduction of various optimizations, even though the same algorithm is used for each.

```
movzx  edx, byte ptr [esi+ecx+0x02]
movzx  ebx, byte ptr [esi+ecx+0x01]
movzx  ecx, byte ptr [esi+ecx]
add    ebx, edx
add    ebx, ecx
mov    ecx, dword ptr [ebp-0x2c]
lea    ecx, [ecx+eax]
movzx  edx, byte ptr [esi+ecx+0x02]
movzx  edi, byte ptr [esi+ecx+0x01]
movzx  ecx, byte ptr [esi+ecx]
add    edi, edx
add    edi, ecx
mov    ecx, dword ptr [ebp-0x28]
lea    ecx, [ecx+eax]
movzx  edx, byte ptr [esi+ecx+0x02]
mov    dword ptr [ebp-0x1c], edx
```

```
movzx  edx, byte ptr [esi+ecx+0x01]
add    edx, dword ptr [ebp-0x1c]
movzx  ecx, byte ptr [esi+ecx]
add    edx, ecx
movzx  ecx, bx
imul   ecx, ecx, 0x0000aaab
shr    ecx, 0x11
movzx  edi, di
imul   edi, edi, 0x0000aaab
shr    edi, 0x11
add    edi, ecx
movzx  ecx, dx
imul   ecx, ecx, 0x0000aaab
shr    ecx, 0x11
add    edi, ecx
movzx  ecx, di
imul   ecx, ecx, 0x0000aaab
shr    ecx, 0x11
```

Listing B.1: Halide assembly excerpt for the 9-point blur stencil with no schdule

```
movzx  esi, byte ptr [edi+eax]
mov    eax, dword ptr [ebp-0x70]
lea    eax, [eax+ebx]
mov    dword ptr [ebp-0x24], eax
movzx  eax, byte ptr [edi+eax]
mov    ecx, dword ptr [ebp-0x7c]
lea    edx, [ecx+ebx]
movzx  edx, byte ptr [edi+edx]
add    eax, esi
add    eax, edx
mov    dword ptr [ebp-0x34], eax
mov    ecx, dword ptr [ebp-0x64]
lea    eax, [ecx+ebx]
mov    dword ptr [ebp-0x30], eax
movzx  edx, byte ptr [edi+eax]
mov    ecx, dword ptr [ebp-0x4c]
lea    eax, [ecx+ebx]
mov    dword ptr [ebp-0x20], eax
```

```
movzx  esi, byte ptr [edi+eax]
add    esi, edx
mov    ecx, dword ptr [ebp-0x6c]
lea    edx, [ecx+ebx]
movzx  edx, byte ptr [edi+edx]
add    esi, edx
mov    ecx, dword ptr [ebp-0x40]
lea    eax, [ecx+ebx]
mov    dword ptr [ebp-0x1c], eax
mov    edx, ebx
movzx  ebx, byte ptr [edi+eax]
mov    ecx, dword ptr [ebp-0x3c]
lea    eax, [ecx+edx]
mov    dword ptr [ebp-0x14], eax
movzx  ecx, byte ptr [edi+eax]
add    ecx, ebx
mov    ebx, dword ptr [ebp-0x48]
lea    ebx, [ebx+edx]
movzx  ebx, byte ptr [edi+ebx]
add    ecx, ebx
mov    eax, dword ptr [ebp-0x34]
movzx  eax, ax
imul   eax, eax, 0x0000aaab
shr    eax, 0x11
movzx  esi, si
imul   esi, esi, 0x0000aaab
shr    esi, 0x11
add    esi, eax
movzx  eax, cx
imul   eax, eax, 0x0000aaab
shr    eax, 0x11
add    esi, eax
mov    eax, dword ptr [ebp-0x38]
mov    ecx, dword ptr [ebp-0x10]
lea    eax, [eax+ecx]
movzx  ecx, si
imul   ecx, ecx, 0x0000aaab
shr    ecx, 0x11
mov    esi, dword ptr [ebp-0x50]
mov    byte ptr [esi+eax], cl
mov    eax, dword ptr [ebp-0x78]
lea    eax, [eax+edx]
movzx  eax, byte ptr [edi+eax]
mov    ecx, dword ptr [ebp-0x28]
movzx  ecx, byte ptr [edi+ecx]
add    ecx, eax
mov    eax, dword ptr [ebp-0x24]
movzx  eax, byte ptr [edi+eax]
add    ecx, eax
mov    eax, dword ptr [ebp-0x68]
lea    eax, [eax+edx]
mov    ebx, edx
movzx  eax, byte ptr [edi+eax]
mov    edx, dword ptr [ebp-0x30]
movzx  esi, byte ptr [edi+edx]
add    esi, eax
mov    eax, dword ptr [ebp-0x20]
movzx  eax, byte ptr [edi+eax]
add    esi, eax
mov    eax, dword ptr [ebp-0x44]
lea    eax, [eax+ebx]
movzx  eax, byte ptr [edi+eax]
mov    edx, dword ptr [ebp-0x1c]
movzx  edx, byte ptr [edi+edx]
add    edx, eax
mov    eax, dword ptr [ebp-0x14]
movzx  eax, byte ptr [edi+eax]
add    edx, eax
movzx  eax, cx
imul   eax, eax, 0x0000aaab
shr    eax, 0x11
movzx  ecx, si
imul   ecx, ecx, 0x0000aaab
shr    ecx, 0x11
add    ecx, eax
movzx  eax, dx
imul   eax, eax, 0x0000aaab
shr    eax, 0x11
```

```
add    ecx, eax
movzx  eax, cx
imul   eax, eax, 0x0000aaab
shr    eax, 0x11
mov    ecx, dword ptr [ebp-0x00000084]
mov    esi, dword ptr [ebp-0x10]
lea    ecx, [ecx+esi]
mov    edx, dword ptr [ebp-0x50]
mov    byte ptr [edx+ecx], al
```

Listing B.2: Halide assembly excerpt for the 9-point blur stencil with tiling

Bibliography

[1] Idapro, hexrays.

[2] Mcsema: Static translation of x86 into llvm. 2014.

[3] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 295–308, New York, NY, USA, 2013. ACM.

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014.

[5] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM.

[6] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 5–23. Springer Berlin Heidelberg, 2004.

[7] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[8] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 133–144, New York, NY, USA, 2012. ACM.

[9] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. HELIX: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93, New York, NY, USA, 2012. ACM.

[11] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with RevNIC. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 167–180, New York, NY, USA, 2010. ACM.

[12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265–278, New York, NY, USA, 2011. ACM.

[13] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 51–60, New York, NY, USA, 2013. ACM.

[14] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. Smartdec: Approaching c++ decompilation. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, pages 347–356, Washington, DC, USA, 2011. IEEE Computer Society.

[15] B. Guo, M.J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D.I. August. Practical and accurate low-level pointer analysis. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 291–302, March 2005.

[16] R. Nigel Horspool and Nenad Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[17] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[18] Aparna Kotha, Kapil Anand, Matthew Smithson, Greeshma Yellareddy, and Rajeev Barua. Automatic parallelization in a binary rewriter. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 547–557, Washington, DC, USA, 2010. IEEE Computer Society.

[19] Jianjun Li, Chenggang Wu, and Wei-Chung Hsu. Dynamic register promotion of stack variables. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 21–31, Washington, DC, USA, 2011. IEEE Computer Society.

[20] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. Helium: Lifting High-Performance Stencil Kernels from Stripped x86 Binaries to Halide DSL Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, 2015.

[21] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[22] Sylvain Paris. Adobe systems. personal communication, 2014.

[23] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

[24] Microsoft Research. Phoenix compiler and shared source common language infrastructure.

[25] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.

[26] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 65–76, New York, NY, USA, 2014. ACM.

[27] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[28] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[29] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The paralax infrastructure: Automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 389–400, New York, NY, USA, 2010. ACM.

[30] Zheng Wang, Georgios Tournavitis, Björn Franke, and Michael F. P. O'boyle. Integrating profile-driven parallelism detection and machine-learning-based mapping. *ACM Trans. Archit. Code Optim.*, 11(1):2:1–2:26, February 2014.

[31] Reinhard Wilhelm, Mooly Sagiv, and Thomas Reps. Shape analysis. In DavidA. Watt, editor, *Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2000.

[32] Samuel Williams, Dhiraj D Kalamkar, Amik Singh, Anand M Deshpande, Brian Van Straalen, Mikhail Smelyanskiy, Ann Almgren, Pradeep Dubey, John Shalf, and Leonid Oliker. Optimization of geometric multigrid for emerging multi-and manycore processors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 96. IEEE Computer Society Press, 2012.

[33] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 210–221, New York, NY, USA, 2002. ACM.

[34] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, and Weng-Fai Wong. Ubiquitous memory introspection. In *International Symposium on Code Generation and Optimization*, San Jose, CA, Mar 2007.