High-Performance Image Processing Using Special-Purpose CPU Instructions: The UltraSPARC Visual Instruction Set

by Daniel S. Rice

Master's Report
under the direction of Carlo Séquin

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley

March 19, 1996

Abstract

The UltraSPARC-I processor implements, in addition to the SPARC v9 instruction set, a set of new instructions that accelerate image and video processing – the visual instruction set, or VIS. These instructions address a number of areas in which traditional instructions perform poorly for these highly parallel tasks. Although these instructions support a wide variety of functions, they represent far less implementation effort than that needed to design dedicated imaging hardware because they leverage the design efforts of the CPU and memory system, and will continue to provide performance improvements as the processor speed is increased.

Unlike traditional CPU features, the performance benefits of such instructions have not been quantified. We attempt to demonstrate the performance effects of the VIS instructions in the context of typical image processing loops.

For the greatest benefit, these instructions must be used with an eye to maximizing various forms of parallelism, including superscalar instruction issue, loop vectorization, and pipelining in both hardware and software. Currently much of this work must be done by hand. We propose some ways to automate portions of this process and describe some of the existing tools.

Contents

List of Figures

List of Tables

1 Introduction

Over the last year, the author and others at the Sun Microsystems Computer Company (SMCC) have developed high-performance image processing software for a new generation SPARC CPU – UltraSPARC-I [SME95a], found in the Sun Ultra 1 and 2 systems at the time of this writing. UltraSPARC-I is the first in a projected line of processors implementing the SPARC v9 instruction set [SPARC94]. A notable feature of this processor is its support for some of the common operations found in image and video processing algorithms through the addition a set of new instructions that extend the SPARC v9 specification. These instructions are documented and available to developers via a kit bundled with SunPro's SPARCWorks 4.0 compiler [SME95b].

These instructions, known collectively as the UltraSPARC visual instruction set, or simply "VIS," offer the potential for high-performance software implementations of many tasks that previously required additional hardware. Functions such as MPEG [MPEG93] video decoding can thus be made available as a standard workstation feature, greatly extending the number of users to whom they are accessible. It is hoped that this in turn will provide an incentive for software vendors to integrate images and video into their applications. As of this writing Intel has announced plans to add multimedia instructions to its processors (see section 2.6). Other vendors such as MIPS are also rumored to have similar plans. Hewlett-Packard has already demonstrated a real-time software MPEG decoder using several special instructions added to the PA-RISC architecture [Lee95] (see section 2.5).

The introduction of a complex instruction set extension raises many questions. On the practical side, there are the problems of generating code that uses the instructions efficiently and without undue programming effort. Tools such as compilers, debuggers, simulators, and performance analyzers must be informed of the new instructions and their properties. The operating system must ensure that any state associated with the instructions is properly initialized, context switched, and multithreaded. The interaction of the new instructions and existing methods and tools, such as compiler optimizations, must be understood. As old bottlenecks are addressed by the new instructions, new ones will come to light and must be identified and dealt with.

On the philosophical side, questions may be raised as to the viability of such an approach. How will compatibility be maintained in a world of incompatible extensions? Will the hardware trade-offs of today's processors that led to the definition of the new instructions remain constant over the next few processor designs? Can special purpose instructions coexist with the RISC philosophy of simple instruction sets, or do they represent a return to the CISC chips of previous decades?

In terms of performance, VIS has been a great success, although its success in the marketplace remains to be seen. An enhanced CPU that remains mindful of RISC design principles has the potential to compete with dedicated circuit designs without compromising overall system performance. Since the enhancements are an integral part of the CPU, the benefits of advanced process technology and high-bandwidth links to memory are leveraged from the existing CPU design. In addition, today's compiler optimization techniques are capable of dealing with these enhancements, provided they interact reasonably with the rest of the instruction set. Finally, although these enhancements required substantial engineering effort to be integrated into library code, techniques exist to create tools to greatly simplify this process.

In this report, we discuss some of the problems facing the writer of high-performance image processing code, and some solutions available in historical and current microprocessor hardware. The visual instruction set is outlined in detail, and its contributions to performance are analyzed. Compilation techniques which allow access to VIS or similar instruction set extensions are discussed. We look at a handful of practical applications of VIS and compare their performance to that of generic C code. Finally, we look at future enhancements to VIS and some technologies which may help to bridge the gaps between multimedia hardware and conventional programming paradigms.

2 Accelerated Image Processing

The field of image processing is a vast one. It has links to signal processing, computer graphics, robotics, vision, and remote sensing, and its techniques are used in a wide array of application areas. The concern of this report is with the computation of various simple image processing primitives, such as pixel arithmetic, convolution, and resampling. Of course, for many applications the results of such operations must still be subjected to further processing, resulting in an interpretation of the data. We will not discuss such interpretation; it is highly application-specific and not necessarily amenable to hardware acceleration. We will use the term *imaging* to differentiate our more limited task from such interpretive processing.

Routines to perform many of these imaging tasks have been encapsulated into standard libraries, such as SunSoft's XIL [SunSoft94] and SGI's ImageVision library (IL) [SGI93]. These libraries offer an assortment of primitives for, *inter alia*, arithmetic on pixels, lookup table operations, resampling and convolution, and display. Both allow hardware-specific reimplementations of their primitives. There is currently no significant cross-platform imaging library standard, although interest in establishing one has been growing.

Many imaging applications, such as Adobe Photoshop, also allow developers to "patch" their internal functions with high-performance versions. Generally these functions are written for maximum performance on a given hardware platform without regard for portability; hardware designers have thus felt free to introduce incompatible upgrades as implementation technologies and trade-offs change.

In this section we discuss some of the recent hardware solutions that have been implemented for the purpose of speeding up imaging tasks and the trade-offs associated with various approaches. We look briefly at two multimedia-enhanced processors, the Intel i860 and the Hewlett-Packard PA-7100LC, which prefigure UltraSPARC's use of VIS, and take note of some of the features of Intel's newly announced MMX enhancements. We then discuss a few of the positive and negative performance features of UltraSPARC and SPARC generally; an understanding of these features is necessary in order to be able to take full advantage of VIS. The section concludes with a formula for estimating VIS performance and a discussion of benchmarking in the context of the design of VIS.

2.1 Dedicated Imaging Hardware

Support for imaging has recently become an integral part of high performance desktop workstations. The recent standardization of compressed motion video (e.g., motion JPEG, MPEG-1, MPEG-2, H.261, and H.263), the proliferation of image capture devices such as scanners and digital cameras, and the ubiquity of high-bandwidth networks for real-time image transmission have converged to produce a demand for fast image processing on general-purpose computers. Because of the real-time demands of video, there has been an emphasis on special-purpose hardware solutions. Even non-video applications have had significant hardware resources applied to them, as in the large market for third party accelerators for various popular image manipulation applications.

Because of the extreme parallelism of most imaging tasks, special purpose hardware has some obvious strengths. It can perform operations on multiple pixels simultaneously, its ALUs can be designed to work at exactly the required precision, and it can be tailored exactly to the application at hand. For example, MPEG decoder hardware may have dedicated circuitry for Huffman decoding, motion compensation, and color conversion, as well as local memory for IDCT tables and inverse quantization matrices. As MPEG grows in popularity, such hardware will become an ever cheaper commodity.

Most imaging tasks, however, are not as clearly circumscribed as MPEG decoding. For more general imaging tasks, the variety of operations to be performed demands some hardware generality. The imaging coprocessor begins to resemble a general-purpose CPU: it has one or more ALUs, control logic, instruction fetch and decoding, and memory access ports.

Another class of imaging accelerators use multiple custom or commodity processors and divide the work between them. This approach is ultimately limited by the available bandwidth to memory, particularly if the processors reside on an expansion bus.

2.2 Imaging In The Graphics Accelerator

Some imaging functions may share hardware with three-dimensional graphics hardware. Such hardware typically does not write to main memory, but rather maintains its own memory buffers that are scanned by the video hardware to form the screen display. For example, texture mapping hardware typically performs bilinear interpolation, which may be used directly as an image resizing operation when the output is to be displayed. In fact any perspective transformation, including affine transformations such as rotation and skew, may be realized as a special case of texture mapping. If texturing hardware is built with some flexibility, it may be possible to write microcode to perform other operations that can be implemented in terms of fixed-point addition and multiplication, such as convolution. Logical operations may be built on top of existing hardware for window-system support. The Silicon Graphics ImageVision library accelerates the aforementioned operations on some SGI graphics devices, particularly the Reality Engine with its real-time texture-mapping capabilities. Linear color conversion is occasionally included as part of a frame buffer, in effect simply extending the set of input datatypes it accepts.

One difficulty with frame buffer-based approaches to imaging is the requirement that results be placed in frame buffer memory. Even if the output is not to be displayed, it may be possible to direct the hardware to output to a non-visible portion of its memory that may then be operated on further. However, copying from such a buffer into main memory will typically be slower than copying between main memory buffers since this is not an important path for most graphics applications, although the X window system does require that such a path be available. Also, frame buffer memory is a limited, non-scalable resource, and is typically significantly more expensive than ordinary DRAM. Multiple processes and especially multiprocessor systems will have to cope with contention for the available space.

2.3 Interfacing With The Cache

For some time, processor speeds have increased significantly faster than those of DRAM. This has resulted in a substantial gap between access times for dedicated SRAM memories, such as CPU registers, and main memory access times. Modern computers depend heavily on the presence of data caches, both on-chip (*internal*, or *L1 cache*) and off (*external*, or *L2 cache*), to bridge this gap. A subset of main memory locations is mirrored in a smaller, faster memory bank, allowing the CPU to access and alter the contents of these locations in a fraction of the full DRAM access time.

The principal fact that makes caching a valid strategy for most programs is *locality*. This is the observation that references to memory tend to involve a relatively small number of addresses (the *working set*); this set of addresses tends to change fairly slowly over time.

The contents of the caches are address-value pairs. For each incoming address, it must be determined whether it is present in cache, and its corresponding value returned or altered. In order to avoid the need for a fully associative comparison of all address tags against the incoming address, some intermediate bits of the address are used to select a cache *line*; this line holds a number of address-value pairs, where only the bits of the address above those used for line selection need actually be stored. An associative comparison of these address bits yields the desired data, or the information that the data are not to be found in cache.

Image processing programs, however, like many scientific programs, do not have a conveniently small working set. Each operation may be performed over a large (multiple megabyte) span of memory, with no address being revisited once read. In this case, the usual caching strategies may behave quite poorly. Image pixels, which are read once, will tend to overwrite other data such as lookup tables and constants that are read repeatedly, since the image data spans a large range of addresses, and thus will be scattered across the entire cache. For this case, a streaming behavior is desirable, in which the cache serves only as a buffer between main memory and the CPU. Tiled image representations in which images are stored as a set of rectangular subpieces can improve cache reuse dramatically, but tend to require a series of operations to be performed over a given region for maximum benefit.

Cache *aliasing* occurs when multiple data sources repeatedly map to the same cache lines. This is particularly likely when data are accessed using pointers to addresses with a constant spacing between them. When this spacing lies near various powers of two or multiples of the cache size, cache lines filled only recently with data read from one of the pointers may be overwritten with data from the other pointer. When the first pointer is again read from, it will incur another cache miss, and so on. Aliasing can usually be avoided by allocating images and other data at addresses with pseudorandom intermediate bits, i.e., not all starting on page boundaries, as in the current implementation of XIL. Our XIL timings (section 8) show some poor cache behavior that is due to this design choice. The operating system attempts to help in this regard by means of a technique known as *page coloring*, in which physical pages are assigned to each virtual page in such a way as to increase the randomness of the intermediate bits of the physical pointers.

This mismatch between the caching needs of general programs and imaging programs suggests that imaging hardware might do well to avoid the cache entirely. By accessing main memory directly, a device may implement its own strategies to cover the memory latency, such as a stream buffer or a vectorized load. Since the CPU may also require access to the same memory addresses, they must not appear in the CPU cache; otherwise the CPU would be able to see invalidated data values. This is the approach taken, for example, by Sun's SX imaging coprocessor [Donovan95], found in the SPARCStation 10 and 20 workstations. It consists essentially of an ALU and register set attached to the system memory controller; it can access uncached regions of DRAM and VRAM directly using a wide variety of load and store operations and operate on them using vector-like instructions. The need to remove segments of memory from cache introduces a difficult trade-off, however. If an application wishes to perform a mix of SX-accelerated and unaccelerated operations on a given image, the image must be flushed from cache each time an SX routine is to operate on it, and gradually brought back into cache as its addresses are referenced by the CPU once again (there are also additional overheads involved in remapping the virtual address space during these transitions). This overhead can make use of the SX impractical for some tasks, and imposes a heavy burden on the writers of SX library or application code to cover all possible operations. It may be necessary to write SX code that actually underperforms the CPU on a single operation, simply to avoid the need to process image data using both SX and non-SX code. These trade-offs continue to add greatly to the difficulties of maintaining the body of existing SX code.

2.4 The Intel i860

No direct successor to the SX coprocessor exists; instead, Sun chose to embed imaging and graphics instructions into their first SPARC v9 compliant CPU, UltraSPARC-I. This approach has roots in the Intel i860 Microprocessor [Intel89], which contains special instructions to support shading and Z-buffered hidden surface elimination. The chief architect of VIS, Leslie Kohn, was also one of the designers of the i860.

The i860 instruction set includes partitioned arithmetic instructions that perform up to four simultaneous fixed-point additions on data in a 64-bit floating point register. Also present are partitioned comparison instructions that allow interpolated depth values to be compared against previously computed Z-buffer values. This comparison results in a bitmask that may be used to control the storage of pixel data. The resulting shaded pixels must be combined in a special merge register before they may be stored.

The i860 also offers a dual-instruction mode, in which a 64-bit instruction word containing both a floating point or graphics instruction and a "core" instruction may be issued atomically. This offers some of the benefits of superscalar instruction issue, such as the ability to perform vectorized floating point operations by loading new data values while operating on the existing ones. Using a combined multiplication/addition operator, it is possible to perform operations such as dot products at full FPU bandwidth. These dual instructions are encoded differently from an equivalent pair of separate instructions, and so must be identified statically.

The pipelining of the i860's floating point units is visible to the programmer; results are written back to the destination register two or three cycles after instruction initiation. Although UltraSPARC-I provides interlocks to prevent access to registers scheduled for writeback, we shall see that observing the pipeline latencies in software remains advantageous. This is not to say that interlocks should be removed; they are essential for binary compatibility between members of an architectural family as timings (invariably) change. Although the i860 offers an interlocked execution mode, it essentially eliminates all the benefits of pipelining and is not suitable for high-performance code.

2.5 The Hewlett-Packard PA-7100LC

Lee [Lee95] describes a set of multimedia enhancements added to the PA-7100LC, a processor used in HP's relatively low-end workstations, to support real-time MPEG-1 decompression. The integer ALUs were partitioned to perform two 16-bit additions, subtractions, averages, and shift/add combinations; the latter is used to implement multiplication by constants. Saturation arithmetic, in which overflow values are clamped at the extremes of the pixel range, was also implemented and was successful in reducing the need to check for overflow in software, an important speedup (see section 5.3). The addition of these instructions added only 0.2% to the die area, fitting mostly into previously unused space.

HP's implementation of MPEG decoding benefited from the presence of color conversion hardware on the frame buffer, as well as a special pixel datapath that requires the CPU to write only 8 bits per pixel; an approximation to 24-bit color is generated on the fly. Section 7.7 describes how VIS may be used to accelerate full 24-bit color conversion for video streams.

2.6 The Intel MMX Enhancements

As of this writing, Intel has just announced a set of multimedia extensions to Pentium and later processors, to ship in volume in 1997. Like VIS, MMX operates on partitioned data values of up to 64 bits. Both signed and unsigned datatypes are supported, in both overflowing and saturating arithmetic modes. Source operands may come from either a dedicated set of eight registers, which share space with the floating point register file, or memory.

In addition to the datatypes supported by VIS, MMX allows most of its operations to take place directly on 8-bit data. MMX also provides a direct 16×16 multiplication operator, although extracting the bits of interest from the result appears to be slightly more complex than it is in VIS.

The presence of only eight MMX registers would appear to disallow much of the advanced instruction scheduling discussed below. We will also see that other factors such as branch prediction and the degree of superscalarity will have an effect on the amount of speedup attainable through the use of partitioned arithmetic. Because of these factors, it is difficult to predict MMX performance without a deeper understanding of its relationship to the remainder of the Pentium architecture.

A technical overview and reference manual may be found at the URL:

`http://www.intel.com/pc-supp/multimed/mmx`

2.7 The UltraSPARC-I

UltraSPARC-I, designed by Sun Microelectronics (SME) (formerly known as SPARC Technology Business, or STB) and manufactured by Texas Instruments, implements the 64-bit SPARC v9 architecture. The initial chips have clock rates of 143, 167, and 200 MHz (i.e., 5, 6, and 7 nanosecond cycle times), and contain 5.2 million devices, including 16 kilobyte internal data and instruction caches. Only 3% of the die area is devoted to VIS-related gates, and the cycle time is not determined by any of the VIS instructions.

[Greenley95], [Kohn95] and [Zhou95] discuss the design of UltraSPARC and VIS. Several white papers describing various aspects of UltraSPARC are available at:

`http://www.sun.com/sparc/WhitePapersMain.html`

UltraSPARC-I has a number of features intended to support high performance and advanced compiler optimizations, described below in sections 2.7.1 through 2.7.6.

2.7.1 Superscalar Instruction Issue

A *superscalar* processor is one that can issue more than one instruction per cycle. We refer to the collection of instructions issued in a given cycle as sharing an *issue slot*. The processor ensures that the results computed by a program fragment obey *program order* semantics; that is, they are the same as if the instructions had been issued sequentially. This restriction may be relaxed somewhat with respect to the order of bus transactions initiated by the processor, but such differences are visible only from another processor. A processor that issues multiple instructions without program order semantics is probably best thought of as a VLIW (very long instruction word) machine and not a superscalar. In addition to being easy to understand, program order semantics allows for binary compatibility between multiple processors in the same family, an important consideration in the marketplace where the pain of transition between processor generations must be minimized.

UltraSPARC-I issues up to four instructions per cycle. Hardware interlocks prevent pairs of instructions that would cause a hazard from issuing together. For example, the instructions:

```
mul a, b, c
add c, d, e
```

would be prevented from being issued in the same cycle since the result of the `mul` in register `c` will not yet be available for use by the `add` instruction. Some pairs of instructions can be grouped together despite the presence of hazards if an appropriate forwarding datapath exists.

In order for UltraSPARC-I to group four instructions together, the last instruction in the group must be either a branch or a floating point/graphics operation (including most VIS operations). Since integer codes rarely contain enough parallelism to take advantage of 4-scalar issue, this is a reasonable trade-off.

Each instruction dispatched in a given cycle must be sent to a distinct functional unit to be processed. UltraSPARC-I has two integer units (symmetrical except for the absence of a shifter in one), a single load/store unit, a branch unit, and two floating point units, one performing additions and the other multiplications. The divide and square root operations are processed by additional, specialized units, but must first pass through the floating-point multiplier. Other processing continues while these side units operate so long as their results are not needed.

2.7.2 Branch Prediction and Following

Up to four instructions per clock are loaded into a 12-entry buffer and partially decoded. These instructions are fetched from the predicted execution path; as long as branches are predicted correctly, the processor will not be starved for instructions.

Branches are sometimes slow on pipelined architectures since the direction of the branch may depend on values still being computed in the pipeline. Branch prediction mitigates this by assuming the branch will be taken or not taken and proceeding without penalty. When the correct branch direction is determined, if it is in accordance with the prediction nothing need be done; if not, all computation after the branch is prevented from altering the machine state, including changes to registers or memory and exceptions of all kinds. If branches can be predicted correctly with high frequency, the extra penalty for backing out of incorrect predictions will be small. UltraSPARC-I can execute up to 18 instructions speculatively.

The prediction of branch direction is based on a classic four-entry state machine. Initially, branches are marked as taken or not taken based on a bit in their encoding which is user-settable (the importance of setting this bit is doubtful, since the first few times the instruction is encountered will probably also involve page faults, cache misses, etc.). If the prediction is correct, the state moves to "strongly taken" or "strongly not taken"; a correct prediction in either of these states does not change the state. An incorrect prediction only moves the state back to "lightly taken" or "lightly not taken." Thus two incorrect predictions in a row are required in general to reverse the sense of the prediction. For the common case of nested loops, this will result in a single incorrect prediction each time the inner loop exits. Only one pair of bits is required for every two instruction cache entries since SPARC does not allow a branch to be followed by another branch in its delay slot. It is estimated that 87% of SPECint92 and 93% of SPECfp92 branches (see section 2.11) are correctly predicted using this mechanism. If the code contains a mixture of easy and hard to predict branches, it may be worthwhile to arrange it so that the hard branches follow the easy ones.

Each set of four instructions in the instruction cache has a "Next Field" associated with it, yielding the cache index of the predicted next group. This index is updated dynamically as branch predictions are evaluated. By storing the predictions in terms of cache lines, instruction prefetching is greatly simplified. In addition, there is logic that can often join instructions preceding and following a branch in the dynamic sense into a single group (depending on memory alignment). This allows a branch to be taken every cycle with minimal penalty.

The graph in Figure 1 illustrates the effects of data-dependent branching. A simple thresholding loop was timed. It reads bytes from a source array and compares them against a threshold parameter. If the value read is less than the threshold, 0 is written to the corresponding byte of the output array. Otherwise, 255 is written. Uniformly distributed random numbers between 0 and 255 were used as the input data. Thus the branches involved in performing the threshold operation were essentially random, with taken/not taken frequencies proportional to the threshold value. The branch predictor requires runs of branches which are all taken or all not taken, perhaps with occasional single mispredictions. The length of such runs is maximized at threshold values of 0 and 255 and minimized for a threshold of 128.

For each possible threshold value, 1,000 iterations were timed and their times averaged. The input and output arrays were small enough to reside entirely in the on-chip cache. The graph exhibits the expected characteristics of symmetry (due to the symmetry of the prediction mechanism) and a central peak, since a threshold of 128 produces the most unpredictable branches. By contrast, removing the conditional altogether results in a time of roughly 3 clocks/byte. We see that performance on code with data-dependent conditionals is subject to wide variation.

2.7.3 Non-Blocking Load/Store Unit

All processors must establish some policy for dealing with cache misses. Many stall the processor until the data have been read from either a higher level cache, main memory, or a memory-mapped device. This policy would penalize imaging code, which tends to revisit cached data only at relatively long intervals, severely. UltraSPARC-I implements a more lenient policy in which loads are handled asynchronously. As long as the result of the load is not used, processing continues regardless of whether the data were found in cache. The

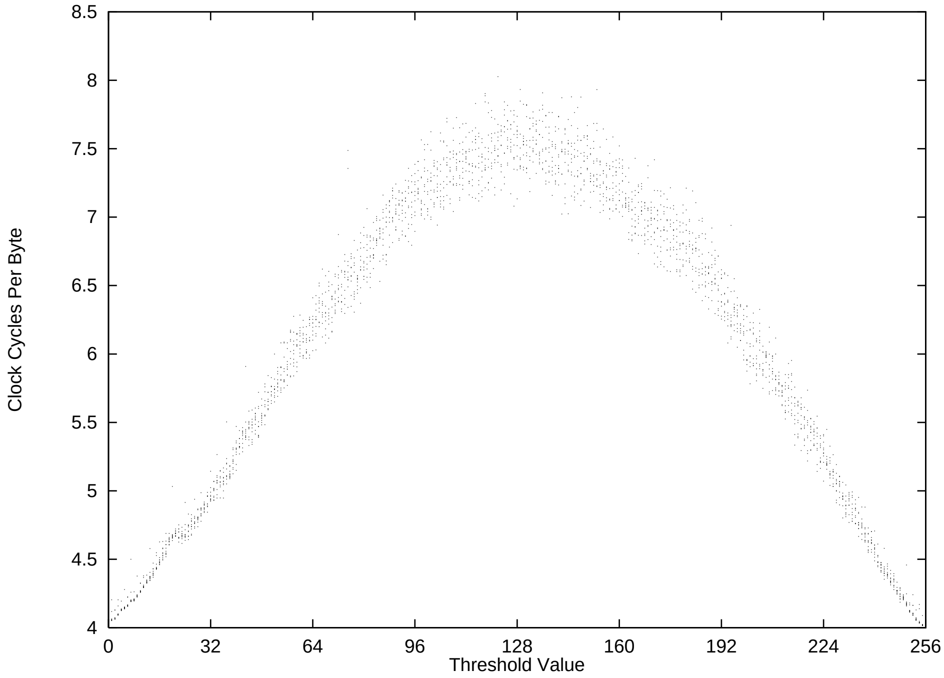

Figure 1: Effects of data dependence on branch timing.

processor still stalls if a value is used before it has been acquired. Also, values are always returned in the order the loads were issued.

By separating the loading of a value from its use, code may be made relatively insensitive to the distribution of data between the various levels of cache. In particular, we may assume an 8-cycle latency between load and use; if the data are found in the on-chip (L1) cache, which has a 3-cycle latency, this is overkill, but if the data are found in the off-chip (L2) cache they will be returned just in time. The full 30-cycle latency to main memory is still very difficult to cover completely, but at any rate the number of stall cycles may be lessened by this strategy.

2.7.4 Automatic Store Compression

As with loads, store execution is decoupled from the main processor pipelines. Processing continues regardless of whether or not the store requires a cache writeback and/or allocation. In addition, multiple stores to adjacent addresses sharing a cache line are combined on-chip and sent to the cache subsystem in a single transaction. Another benefit of store compression is the ability to group loads with loads and stores with stores on the memory bus, avoiding extra bus turnaround penalties.

2.7.5 Non-Faulting (Speculative) Loads

Modern optimization techniques sometimes require the ability to perform a load *speculatively*, that is, without regard to whether it accesses a legal address. The classic example is optimizing a guarded load:

```
if (ptr != NULL)
  value = *ptr;
```

In order to increase the parallelism of the code, the implicit serialization implied by such a test must be eliminated. However, it is not safe to simply dereference `ptr` on most architectures since a null value will cause a segmentation violation. The non-faulting load variants allow loads to any address to proceed, returning zero if the load cannot be completed normally without generating an exception.

For integer codes in one simulation, the average size of instruction groups was increased by 33% when loads were allowed to be moved past a single branch. For floating point codes, the size increase was 18%.

In the case of imaging code, non-faulting loads may also be used in circumstances where data are to be read from addresses slightly outside image boundaries. As we will see, these reads are artifacts of the need to work with aligned pointers, and the values they return are of no consequence. The use of non-faulting loads allows such code to be written straightforwardly, without concern for pathological cases in which page boundaries are violated. Since the use of non-faulting loads can easily mask the presence of real bugs, they should be used sparingly in user code. Non-faulting loads may be accessed from C using techniques described in section 6.1.

A speculative load to an invalid page will still cause a trap, but the trap handler will exit as soon as the offending instruction is identified as non-faulting. The price of these traps may be acceptable as they occur only at the beginning and end of some images, and not at all for most images.

2.7.6 Conditional Move Instructions

Another technique for increasing parallelism is the elimination of conditionals. This increases the sizes of basic blocks (section 6.4.1), which are amenable to the most aggressive scheduling techniques. The conditional move, new to SPARC in v9, replaces constructs such as:

```
if (x > 0)
  y = z;
```

with a comparison on `x` that sets an appropriate condition code, and a conditional move that copies the value of `z` into `y` if the condition code is true. Conditional moves can also be used to implement `min` and `max` operators, which in turn are highly useful for keeping array references within bounds. For example, a texture-mapping implementation may wish to repeat the edge values of the texture map if the texture coordinates go out of bounds (see section 7.8).

2.8 Some UltraSPARC-I Caveats

UltraSPARC-I also has shortcomings in a number of areas. In no particular order, they include:

- Multi-cycle integer multiplication.

  The integer multiplier is not pipelined; it consumes 2 bits from its second argument per cycle, stalling other processing during its execution. It has an early exit feature to allow processing to continue as soon as the second argument becomes all zero (or all ones if performing a signed multiplication); the maximum delay is 18 clocks. In practice, most multiplications in integer SPEC code involve either constants or very small values; the former can be rewritten by the compiler as a series of shifts and logical "and"s. Unfortunately, fixed point arithmetic uses a large number of bits throughout the argument range. This difficulty is more than compensated for by the power of the VIS fixed-point operations, but occasionally some non-VIS multiplications can become a bottleneck. It is advisable to use the floating point multiplier where possible since it is fully pipelined. Legacy imaging codes, which make heavy use of the integer multiplier, will suffer substantially from the lack of pipelining.

If an integer multiplication must be used, it may be advantageous to shift the multiplier right to eliminate any known low-order zero bits, e.g., in the case of multiplication by a scanline stride known to be a multiple of 8. If multiplications are to occur over a limited range of multipliers a table lookup may be preferable.

- Lack of store forwarding.

  A write to a given address followed by a load from that address while the new data is still in the store buffer is not detected and forwarded. Instead, a portion of the address bits of outstanding loads and stores are compared and the loads are stalled until there is no longer a conflict. This strategy is correct, in the sense that older data is never loaded, but non-optimal in the case where there is a true match. This deficiency is especially noticeable when values are to be transferred between the integer and floating point register files. It also implies that data spilled to the stack due to lack of sufficient registers will incur an additional penalty on top of the additional loads and stores themselves. Fortunately the compiler's register allocation is good enough to make spills rare.

  For those occasions in which processing must be split between the floating point and integer units, a construct such as `isum += (int) fval` will most likely incur the aforementioned penalty, since the value `fval` will be converted into integer format, stored to the stack, and loaded into an integer register immediately. It may be advantageous to split the inner loop, writing out a line of intermediate results that have been converted to integer format for further processing by a second loop body:

```
int isum, *buf;
float fval;

/* loop */ {
  /* compute fval */
  buf[i] = (int) fval; /* convert and store */
}

/* loop */ {
  isum += buf[i];
}
```

  In this way, values read by the second loop will have had time to exit the store buffer, while still retaining a high likelihood of being found in cache.

- Microtraps for violations of VIS latency.

  Latencies of more than one cycle exist between some pairs of VIS instructions. Table 2 in section 4.11 describes these latencies in detail. For some latencies involving cross-precision operations, violations incur not only a stall but a "microtrap." In effect, the hardware interlocks are optimistic about cross-precision use of registers. When this optimism turns out to be unjustified, i.e., a read is attempted on a register whose value is not up-to-date, the processor state must be cleaned up. This takes 9 cycles.

The modulo scheduler (section 6.4.4) forces these latencies to be respected, inserting no-ops if necessary. Less sophisticated schedulers do not know how the instructions will group at run time, and so cannot determine how many no-ops will be necessary. This can lead to some rude surprises in which small changes in the schedule can result in large swings in performance.

- No prefetch instruction.

  A significant addition to the SPARC architecture for v9 is a prefetch instruction allowing data at an address to be accessed in the future to be brought into the CPU without a stall. UltraSPARC-I implements this as a no-op. Future UltraSPARC CPUs will implement at least some variants of this instruction, which has the potential to make imaging code highly insensitive to the limitations of the cache.

2.9 The SPARC Register Architecture

Several general features of the SPARC register architecture are worth noting at this point since they will bear on the discussions of implementation in the sequel. Full details are found in [SPARC94].

- Separate Integer and Floating Point Register Files.

  Integer and floating point values are stored in completely separate register files, accessed by separate instructions. Load and store instructions can access both files. There is no way to transfer values between the register files without passing through memory.

- Register Windows.

  The integer register file is organized into a number of overlapping sets. At any given time four ranges, each comprising eight registers, are available: the `%g`, `%i` `%l`, and `%o` registers. The `%g` registers are global. Register `%g0` is hard-wired to 0, as in many RISC processors. Registers `%g5`-`%g7` are used by the operating system and should not be altered by user code.

  Procedures take their arguments via the `%i` registers and pass arguments to any procedures they call via the `%o` registers. The `%l` registers are local to the procedure. When a procedure call occurs, the "window" is shifted so that the `%o` registers of the caller become the `%i` registers of the callee, which acquires its own `%l` and `%o` registers. Arguments that do not fit into the `%o` registers are passed on the stack. Upon return from the callee, the window is shifted back and the callee's `%l` and `%o` registers become inaccessible. A certain number of shifts are supported by the hardware; if this number is exceeded the operating system will spill the contents of the windows as needed. The frequency of such spills is dependent on the dynamic procedure call depth of the user's code. Languages such as Lisp often require very deep calls, but C programs have typically required a much lesser dynamic depth. This depth has decreased further as compilers have come to support function inlining and other interprocedural optimizations. The growing popularity of object-oriented techniques in C and C++ as well as the increased number of calls to dynamically linked library functions (which the compiler is powerless to optimize across) is now causing a reversal of this trend; graphical user interfaces and windowing code are particular culprits due to the heavy use of inheritance and the separation between device-independent and device-dependent layers. This illustrates the difficulty of establishing the value of any architectural feature given the ever-changing demands of software. UltraSPARC-I does offer substantially better window spill performance than its predecessors.

The floating point register file is not windowed. The compiler is responsible for implementing a caller-saves mechanism for values stored in the floating point registers. Thus only those registers that are live across a procedure call must be saved.

- Register Pairs.

  Some instructions refer to a pair of registers, always beginning with an even-numbered register. On the integer side this is relatively rare, but on the floating point side it is commonplace. The double-precision register `%f`$(2n)$ (also written as `%d`$(2n)$) shares space with the single-precision registers `%f`$(2n)$ and `%f`$(2n+1)$. Double-precision registers `%f32` through `%f62` have no single-precision counterparts. In effect the same five bits of register specification in a floating point instruction are decoded differently depending on the argument precision.

2.10 Estimating UltraSPARC-I Performance

In section 6.4.3 we will explain how to combine work from multiple loop iterations in order to avoid wasted cycles. Thus, in the context of a loop, the time taken to run a single iteration in isolation is not necessarily relevant to performance. No matter what sort of optimization is used, however, there is only a finite amount of computational bandwidth available. In particular, each instruction must be issued to a particular functional unit which is then unable to accept more instructions for the duration of the cycle. By counting the use of the various functional units, we can estimate the minimum cycle count of a loop iteration. By combining several such lower bounds and taking the maximum, we obtain a rough performance metric that is highly useful when making algorithmic decisions:

$$\text{cycles} \geq \max\left(\frac{\text{Instructions}}{4}, \frac{\text{IEU}}{2}, \text{Shifts}, \text{FGA}, \text{FGM}, \text{LSU}\right) + \texttt{alignaddr}s$$

IEU stands for the twin integer units, FGA and FGM stand for the floating point/graphics addition and multiplication units respectively, and LSU stands for the load/store unit. The `alignaddr` instruction (section 4.5) does not group with any other instructions, and so adds a cycle for each invocation (strictly speaking, they should be excluded from the total instruction count in order for the formula to be correct). Branching is ignored since it will never be the bottleneck in the sorts of loops we are interested in due to the ease of correct prediction. The compiler computes a quantity similar to this before attempting loop scheduling, since any attempt to schedule loop initiations closer together than this minimal number of cycles will obviously fail.

2.11 The Role of Benchmarking

As in any CPU design process, there is a question as to how various finite resources are to be allocated amongst competing functional units. These resources include such things as die area, power consumption, and critical path length. The main metric of feature importance at the present time is the 1992 version of the commercially available benchmark suite from the Standard Performance Evaluation Corporation (SPEC92) that measures the speed of a given CPU/compiler pair on a fixed set of 20 programs – 6 integer, 14 floating point – that are intended to be representative of a variety of common processing tasks. During the design of a CPU, a hardware feature affecting the timing of various operations is typically evaluated according to how it will affect the final SPEC weighted average, or other benchmarks of interest to the designers. While inexact, this process affords designers the opportunity to shift responsibilities for different aspects of the computation between the CPU and the compiler in any manner they wish. This is a major improvement on earlier benchmarks that explicitly required compiler optimizations to be disabled in an attempt to measure "pure" hardware performance, a quantity that has become ever more irrelevant as hardware design and compiler technology have found common ground.

Since there is neither a standard imaging benchmark suite, nor a compiler capable of emitting VIS instructions, this sort of quantification was not possible. The designers of the chip were therefore forced to rely on small hand-written loops and back-of-the-envelope calculations of their performance. The VIS instructions were also subject to far less testing than the regular v9 instructions, since no real-world VIS binaries existed (VIS instructions were however included in randomly-generated test codes). It was left to the imaging library programmers to test the instructions in the context of actual programs, and to make good on the promise of performance. Thus the decision to implement the VIS extensions was not without risk, having the potential to increase the time-to-market without a definite guarantee of performance.

3 Characterization of Imaging Tasks

Although most imaging operators appear simple to specify and code, naïve implementations typically suffer from extremely poor performance. The effort required to achieve optimal performance is substantial, as is the size of the optimal code, both in source and binary form. Maintaining such code can be difficult, as each incorrect assumption made during the design propagates throughout a labyrinth of special cases. Finally, some hardware resources are simply unused or utilized at a fraction of their theoretical capacity, and little can be done to remedy this.

For the purposes of this section, we will assume code is to be run on a processor similar to UltraSPARC-I, without the use of the VIS extensions. A few shortcomings of the UltraSPARC-I implementation will be ignored when they affect the theoretical performance of the non-VIS code; in particular, we will consider both integer execution units as possessing shifters, and we will not impose a penalty for using integer conditional move instructions.

In this section we aim to provide the reader with a brief summary of the data formats used in imaging as well as a look at some of the difficulties involved in writing generic imaging code with acceptable performance. The observation of these difficulties provides the motivation for several key operations of VIS for performing data alignment, parallel arithmetic, and packing and clamping of data.

3.1 Image Data Formats

For the purposes of this report, we will use a straightforward image format. Embellishments such as *tiling* (processing images a subrectangle at a time) or non-rectangular *regions of interest* (areas to be processed) may be assumed to be implemented on top of this format.

Pixel data will consist of one or more *channels* (or *bands*). Each channel represents some independent measurement associated with a given pixel's (x, y) position in whatever coordinate system was sampled to acquire the image. Some common special cases are: a single channel representing grayscale (i.e., the eye's scotopic response to a scene); a single channel representing indices into a color table; three channels representing the coordinates of a three-dimensional color space such as RGB, YC_bC_r, or Lab; or four channels representing the coordinates of a four-dimensional (overspecified) color space such as $CMYK$. However, there is no necessary restriction to optical data. For example, a satellite image might contain information concerning color, temperature, height, and wind velocity at each point. Our multi-channel images can represent any number of variables, provided they are all to be represented in the same underlying format. If some channels require floating point accuracy and others require only 8-bit fixed point, they should be stored in separate images. Discussions of color spaces may be found in most graphics texts; a comprehensive discussion may be found in [Foley90].

An image will be represented as a two-dimensional array of (possibly multi-channel) pixels (a *raster*). The coordinates increase as we move down and to the right (this is an arbitrary choice and some systems, e.g., OpenGL increase upwards in the manner of the Cartesian plane). Each pixel consists of either unsigned bytes or signed shorts, one per channel. There may be padding bytes between pixels; this is captured in the notion of *pixel stride*, which is an integer greater than or equal to the number of bands multiplied by the size of the image datatype in bytes. This quantity, when added to the byte address of a given channel within a particular pixel, yields the byte address of the same band within the pixel's horizontal successor.

Similarly, the separation in bytes between vertically adjacent pixels is recorded as the image's *scanline stride* (or "linebytes"). This stride is greater than or equal to the image width multiplied by the pixel stride, i.e., there may be unused bytes in between adjacent scanlines.

Our multi-channel format is sometimes referred to as *band interleaved* format; storing each band in its own raster is known as *band separated* format. Both formats are used extensively, often within the same application.

Given a pointer `base` to the first band of the upper left pixel of a byte image with pixel stride `ps` and scanline stride `ss`, the byte representing channel `c` of pixel (x, y) may be indexed as:

```
*((unsigned char *) base + y*ss + x*ps + c)
```

Figure 2 illustrates the mapping between a linear range of addresses and a rectangular raster. For the image shown, the number of channels and the pixel stride are both equal to 1. The image is 3 pixels wide but has a scanline stride of 4. The highlighted pixel lies at position $(1, 2)$, and so would be addressed as `base + 2*ss + 1`. Figure 3 illustrates the multi-channel case. The number of bands and the pixel stride equal 3. The width is 2 and the scanline stride is $7 > 6 = 2 \cdot 3$. Channel 1 of pixel $(1, 1)$ is highlighted, and is addressed as `base + ss + ps + 1`.

For short images, the computation is similar:

```
*((short *) ((unsigned char *) base + y*ss + x*ps + c*2))
```

Note that `base` is assumed to be short-aligned. The quantities `ss` and `ps` are represented in terms of bytes, so it is inappropriate to cast `base` to `(short *)` during the address computation. Alternatively, `ss` and `ps` may be divided by two and the computation done in terms of `short`s. Both `ss` and `ps` should be even for `short` images.

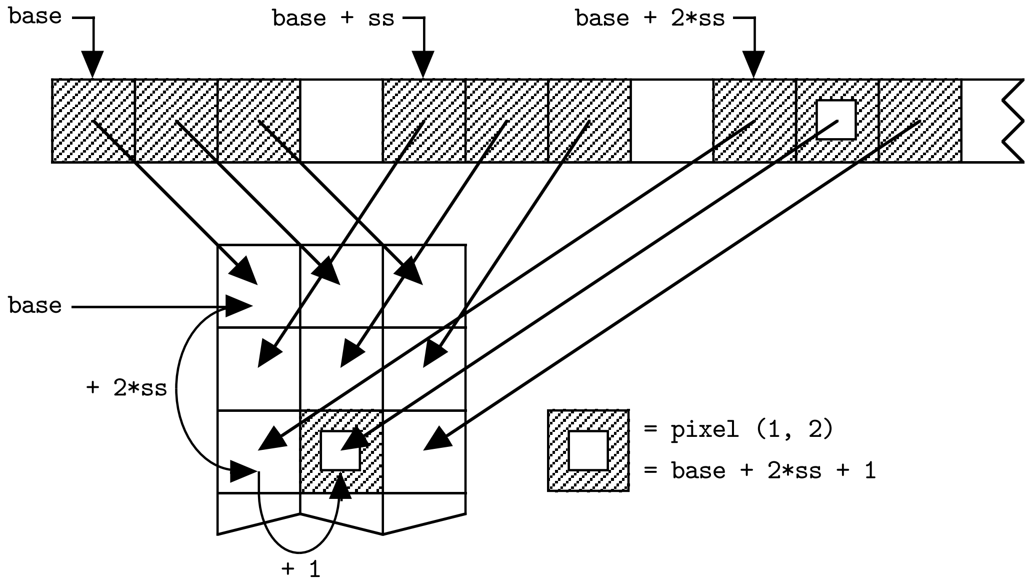

Figure 2: The mapping between an address range and a raster.

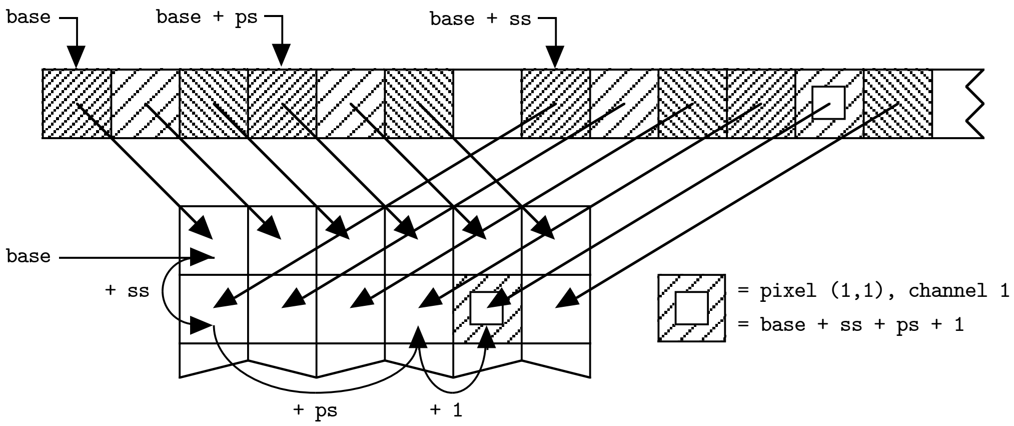

Figure 3: Multichannel image layout.

Naturally we do not wish to perform three multiplications for each pixel reference. The technique of *strength reduction*, either performed manually as part of algorithm development or automatically by an optimizing compiler, turns a series of multiplications of an incrementing argument by a constant into a series of constant additions. For example, the expression `x*ps` may be reevaluated by simply adding `ps` to its previous value each time `x` is incremented. Similarly, each time a new scanline is to be processed we add `ss` to the previous row pointer to obtain a pointer to the new row.

The "extra" bytes between pixels and scanlines may not be altered by a correct algorithm, although it is legal to read them provided they have no effect on the output. This allows sub-rectangles and/or subsets of the channels to be processed without affecting the remainder of the image. Consider writing to the green channel (`c` = 1) of a rectangle from $(10, 12)$ to $(19, 19)$ (inclusive) in an image with pixel stride `ps` and scanline stride `ss`.

The base pointer of the new image is computed as:

```
(unsigned char *) base + 12*ss + 10*ps + c
```

The "child" image inherits its parents `ps` and `ss` parameters, and has a width of $19-10+1 = 10$, a height of $19 - 12 + 1 = 8$, and a single band. It may be processed just like any other image without the need to consider the fact that it shares pixel data with a larger parent image.

3.2 Implementing an Algorithm in C

Consider a simple routine to add two images pixelwise, clamping the results between 0 and 255 (in this case only the latter need be considered), shown in Figure 4. Our goal is to determine how fast this routine may be made to go on an UltraSPARC-I class processor, using only generic C code; at the same time, we wish to observe the shortcomings of this approach in order to understand the opportunities for hardware assistance.

The performance of such a simple loop is dominated by the number of load and store operations. We will show that the obvious techniques for lowering this burden, namely consolidating output stores and reading multiple input values at once, are ineffective due to the large amount of data realignment they require. In effect, software is forced to assume the responsibility for a function normally performed by dedicated hardware within the load/store unit. VIS introduces several instructions that deal with the problem of alignment.

VIS offers instructions for parallel arithmetic. But as we shall see, such a capability is useless without some effective way of bringing appropriate sets of values into the processor in parallel.

We will also see that the need to perform clamping using conditional branches is a significant drain on performance. Accordingly VIS provides a facility for unconditional clamping.

Since three memory references are required per pixel of output, this loop cannot run faster than three clocks per pixel on a machine with a single load/store unit. The use of `C`'s "`?`" selection operator (in the definition of `min`) may require the generation of a conditional in the assembly output if the compiler does not generate conditional moves, which may also affect performance by preventing some classes of loop optimization. We will ignore this effect for now.

Since the memory references appear to dominate at the moment, consider reducing them by loading more than one byte at a time. Integer loads of four or eight bytes are available; however, they require that the source address be aligned according to the size

```
#define min(a,b) ((a) < (b) ? (a) : (b))
#define CLAMP255(a) min((a), 255)

void
add_images (unsigned char *im1, unsigned char *im2,
            unsigned char *dest,
            int width, int height,
            int im1_lb, int im2_lb, int dest_lb)
{
  int row, col, val;

  for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
      val = im1[col] + im2[col];
      dest[col] = CLAMP255(val);
    }

    im1 += im1_lb;
    im2 += im2_lb;
    dest += dest_lb;
  }
}
```

Figure 4: Naïve routine to add two images.

of the unit to be loaded. Multi-byte store instructions have the same restriction. In fact the store restriction is more significant, since extra data may be read harmlessly (within a valid segment of memory), but only the requested bytes may be written. Even a read-modify-write strategy is not valid, since the spurious output bytes might be mapped onto a frame buffer or other output device where incorrect values must not appear. Accordingly, we replace the inner loop with three sections of code, shown in Figures 5-7, in preparation for further optimization. This code deals with the destination alignment only, continuing to read the source data a byte at a time.

The main loop now contains 9 load and store operations for 4 pixels, potentially allowing a speedup of 1.33 if memory access remains the resource constraint. There are 4 additions, 3 shifts, and 3 logical "or"s that will require at least 5 cycles to be issued, but this constraint

```
save_width = width; save_im1 = im1; save_im2 = im2; save_dest = dest;
align = 4 - ((unsigned long) dest & 0x3); /* Number of unaligned bytes. */
if (align > width) align = width;
for (col = 0; col < align; ++col) { /* Work until dest is aligned. */
  val = im1[col] + im2[col];
  dest[col] = CLAMP255(val);
}

/* Update variables. */
width -= align; im1 += align; im2 += align; dest += align;
dptr_4 = (unsigned long *) dest;
```

Figure 5: Initial loop to align the destination.

```
times = width/4;
for (i = 0; i < times; ++i) {
  col = 4*i;
  val0 = CLAMP255(im1[col]     + im2[col]);
  val1 = CLAMP255(im1[col + 1] + im2[col + 1]);
  val2 = CLAMP255(im1[col + 2] + im2[col + 2]);
  val3 = CLAMP255(im1[col + 3] + im2[col + 3]);

  dptr_4[i] = (val0 << 24) | (val1 << 16) | (val2 <<  8) |  val3;
}
width -= 4*times; im1 += 4*times; im2 += 4*times; dest += 4*times;
```

Figure 6: Main loop to add and store four pixels at once.

```
for (col = 0; col < width; ++col) { /* Clean up extra pixels. */
  val = im1[col] + im2[col];
  dest[col] = CLAMP255(val);
}

/* Increment pointers to the next scanline. */
width = save_width;
im1 = save_im1 + im1_lb; im2 = save_im2 + im2_lb;
dest = save_dest + dest_lb;
```

Figure 7: Cleanup of final three or fewer pixels.

is still overshadowed by the memory constraint.

Figure 8 shows the breakup of the sources and destination into initial and final cleanup regions of fewer than four bytes, and four byte internal regions. The heavy lines show the division of memory into four byte words.

In order to read sources more than a single byte at a time, it becomes necessary to account for various alignments of the source relative to the destination. After the initial loop to align the destination pointer, 16 cases will be needed to account for the various possible alignments of `im1` and `im2`. Before entering the case structure, the source pointer offsets are recorded and 8 bytes are read from the aligned addresses immediately preceding the source addresses. This is shown in Figure 9.

The four pixels required from each source must lie somewhere within the eight bytes stored in `im1_data0`, `im1_data1`, `im2_data0`, and `im2_data1`. We do not have to worry about reading from an illegal address since the memory protection system controls memory on a page-by-page basis; if a given address is valid for reading (i.e., it does not cause a segmentation fault), all other addresses on the same page will also be legal. Aligning an address by masking out its lower bits does not change its page number as long as no more than lg(pagesize) bits are masked.

The quantity `4*im1_offset + im2_offset` may be used to control a `switch` statement with cases numbered from 0 to 15. Consider case number 9, for which `im1_offset` is 2 and `im2_offset` is 1. The desired data lie in bytes 2 and 3 of `im1_data0`, bytes 0 and 1 of `im1_data1`, bytes 1, 2 and 3 of `im2_data0` and byte 0 of `im2_data1`, counting from the left. The first four destination bytes can therefore be computed (ignoring clamping) as shown in Figure 10.

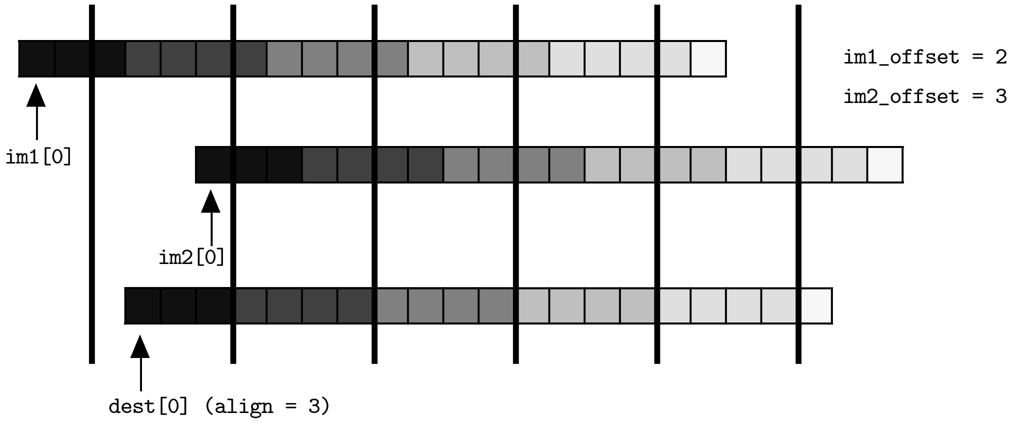

Figure 8: Aligning source and destination spans to memory.

```
#define OFFSET(ptr) ((unsigned long) ptr &  0x3)
#define ALIGN(ptr)  ((unsigned long) ptr & ~0x3)

im1_offset = OFFSET(im1); im2_offset = OFFSET(im2);
im1_aligned = (unsigned long *) ALIGN(im1);
im2_aligned = (unsigned long *) ALIGN(im2);

im1_data0 = *im1_aligned++; im1_data1 = *im1_aligned++;
im2_data0 = *im2_aligned++; im2_data1 = *im2_aligned++;
```

Figure 9: Initial alignment of source pixels.

Note that there is no need to mask with `0xff` after shifting an argument right by three bytes. Because results above 255 are to be clamped, the masks must be performed in order to remove junk bits prior to the addition. If wraparound were permitted, these bits could be left intact to participate in the (two's-complement) addition, and the sum masked, reducing the number of mask operations from 6 to 4.

After such a loop iteration, the value of `im1_data1` is transferred to `im1_data0`, and similarly for the second source image variables. New image data are read through the aligned pointers into `im1_data1` and `im2_data1`. The pattern of shifts remains invariant from iteration to iteration.

```
val0 = ((im1_data0 >>  8) & 0xff) + ((im2_data0 >> 16) & 0xff);
val1 = ((im1_data0)       & 0xff) + ((im2_data0 >>  8) & 0xff);
val2 = ((im1_data1 >> 24)       ) + ((im2_data0)       & 0xff);
val3 = ((im1_data1 >> 16) & 0xff) + ((im2_data1 >> 24)       );
/* Clamping would be performed here. */
dptr_4[i] = (val0 << 24) | (val1 << 16) | (val2 <<  8) |  val3;
```

Figure 10: Extraction of source pixels.

```
im1_data = (im1_data0 << 16) | (im1_data1 >> 16);
im2_data = (im2_data0 <<  8) | (im2_data1 >> 24);
```

Figure 11: One-time extraction of source pixels.

```
val0 = (im1_data & 0xff000000) + (im2_data & 0xff000000);
val1 = (im1_data &   0xff0000) + (im2_data &   0xff0000);
val2 = (im1_data &     0xff00) + (im2_data &     0xff00);
val3 = (im1_data &       0xff) + (im2_data &       0xff);
dptr_4[i] = val0 | val1 | val2 | val3;
```

Figure 12: Use of pre-extracted pixel data.

This approach requires 9 shifts, 9 logical operations, and 4 additions for each four bytes. Assuming, as noted above, that these are all simply IEU operations, $22/2 = 11$ cycles will be required for issue, or 2.75 clocks/pixel, a net slowdown over the read one/write four approach.

We can simplify things further by forcing `val0`, etc., to emerge in the proper places to be joined and stored without additional shifting. For example, `val1` and `val2` may be computed as:

```
val1 = ((im1_data0 << 16) & 0xff0000) + ((im2_data0 <<  8) & 0xff0000);
val2 = ((im1_data1 << 16) &   0xff00) + ((im2_data0 <<  8) &   0xff00);
```

The sum `val1` (resp. `val2`) now appears in byte 1 (resp. 2), followed by 16 (resp. 8) zero bits. The clamping must now be performed against `255 << 16`, or 16711680 (resp. `255 << 8`, or 65280). Note that where computing `val2` previously required only one shift, it now requires two, eliminating any savings.

Clamping `val0` is trickier, since the intermediate result could overflow 32 bits. `val3` is already in the desired place, so no change in its computation is needed. As a practical matter, then, only one shift is saved for this case. Other alignment cases may differ slightly, with some offering more fortuitous positioning than others.

What we are seeing here is the power of the memory system's alignment network. When a byte is read from memory, this network places it into the proper position of the destination register. In exchange for reading several bytes at once, the software must perform this function itself. For each two pixels to be added, two shifts and two logical ands are generally required for this realignment, as opposed to a single shift for the byte-at-a-time approach. Given our machine model, we are in rough terms trading one 2 cycle constraint (two loads) against another (four IEU operations).

Consider the effect of a faster alignment technique. The idea is to take data from `im1_data0` and `im1_data1` and realign it so that byte 2 of `im1_data0` occupies position 0. This can be performed as shown in Figure 11. Rewriting the computations of `val0` through `val3` using these new values, we obtain the code shown in Figure 12.

Here we are ignoring the `val0` clamping problem, and in fact all clamping. We still require 22 IEU operations, however. Suppose now that an alignment operator were available in hardware. The computation of `im1_data` and `im2_data` above would be rewritten as:

```
im1_data = align(im1_data0, im1_data1, 2);
im2_data = align(im2_data0, im2_data1, 1);
```

```
im1_data = align(im1_data0, im1_data1, 2);
im1_data0 = im1_data1; im1_data1 = *im1_aligned++;

im2_data = align(im2_data0, im2_data1, 1);
im2_data0 = im2_data1; im2_data1 = *im2_aligned++;

dptr_4[i] = clamped_add(im1_data, im2_data);
```

Figure 13: Use of a hypothetical clamped addition operator.

Assuming each alignment is a single IEU operation, we are down to 18 operations, or a 9 cycle loop minimum. That is, we have finally broken even. A more significant benefit is the ability to collapse the loop from 16 cases back to only one by using `im1_offset` and `im2_offset` as arguments to `align` in place of hard-coded constants.

A further benefit is the regularization of the loop body. Imagine collapsing the four lines in which the additions take place into simply `val = im1_data + im2_data`. This almost computes the correct answer, with two caveats: first, overflow bits will be carried from one sum to the next; and second, there will be no way to recover the information required for clamping. If a version of addition were available that dealt with these problems, the entire loop could be handled simply by the code in Figure 13.

This loop requires 3 load/store instructions, two alignments, and a single clamped addition for every four pixels. The cost of copying the old data values may be eliminated by unrolling the loop and renaming the variables in each alternate iteration. The theoretical performance of this loop is thus most likely around 3 clocks/4 pixels, or .75 clocks/pixel. This is a factor of 3 over the previous best case. It also occupies significantly less space than our intermediate attempts. In addition, it is possible to consider reading data in larger quantities, such as 8 bytes at a time, without any corresponding code size explosion.

3.3 Conclusions

By introducing the alignment and clamped addition operators, we have constructed a loop that is more similar in spirit to the SPECfp loops than the SPECint ones – data are read in large blocks which are processed identically, with only simple branching. The artifact of source alignment no longer dominates performance, and hardware resources such as an addition circuit are utilized fully. The techniques used to ensure that floating point loops run fast should also apply to such a loop structure. In the next section, we examine the VIS instructions, which provide the needed fusion between integer and fixed point imaging computation and floating point performance.

4 The Visual Instruction Set

This section outlines the various VIS operations and attempts to define their semantics more precisely than do other existing documents. It does not attempt to be complete in all respects; those interested in using VIS should consult the VIS Users' Guide [SME95b] for detailed information. Some memory operations, as well as the `edge` instructions, have versions for use on machines that access memory addresses in so-called "little-endian" (byte-reversed) order, which we will not discuss.

```
typedef float          vis_f32;
typedef double         vis_d64;

typedef unsigned char  vis_u8;
typedef char           vis_s8;
typedef unsigned short vis_u16;
typedef short          vis_s16;
typedef unsigned long  vis_u32;
typedef long           vis_s32;

typedef void          *vis_ras;
```

Figure 14: VIS type definitions.

In the following discussion, single-precision (32-bit) register arguments are prefixed by "f" (for "float") in the instruction definitions; double-precision (64-bit) arguments are prefixed by "d." Integer registers are prefixed by "i" (64-bit integer registers, used by the `array` instruction are denoted with an "x").

The VIS 32- and 64-bit datatypes may be manipulated from C as `float` and `double` values (section 6.1.1 discusses some caveats). This relies on the fact that the floating point load and store instructions do not force their data to adhere to the IEEE 754 floating point format, but rather transfer data bit for bit. Thus it is possible to move VIS data between the floating point registers and main memory.

In order to differentiate VIS values from ordinary floating point values, we will always declare such values using the C `typedef`s shown in Figure 14. The other `typedef`s aid in distinguishing the sizes and signed/unsigned status of the various integral types. The `vis_ras` type ensures that we cast image pointers to pointers of the appropriate size before use when writing generic routines, since a `(void *)` cannot be dereferenced as-is.

4.1 Data Formats and Conversions

VIS operates on register data in five distinct formats, shown in Figure 15. Images are commonly represented either by one 8-bit unsigned byte or one 16-bit signed short per channel. VIS arithmetic generally requires some expansion before 8-bit data can be used, although the `pdist` instruction (see section 4.9) can operate directly on such data. All signed values are represented in two's-complement format.

4.1.1 The Graphics Status Register

`rd`	`%gsr`,	*idst*	
`wr`	*ireg*,	*reg_or_imm*,	`%gsr`

The graphics status register (`%gsr`) is a 64-bit register containing arguments used implicitly by several VIS operators. Currently two fields are defined, a 3-bit field (in bits $2 \ldots 0$) holding an offset that is set by `alignaddr` and used by `faligndata` (section 4.5), and a 4-bit shift value (in bits $6 \ldots 3$) that must be set via an explicit `wr` instruction and which is used by the various `fpack` instructions (section 4.1.3). Explicit writes to the `%gsr` stall the processor for 6 cycles, so changing the shift field during processing is best avoided.

63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0

Eight unsigned 8-bit values (raw format)

63 48	47 32	31 16	15 0

Four signed 16-bit values

31 16	15 0

Two signed 16-bit values

63 32	31 0

Two signed 32-bit values

31 0

One signed 32-bit value

Figure 15: VIS data formats.

In the sequel we access the `wr` instruction using an inline template `vis_write_gsr` (see section 6.1). Reading the `%gsr` is uncommon; in the existing VIS code each routine requiring a particular setting overwrites the previous value, and resets its value after any subroutine calls to other VIS routines. In other words, we have chosen a "caller saves" semantics in which routines are not required to set the `%gsr` to its previous value on exit. A "callee saves" semantics would also be possible. Since the `%gsr` is not part of the SPARC v9 standard, the SPARC ABI (application binary interface) does not impose a particular choice of convention, as it does for the other portions of the processor state. Solaris 2.5, the current version of Sun's operating system, saves and restores the `%gsr` on a context switch but does not initialize it to a particular value during process startup.

4.1.2 Expansion

`fexpand`	*fsrc*,	*ddst*

Raw pixel data may be converted to signed 16-bit format four pixels at a time via the `fexpand` instruction, or by an appropriate multiplication operation (see section 4.2.2). Each byte of the source register *fsrc* is effectively shifted left four places and stored to the destination *ddst* in 16-bit signed format. Expansion is necessary if values are to be added, subtracted, or compared. Figure 16 illustrates the `fexpand` instruction.

The `fexpand` instruction is processed by the graphics adder.

4.1.3 Packing

`fpack16`	*dsrc*,	*fdst*	
`fpackfix`	*dsrc*,	*fdst*	
`fpack32`	*dsrc1*,	*dsrc2*,	*ddst*

The inverse of expansion is packing. The `fpack` instructions convert data from a wide format to a narrower one, clamping the result at both ends of the legal output range. The `fpack16` instruction takes partitioned 16-bit signed data in *dsrc* and implicitly positions a

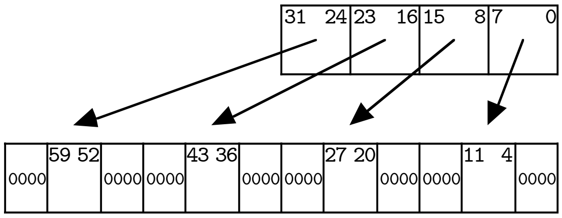

Figure 16: The `fexpand` instruction

binary point before the ($7 -$ `%gsr`) least significant bits of each of the four partitions. Thus it is possible for data to have between 0 and 7 bits of fractional precision prior to packing. The result is clamped to 0 if it is negative, and 255 if it overflows. Figure 17 illustrates the operation of the `fpack16` instruction.

When data values are multiplied, the number of integral bits of the product is equal to the sum of the integral bits of each argument. The number of fractional bits is simply whatever number remains. When multiplying raw pixel data (8 integral bits, 0 fractional bits) by a 16-bit constant, the output has the same number of fractional bits as the constant (since the multiply operation may be thought of as implicitly shifting the data right by 8 places, i.e., discarding 8 fractional bits). For example, if the constant were to be created by an `fexpand` operation on pixel data, it would have 12 integer and 4 fractional bits, and so would its product with raw pixel data. Accordingly a `%gsr` shift of 3 ($= 7 - 4$) would be used when the results are to be packed.

This example illustrates that the binary point is a somewhat abstract notion; operations such as addition and multiplication do not care about its location. As long as the programmer can establish that adequate precision exists throughout the computation of interest, the only instructions that require its position to be determined are the `fpack` instructions.

The `fpackfix` instruction packs and clamps signed 32-bit data in *dsrc* into signed 16-bit format. The effective number of bits of fractional precision is ($16 -$ `%gsr`). Data values are clamped between -32768 and 32767. Figure 18 illustrates the operation of the `fpackfix` instruction.

The `fpack32` instruction packs and clamps signed 32-bit data in *dsrc1* into unsigned 8-bit format. The effective number of bits of fractional precision is ($23 -$ `%gsr`). This instruction takes a second argument *dsrc2* consisting of 8-bit (packed) data, which is shifted left by 8 bits. The results of the `fpack32` instruction are inserted into bit positions 0-7 and 32-39 of the result. The intended use of this feature is packing of multiple 32-bit values into a stream of 8-bit values, as follows:

```
fpack32 vals_ae, accum, accum  ! accum = ...a...e
fpack32 vals_bf, accum, accum  ! accum = ..ab..ef
fpack32 vals_cg, accum, accum  ! accum = .abc.efg
fpack32 vals_dh, accum, accum  ! accum = abcdefgh
```

Figure 19 illustrates the operation of the `fpack32` instruction.

The packing instructions are processed by the graphics multiplier.

4.2 Arithmetic Operators

The arithmetic operators offer 1-, 2-, and 4-way fixed point arithmetic.

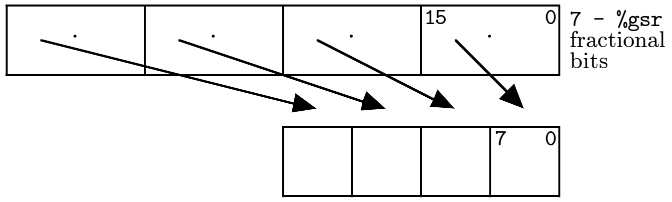

Figure 17: The `fpack16` instruction.

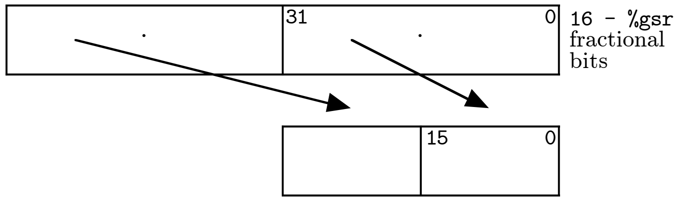

Figure 18: The `fpackfix` instruction.

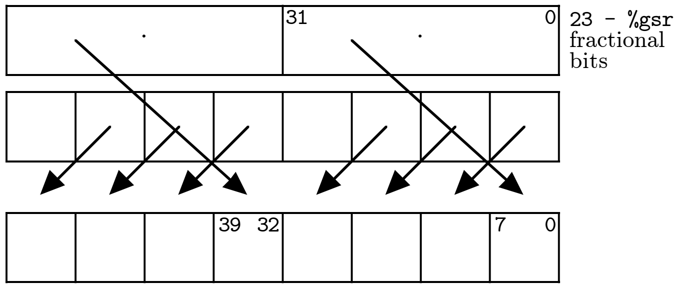

Figure 19: The `fpack32` instruction.

4.2.1 Addition and Subtraction

`fpadd16`	*dsrc1*,	*dsrc2*,	*ddst*
`fpadd32`	*dsrc1*,	*dsrc2*,	*ddst*
`fpsub16`	*dsrc1*,	*dsrc2*,	*ddst*
`fpsub32`	*dsrc1*,	*dsrc2*,	*ddst*

`fpadd16s`	*fsrc1*,	*fsrc2*,	*fdst*
`fpadd32s`	*fsrc1*,	*fsrc2*,	*fdst*
`fpsub16s`	*fsrc1*,	*fsrc2*,	*fdst*
`fpsub32s`	*fsrc1*,	*fsrc2*,	*fdst*

The `fpadd16` and `fpsub16` instructions perform four-way addition and subtraction of the partitioned values *dsrc1* and *dsrc2*. They may be thought of as two's complement addition and subtraction with carries (or borrows) between the partitions suppressed. There is no detection of overflow or underflow; instead, the programmer is responsible for maintaining all intermediate values within the proper range.

`fpadd32` and `fpsub32` perform the same operations on partitioned 32-bit data. All four instructions have short variants obtained by adding an "s" to their name that take 32-bit arguments.

These instructions are processed by the graphics adder.

4.2.2 Multiplication

`fmul8x16`	*fsrc1*,	*dsrc2*,	*ddst*
`fmul8x16au`	*fsrc1*,	*fsrc2*,	*ddst*
`fmul8x16al`	*fsrc1*,	*fsrc2*,	*ddst*
`fmul8sux16`	*dsrc1*,	*dsrc2*,	*ddst*
`fmul8ulx16`	*dsrc1*,	*dsrc2*,	*ddst*
`fmuld8sux16`	*fsrc1*,	*fsrc2*,	*ddst*
`fmuld8ulx16`	*fsrc1*,	*fsrc2*,	*ddst*

All of the multiply instructions perform a multiplication of a set of 8-bit quantities by 16-bit quantities to produce rounded 16-bit results. Rounding is towards positive infinity. In effect the multiplier computes $\lfloor (x \cdot y + 128)/256 \rfloor$.

The first three variants take four 8-bit pixels, in *fsrc1*, and multiply them by either four distinct 16-bit signed coefficients (`fmul8x16`) from *dsrc2*, or a single coefficient taken from either the top half (`fmul8x16au`) or the bottom half (`fmul8x16al`) of a 32-bit word, *fsrc2*. The latter two are useful when fixed coefficients are to be used, as in convolution and resampling. The ability to use both halves of a word can reduce the size of filter tables by half. Figures 20-24 illustrate the process of multiplication, rounding, and scaling for these instructions.

The next two variants, `fmul8sux16` and `fmul8ulx16`, are generally used in conjunction with the `fpadd16` instruction to produce an approximation to a 16×16-bit multiplication yielding a 16-bit result. Both take two arguments, *dsrc1* and *dsrc2*, each consisting of partitioned signed 16-bit data. The `fmul8sux16` instruction multiplies *dsrc2* by the top 8 bits of *dsrc1* to produce a 24-bit intermediate result which is rounded and truncated to 16 bits. The `fmul8ulx16` instruction does the same, only using the lower half of each partition of *dsrc1* and implicitly shifting its result right by an additional 8 bits to match the significance of the first product. Note that the sign of the first argument is significant in this process, even when it is the low bits that are being multiplied. An early hardware bug was caused by the failure of designers to take this into account – both the logic designer and the writer of the instruction-level simulator made the same mistake, preventing the bug from being caught until after the initial run of silicon.

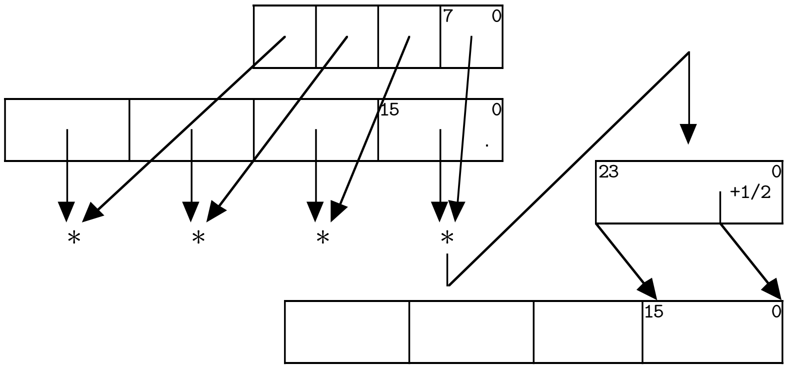

Figure 20: The `fmul8x16` instruction.

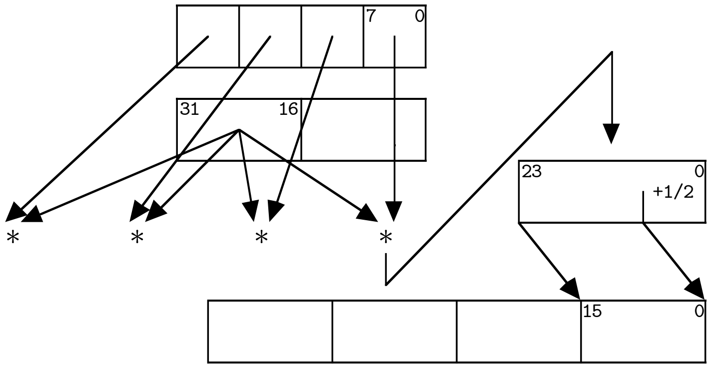

Figure 21: The `fmul8x16au` instruction.

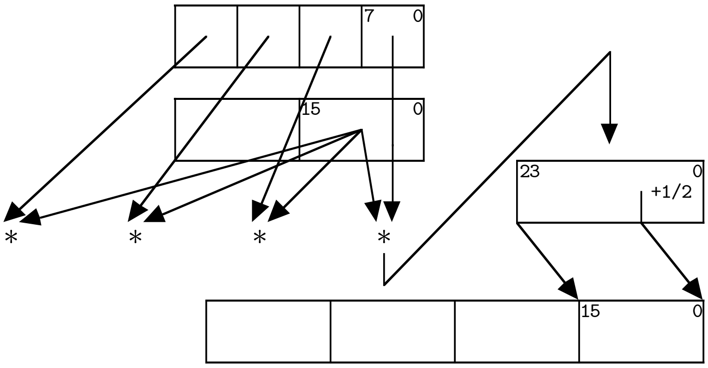

Figure 22: The `fmul8x16al` instruction.

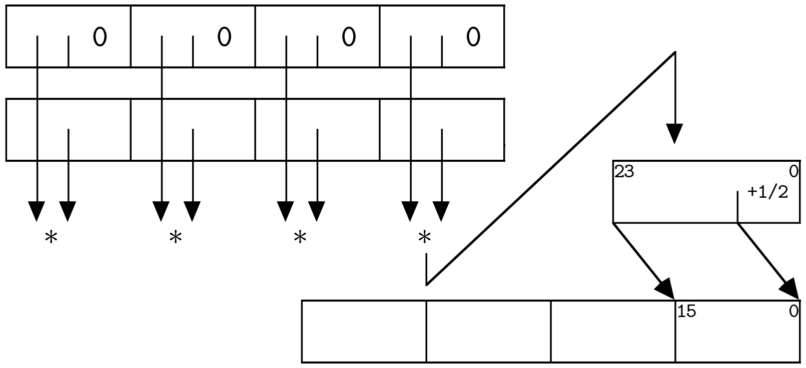

Figure 23: The `fmul8sux16` instruction.

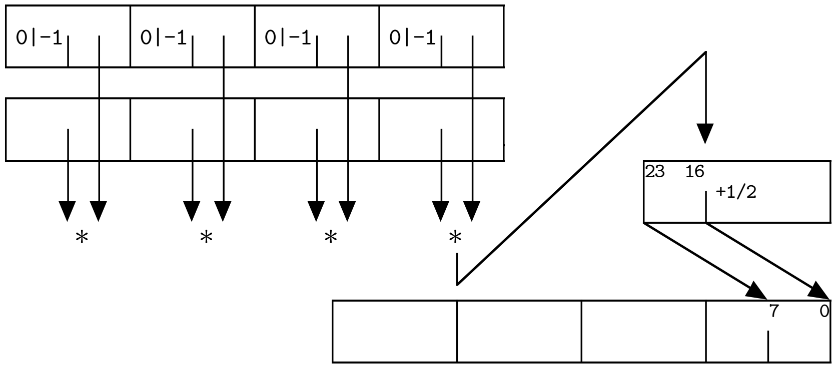

Figure 24: The `fmul8ulx16` instruction.

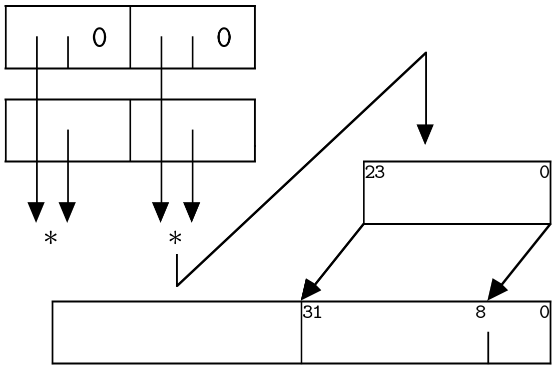

Figure 25: The `fmuld8sux16` instruction.

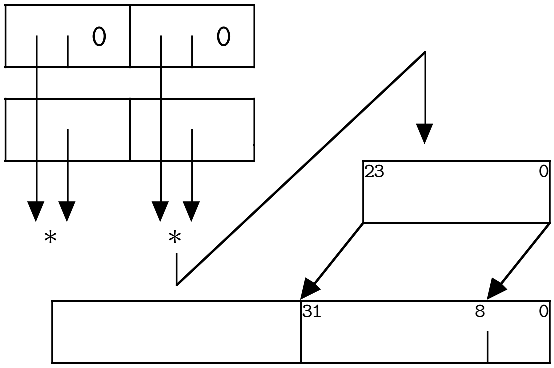

Figure 26: The `fmuld8ulx16` instruction.

Consider the case where both arguments are positive. Then the sum of the multiplications is:

$$\left\lfloor \frac{256\lfloor x/256 \rfloor \cdot y + 32768}{65536} \right\rfloor + \left\lfloor \frac{(x \bmod 256) \cdot y + 32768}{65536} \right\rfloor$$

The outer $\lfloor\ \rfloor$ operations represent the fact that only 16 bits of the intermediate values are preserved. On average half a bit of precision is lost in each half of the multiplication, so overall we lose about one bit. An exhaustive test of all 65536^2 possible argument pairs shows that the correct result (by comparison with `rint((double) x*(double) y/65536.0)`) is achieved 75.19% of the time. The result, as expected, is never more than 1 away from the true rounded product; it is too large $549,658,624$ times (12.80%) and too small $515,973,120$ times (12.01%).

The final variants (`fmuld8sux16` and `fmuld8ulx16`) each implement a true 8×16-bit multiplication, placing their result within a 32-bit partition. The result of `fmuld8sux16` is shifted left by 8 places while the result of `fmuld8ulx16` is sign-extended. Taking the sum of these two results (using `fpadd32`) yields the desired product. Figures 25 and 26 illustrate these instructions.

All of the multiply instructions are processed by the graphics multiplier.

4.3 Logical Operators

`fzero`	*ddst*		
`fone`	*ddst*		
`fsrc`	*dsrc1*,	*ddst*	
`fnot`	*dsrc1*,	*ddst*	
`for`	*dsrc1*,	*dsrc2*,	*ddst*
`fnor`	*dsrc1*,	*dsrc2*,	*ddst*
`fand`	*dsrc1*,	*dsrc2*,	*ddst*
`fnand`	*dsrc1*,	*dsrc2*,	*ddst*
`fxor`	*dsrc1*,	*dsrc2*,	*ddst*
`fxnor`	*dsrc1*,	*dsrc2*,	*ddst*
`fornot1`	*dsrc1*,	*dsrc2*,	*ddst*
`fornot2`	*dsrc1*,	*dsrc2*,	*ddst*
`fandnot1`	*dsrc1*,	*dsrc2*,	*ddst*
`fandnot2`	*dsrc1*,	*dsrc2*,	*ddst*

`fzeros`	*fdst*		
`fones`	*fdst*		
`fsrcs`	*fsrc1*,	*fdst*	
`fnots`	*fsrc1*,	*fdst*	
`fors`	*fsrc1*,	*fsrc2*,	*fdst*
`fnors`	*fsrc1*,	*fsrc2*,	*fdst*
`fands`	*fsrc1*,	*fsrc2*,	*fdst*
`fnands`	*fsrc1*,	*fsrc2*,	*fdst*
`fxors`	*fsrc1*,	*fsrc2*,	*fdst*
`fxnors`	*fsrc1*,	*fsrc2*,	*fdst*
`fornot1s`	*fsrc1*,	*fsrc2*,	*fdst*
`fornot2s`	*fsrc1*,	*fsrc2*,	*fdst*
`fandnot1s`	*fsrc1*,	*fsrc2*,	*fdst*
`fandnot2s`	*fsrc1*,	*fsrc2*,	*fdst*

The VIS logical operators perform all 16 possible logical operations on zero, one, or two arguments. In the case of `fzero` and `fone`, both arguments are ignored. These instructions are useful for generating a pattern of all zeros (distinct, at least in principle, from the floating point constants `(float) 0.0` and `(double) 0.0`) or all ones in a floating point register.

In the case of `fsrc` and `fnot`, only one argument is used. The hardware actually implements two versions of these operators (i.e., copy/negate first argument and copy/negate second argument). The assembler need only provide a mnemonic for one of these variants, of course, as shown above. The remaining two-argument operations also have some anti-symmetric cases which are shown (the operations with "1" and "2" in their names). The redundancies are a result of the implementation of these instructions, which makes direct use a 4-bit field within the instruction word to specify the operation's truth table.

The VIS logical instructions are processed by the graphics adder.

4.4 Merging

`fpmerge`	*fsrc1*,	*fsrc2*,	*ddst*

The `fpmerge` operator takes two single-precision arguments, *fsrc1* and *fsrc2*, and interleaves their bytes, as shown in Figure 27. The intended use of `fpmerge` is conversion from band interleaved to band separated format and vice-versa. Consider pixel data in (r, g, b, α) format, where α might represent the transparency of the pixel. The data may be copied into a separate span for each channel as follows:

```
fpmerge rgba0,rgba2,tmp1          ! tmp1 = r0r2g0g2b0b2a0a2
fpmerge rgba1,rgba3,tmp2          ! tmp2 = r1r3g1g3b1b3a1a3
fpmerge hi(tmp1),hi(tmp2),rg      !   rg = r0r1r2r3g0g1g2g3
fpmerge lo(tmp1),lo(tmp2),ba      !   ba = b0b1b2b3a0a1a2a3
```

The separated channels are now available in `hi(rg)`, `lo(rg)`, `hi(ba)`, and `lo(ba)`, where `hi(x)` refers to the even portion of a register pair `x` and `lo(x)` refers to its odd portion. If data are being converted in bulk, this code may be duplicated so as to write eight bytes of output at a time for each channel.

Reversing the process is equally simple:

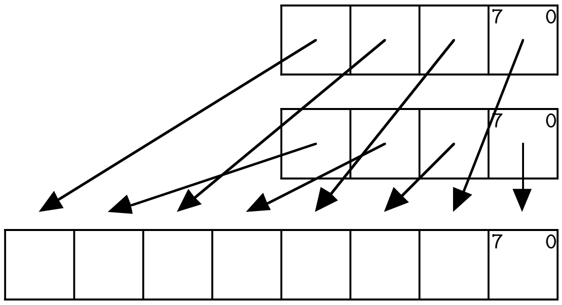

Figure 27: The `fpmerge` instruction

```
fpmerge r0,b0,tmp1               !   tmp1 = r0b0r1b1r2b2r3b3
fpmerge g0,a0,tmp2               !   tmp2 = g0a0g1a1g2a2g3a3
fpmerge hi(tmp1),hi(tmp2),rgba01 ! rgba01 = r0g0b0a0r1b1g1a1
fpmerge lo(tmp1),lo(tmp2),rgba23 ! rgba23 = r2g2b2a2r3g3b3a3
```

Note that the pixel format may be reordered by concatenating these operations with an appropriate reordering of the inputs to the second half. As we will see, `fpmerge` has other uses such as data transposition.

The `fpmerge` instruction is processed by the graphics adder.

4.5 Alignment Operators

`alignaddr`	*isrc1*,	*isrc2*,	*idst*
`faligndata`	*dsrc1*,	*dsrc2*,	*ddst*

The `alignaddr` instruction sums its integral *isrc1* and *isrc2* arguments and sets the alignment field of the `%gsr` to the lower three bits of the result. Typically, the first argument will be a pointer and the second will be a byte offset. The sum, with its lower three bits masked to zero, is returned as *idst*. This value may be subsequently used as an aligned pointer.

The `faligndata` instruction takes two 8-byte arguments *dsrc1* and *dsrc2* and conceptually concatenates them into a single sequence of 16 bytes:

$$p_0 p_1 p_2 \cdots p_{15}$$

For a `%gsr` alignment of s, $0 \leq s \leq 7$, bytes:

$$p_s p_{s+1} p_{s+2} \cdots p_{s+7}$$

are selected and stored into the destination, *ddst*. The combination of `alignaddr` and `faligndata` allows an unaligned span of 8 bytes to be read from an address `(ptr + offset)` using the idiom:

```
alignaddr   ptr, offset, aligned_ptr
ldd         [aligned_ptr + 0x0], data_hi
ldd         [aligned_ptr + 0x8], data_lo
faligndata  data_hi, data_lo, data
```

Instruction	3 LSBs	Left Edge	Right Edge
edge8	000	11111111	10000000
edge8	001	01111111	11000000
edge8	010	00111111	11100000
edge8	011	00011111	11110000
edge8	100	00001111	11111000
edge8	101	00000111	11111100
edge8	110	00000011	11111110
edge8	111	00000001	11111111
edge16	00x	1111	1000
edge16	01x	0111	1100
edge16	10x	0011	1110
edge16	11x	0001	1111
edge32	0xx	11	10
edge32	1xx	01	11

Table 1: Generation of left and right edge masks from pointers.

Software is responsible for ensuring that both loads are from valid addresses. The semantics of aligning pointers was discussed in section 3.2 above; the proper way to align data in a loop context is discussed below in section 7.1.

The `faligndata` instruction is processed by the graphics adder. The `alignaddr` instruction is processed by the integer unit and currently does not group with other instructions. The altered `%gsr` is available for use in the following cycle.

4.6 Edge Masking and Comparison

`edge8`	*isrc1*,	*isrc2*,	*idst*
`edge16`	*isrc1*,	*isrc2*,	*idst*
`edge32`	*isrc1*,	*isrc2*,	*idst*
`fcmpgt16`	*dsrc1*,	*dsrc2*,	*idst*
`fcmpgt32`	*dsrc1*,	*dsrc2*,	*idst*
`fcmple16`	*dsrc1*,	*dsrc2*,	*idst*
`fcmple32`	*dsrc1*,	*dsrc2*,	*idst*
`fcmpne16`	*dsrc1*,	*dsrc2*,	*idst*
`fcmpne32`	*dsrc1*,	*dsrc2*,	*idst*
`fcmpeq16`	*dsrc1*,	*dsrc2*,	*idst*
`fcmpeq32`	*dsrc1*,	*dsrc2*,	*idst*

4.6.1 Edge Masking

The `edge` instructions generate 8-, 4-, or 2-bit masks that may be used as arguments to a partial store instruction based on two pointers *isrc1* and *isrc2* such that $isrc1 \leq isrc2$. The mask is created so that a write to the (8-byte) doubleword containing the address pointed to by *isrc1* will not affect any bytes to the left of *isrc1* or to the right of *isrc2*. There are two cases: either *isrc1* and *isrc2* fall in different doublewords, or else they fall in the same

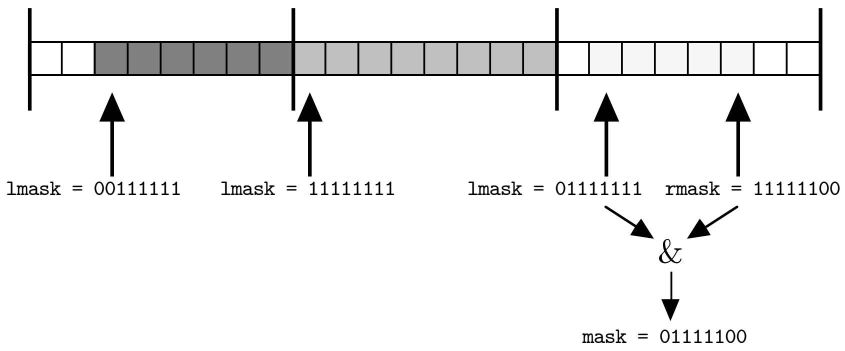

Figure 28: Three examples of edge masking.

doubleword. In the former case, only the bytes within the doubleword that are to the left of *isrc1* require masking, so only a left edge mask is used. In the latter case, the two pointers yield both left and right edge masks, which are logically "and"ed together. Table 1 shows the set of left and right edge masks generated by the various edge instructions, based on the 3 least significant bits of the two pointers.

Figure 28 illustrates the edge masking process. The first three arrows (reading from left to right) represent the *isrc1* arguments to three `edge8` instructions; the final arrow represents their common *isrc2* argument. For the first two calls, the *isrc2* argument is not relevant since the arguments fall in different words. The shaded regions denote the bytes covered by the resulting masks. The third call shows the "and"ing of the left and right edge masks to produce a mask with zeros on both ends.

When *isrc2* exceeds *isrc1*, the results of the edge instructions are undefined, and cannot be counted on to be zero. This means that loops that depend on the edge instruction to prevent writes beyond the end of a scanline must still take responsibility for terminating after writing to the doubleword containing the last image byte. In particular, hand-unrolled loops must contain exit tests after each write, or else must be invoked for fewer operations and followed by a separate cleanup loop.

The edge masking instructions are processed by the integer unit.

4.6.2 Partitioned Comparison

The comparison operators compare partitioned 16- or 32-bit data in *dsrc1* and *dsrc2* and produce a 4- or 2-bit mask in an integer register, *idst*. This mask may be logically "and"ed with the mask produced by an appropriate `edge` instruction, or a user-generated mask, or used in any other way desired. The "missing" comparisons ($\geq$ and $<$) may be synthesized as pseduo-ops: `fcmpge`$(a, b) \equiv$ `fcmple`(b, a) and `fcmplt`$(a, b) \equiv$ `fcmpgt`(b, a).

VIS does not possess a selection operator; instead, the result of a comparison may be realized by writing results that assume the comparison succeeded to memory using the mask generated using the partial store instruction described in section 4.7. Then results that assume the comparison failed are written to the same address using an inverted mask. If these stores are performed to cacheable memory (or to a device which supports this store mode) without any other stores intervening, the stores will be combined in the store buffer, and will not result in separate bus transactions.

The comparison instructions are processed by the graphics multiplier. The results of the comparison should not be used until the third following cycle; otherwise a microtrap (section 2.8) may be generated.

4.7 Partial Stores

`stda` *dsrc*, [*iaddr*]*imask*, *imm_asi*

The partial store operation stores 8 bytes from *dsrc* to a properly aligned address *iaddr*, just as a regular double-precision floating point store does; however, the partial store uses an integral mask, *imask*, to control the replacement of the output bytes. For each bit of *imask* that is zero, the data in memory (or on an appropriate memory-mapped device) at the corresponding postion is left unchanged.

The *imm_asi* field determines the ASI (address space identifier) of the store. ASIs allow selection among a number of hardware load and store variants, including big- and little-endian byte ordering and access to primary and secondary address spaces. VIS provides ASIs that perform partial stores of eight 1-byte, four 2-byte, or two 4-byte quantities. The mask bits, found in *imask*, are right-justified in each case.

4.8 Short and Block Loads and Stores

`ldda`	[*ireg*+*ireg*]*imm_asi*,	*ddst*
`stda`	*dsrc*,	[*ireg*+*ireg*]*imm_asi*

The short load and store instructions allow quantities smaller than four bytes, namely one and two bytes, to be transferred between memory and the floating point register file by means of a non-standard ASI, which is otherwise not possible in SPARC assembly language. The data need only be aligned on its natural boundary, i.e, even addresses for 2-byte quantities and arbitrary alignment for 1-byte quantities. The data will be right-justified within a double-precision floating point register in all cases. Quantities of four or eight bytes may be loaded and stored using ordinary `float` and `double` LSU instructions. These instructions may be accessed from C via inlines (section 6.1):

```
vis_d64 vis_ld_u8(vis_ras addr);
vis_d64 vis_ld_s16(vis_ras addr);
vis_d64 vis_ld_u8_i(vis_ras addr, int index);
vis_d64 vis_ld_s16_i(vis_ras addr, int index);

void vis_st_u8(vis_d64 value, vis_ras addr);
void vis_st_s16(vis_d64 value, vis_ras addr);
void vis_st_u8_i(vis_d64 value, vis_ras addr, int index);
void vis_st_s16_i(vis_d64 value, vis_ras addr, int index);
```

The indexed variants (having names ending with `_i`) add the index and address arguments together to produce the load address. This is free since SPARC load and store instructions calculate their effective addresses as the sum of two arbitrary integer registers. In the interest of performance the index is not scaled when accessing 16-bit data, nor is the resulting pointer checked for alignment. The enclosing routine is expected to take care of these details.

Other variants of the alternate ASI load and store instructions, known as block loads and stores, allow transfer of 64 bytes (512 bits) of data between the processor and main memory (or a suitable device such as the frame buffer), bypassing the cache. The source or destination must be a set of 8 contiguous double-precision registers starting with a multiple of eight, and the memory address must be 64-byte aligned. Load data fill the destination registers over several cycles, and software must be careful not to attempt to perform a block store involving these registers prematurely in order to ensure correctness. Inserting a floating point move instruction from the first destination register to a dummy register will create an interlock, forcing a stall until that register's data arrives.

The restriction on register placement, as well as the need for timing-sensitive code, prevent the use of the block load and store instructions from C. Thus they have not been used for any of the core imaging routines. However, the system copy routine (`memcpy`) as well as frame buffer copies (e.g., for window movement) do make use of block load and store. The destination is copied using double loads and stores and the `faligndata` instruction until a 64-byte boundary is reached. Two block loads are performed to guarantee the presence of the desired (unaligned) 64 source bytes, which are copied into an 8-aligned block of registers. This requires the use of eight sections of code to perform a "crude" alignment, as well as the `faligndata` instruction for "fine-tuning." A new block is loaded, and it along with the previously loaded block provide the data for the next block store, and so on. The last, unaligned destination bytes are again copied in the same manner as the initial segment. A code fragment illustrating this may be found in [Kohn95].

Figure 29 shows the performance of the system `memcpy` routine for spans of up to $1,000,000$ bytes. Each copy was performed at least 50 times in order to smooth out any overhead introduced by the test program, with each call having randomly perturbed source and destination pointers in order to cover different cases of alignment. We see that spans greater than $100,000$ bytes or so are copied at a nearly constant speed of around 175 megabytes/second. Below this, performance appears to vary widely. Figure 30 shows the performance on spans less that $100,000$ bytes in more detail. Since memory copy measurements are often quoted in terms of the aggregate number of bytes read and written across the bus, one can view the block load and store instructions as providing uncached access to memory at 350 megabytes/second.

In principle, there are several cycles available within such a copy loop in which the incoming data may be processed without affecting the transfer rate. Possible applications include lightweight imaging operators, the computation of cryptographic checksums, and data format conversions such as reversing the data's "endian-ness" (i.e., rearranging bytes from `abcd` to `dcba` order in order to compensate for different CPU preferences), or remapping pixel channels from the order required by some application or library (e.g., (r, g, b, α)) into frame buffer order (e.g., (α, b, g, r)).

In section 9.3.1 we discuss prefetching, which should provide an alternative way for future processors to copy data at full memory bandwidth without the need for timing-sensitive software or assembly-language coding.

4.9 Miscellany

`array8`	*xsrc*	*isize*	*idst*
`array16`	*xsrc*	*isize*	*idst*
`array32`	*xsrc*	*isize*	*idst*
`pdist`	*dsrc1*	*dsrc2*	*dsrc3/ddst*

Volumetric imaging is the process of displaying three-dimensional data sampled at a discrete set of locations in space. Typically a two-dimensional slice is obtained for display by stepping through the data along a set of lines. In hardware, this amounts to reading from a sequence of addresses computed from a linear sequence of (x, y, z) coordinates.

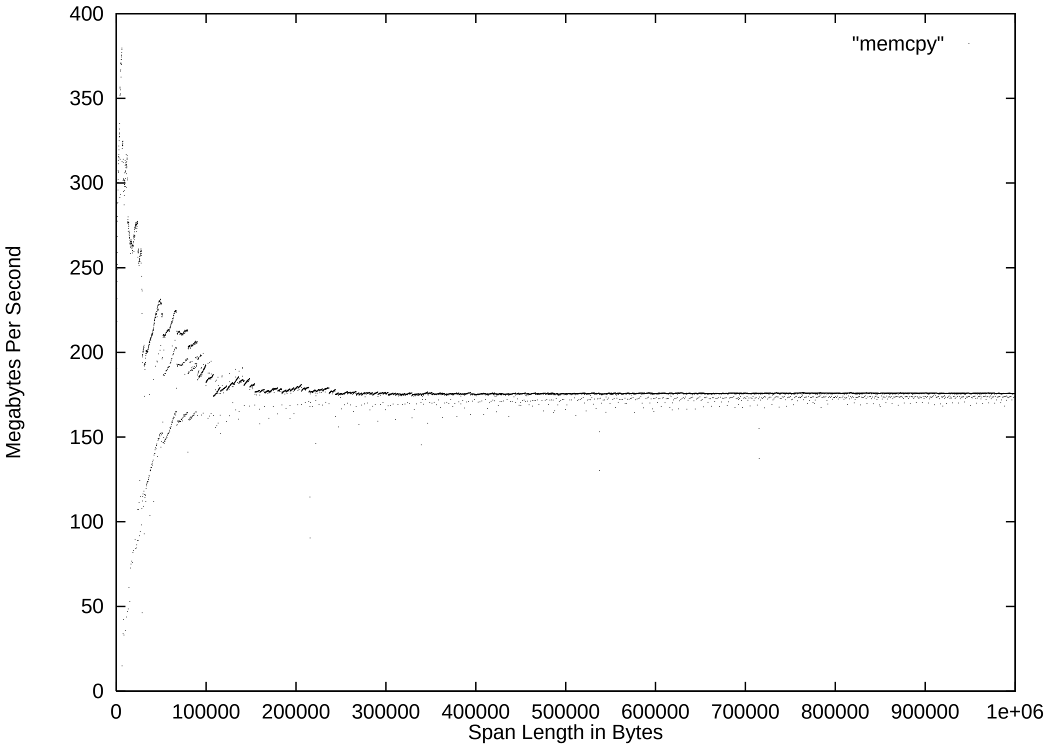

Figure 29: The performance of the system `memcpy` routine.

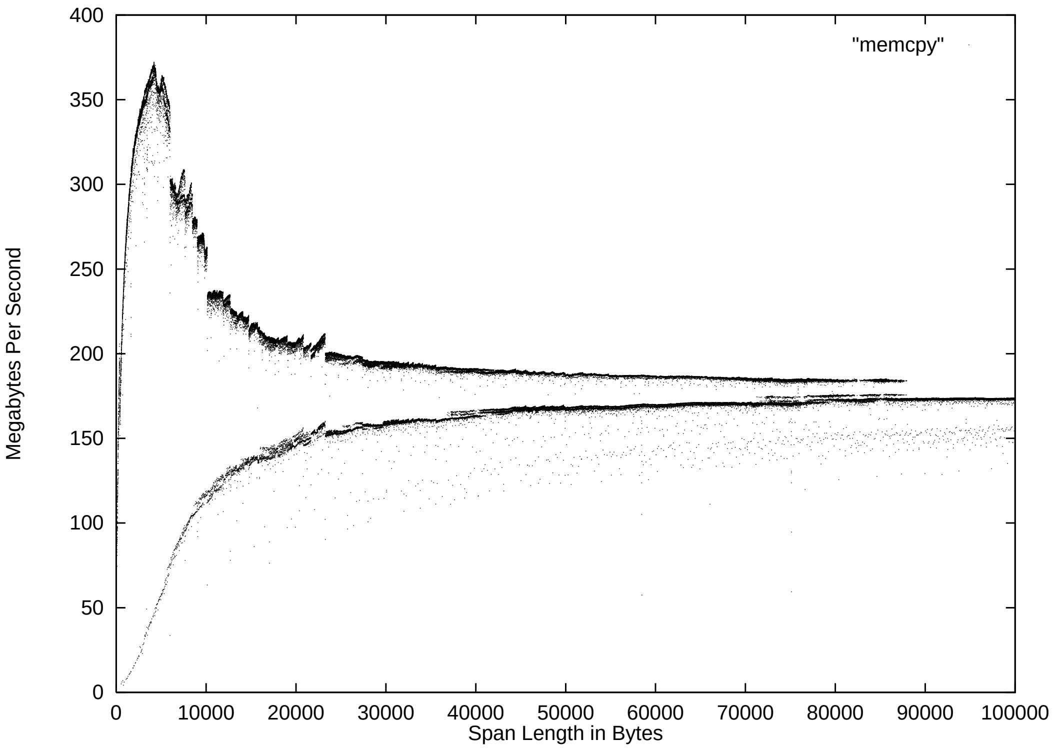

Figure 30: The performance of the system `memcpy` routine on small spans.

63 z 55	54 z frac 44	43 y 33	32 y frac 22	21 x 11	10 x frac 0

Figure 31: Layout of fixed point coordinates for the `array` instructions.

$z_8 \ldots z_5$	$y_{6+\text{size}} \ldots y_6$	$x_{6+\text{size}} \ldots x_6$	$z_4 \ldots z_1$	$y_5 \ldots y_2$	$x_5 \ldots x_2$	z_0	$y_1 y_0$	$x_1 x_0$

Figure 32: Layout of blocked address offsets for the `array` instructions.

The `array` instructions accelerate volumetric imaging by converting (x, y, z) coordinates in *xsrc* into a blocked address, *idst*, that intermingles the bits of the three coordinates into a single array offset. This has the effect of equalizing the changes in the offset as the sample position moves along any direction $(\Delta x, \Delta y, \Delta z)$. If the bits of the three coordinates were simply concatenated, say in zyx order, a line of samples moving principally in the x direction would perform well, since the sequence of addresses would allow full use of each cache line. Movement in y would yield very poor cache line utilization of only 1-4 bytes per line (depending on the datatype being sampled). Movement in z would only rarely reuse cached data and would additionally perform poorly in the TLB (the translation lookaside buffer, where virtual page numbers are mapped to physical ones), since each reference would be to a different page. TLB misses are typically penalized severely, as they require operating system intervention, providing a powerful incentive for code to keep its page locality high.

The second argument of the `array` instructions, *isize*, specifies the size of the x and y dimensions of the array, which must be equal powers of two between 64 (size = 0) and 2048 (size = 5). The actual argument is equal to $\lg(\text{size}) - 6$.

The *xsrc* input to the `array` instructions is a 64-bit (x, y, z) tuple, with 11 bits of fractional positional precision in each dimension. The layout is illustrated in Figure 31. Incrementing of the position may be accomplished by the v9 `addx` instruction, which is a true 64-bit addition. The result is a 32-bit offset, which can be used as the offset for an ordinary load instruction, shown in Figure 32. The application must preprocess the volumetric data into this form. The three variants of the `array` instruction differ only in their significance; `array16` adds one, and `array32` adds two zeros to the right of the blocked offset to facilitate access to 16- and 32-bit array data.

Since the `array` instructions are executed in the IEU, they group naturally with a load, an `addx`, and a `faligndata` for a maximum throughput of one sample per clock (storing every 8 samples). The use of `faligndata` to accumulate bytes will be discussed in section 7.3. Higher quality may be achieved by sampling the corners of a unit cube surrounding the sample point and interpolating based on the x, y, and z fractions, as shown in [Kohn95].

For large arrays, TLB misses will still be problematic. Blocking references in software to small portions of the volume (e.g., $32 \times 64 \times 64$) can help to reduce TLB misses.

Because Solaris 2.5 is a 32-bit operating system, the top 32 bits of the integer registers are not saved across context switches. This would mean that the top half of a fixed point address could disappear at any time as the program is run. Fortunately, it was possible to arrange to have the entirety of registers `%o0`-`%o7` and the `%g` registers saved, so it is possible to use the `array` instructions. It still requires assembly language coding, both to ensure the proper register assignments and to use the `addx` instruction, which the compiler does not now generate (lacking a true 64-bit datatype).

The pdist instruction takes two doubles $p_0 \cdots p_7$ and $q_0 \cdots q_7$, treating them as eight 8-bit pixels. Corresponding pixel values are subtracted, and the sum of the absolute values of the differences are added to the existing integral value in ddst: $d \leftarrow d + \sum_{i=0}^{7} |p_i - q_i|$. The main application of pdist is in the calculation of motion estimation during video encoding, to provide an estimate of the difference between corresponding blocks between two frames in the video sequence. Video compression standards such as MPEG and H.261 (for real-time video conferencing) can increase compression ratios by locating similarities between frames and encoding only the differences. The comparison need not be performed against raw image data; for example, in so-called *half-pel* interpolation one of the sources is allowed to be interpolated at pixel centers or edge midpoints. An experimental H.261 encoder written by the author makes use of the pdist instruction, resulting in a dramatic decrease in the time spent performing motion estimation.

4.10 Useful Pseudo-Operations

A number of inline templates (a mechanism for accessing assembly instructions from C, described in section 6.1) that do not correspond directly to VIS instructions have been defined. Most deal with access to register pairs. In the SPARC architecture, some of the floating point registers may be accessed both as a single-precision value or as one half of a double-precision value. Register pairs always begin with an even-numbered register.

Consider the instruction sequence shown in Figure 33. We require some way to specify which piece of the loaded data the fexpand instructions are to operate on. We define inlines vis_read_hi and vis_read_lo as returning, respectively, the high and low parts of a vis_d64 as a vis_f32. The code in the figure, except for the final store, may then be expressed in C as:

```
vis_d64 s1, s1_hi, s1_lo, s2, s2_hi, s2_lo, sum_hi, sum_lo;
vis_d64 *s1addr, *s2addr, *daddr;
vis_f32 pack_hi, pack_lo;

s1 = *s1addr; s2 = *s2addr;
s1_hi = vis_fexpand(vis_read_hi(s1)); s1_lo = vis_fexpand(vis_read_lo(s1));
s2_hi = vis_fexpand(vis_read_hi(s2)); s2_lo = vis_fexpand(vis_read_lo(s2));
sum_hi = vis_fpadd16(s1_hi, s2_hi);   sum_lo = vis_fpadd16(s1_lo, s2_lo);
pack_hi = vis_fpack16(sum_hi);        pack_lo = vis_fpack16(sum_lo);
```

We still require a way to combine two vis_f32 variables and store them as a single vis_d64. For this purpose we define the inline vis_freg_pair.

The final store may now be expressed as:

```
*daddr = vis_freg_pair(pack_hi, pack_lo);
```

4.11 VIS Instruction Latencies

In Table 2, FGA instructions are: fmov* (register copy), fpadd*, fpsub*, faligndata, fpmerge, and fexpand. FGM instructions are: fpack*, fmul*, and fcmp*. Numbers in brackets represent cross-precision latency, as when part of a 64-bit result is used as a 32-bit argument. A simple example is the sequence:

```
faligndata %f10,%f12,%f14
fmul8x16   %f14,%f30,%f32
fmul8x16   %f15,%f30,%f34
```

Since registers %f14 and %f15 are written as a pair and then used by separate instructions, the latency is 2 rather than 1 (faligndata $\rightarrow$ FGM).

```
ldd     [s1addr+0x0],%f6   ! double load,         writes %f6-%f7
ldd     [s2addr+0x0],%f30  ! double load,         writes %f30-%f31
fexpand %f6,%f10           ! single->double,      writes %f10-%f11
fexpand %f7,%f12           ! single->double,      writes %f12-%f13
fexpand %f30,%f14          ! single->double,      writes %f14-%f15
fexpand %f31,%f16          ! single->double,      writes %f16-%f17
fpadd16 %f10,%f14,%f24     ! all arguments double, writes %f24-%f25
fpadd16 %f12,%f16,%f38     ! all arguments double, writes %f38-%f39
fpack16 %f24,%f40          ! double->single,      writes %f40
fpack16 %f38,%f41          ! double->single,      writes %f41
std     %f40,[daddr+0x0]   ! double store,        writes %f40-%f41 to memory
```

Figure 33: Assembly code for a simple VIS addition.

Result used by:	FGA	FGM	pdist
fmov*	1	1	[2]
fpadd* fpsub* faligndata fpmerge fexpand	1	1[2]	[2]
fpack	3	1[4]	[2]
fmul* pdist	3	3[4]	1

Table 2: Selected UltraSPARC-I VIS instruction latencies.

5 Sources of Speedups

Although the speedup of a particular operator compared to an equivalent software implementation is simple to estimate, it is far more difficult to understand its impact in the context of real programs. Consider just a handful of the complexities of accelerating real code:

- The need to make either/or coding decisions, such as the placement of a value in either an integer or a floating point register, mean that the theoretically fastest operation is not always available within the context of a given piece of code.
- Instructions that take advantage of parallelism require the existence of appropriate quantities of parallelizable computation. An algorithm exposing this parallelism may be slower in other respects, e.g., it might recompute expressions that could otherwise be shared so as to avoid serializing on their computation.
- Particular compiler optimizations may apply to some instructions and not to others. Thus a nominally fast approach that defeats a powerful optimization may be slower in practice than a more naïve one.
- Some operators involve a trade-off between speed and accuracy that cannot be determined without human input. Since the operations are not exactly equivalent, the

notion of speedup is ill-defined. The number of operations required may be a function not only of the quantity of data but of the desired accuracy as well.

- Many algorithms are resource-constrained by their use of a particular functional unit. Lessening the usage of the other units will not improve performance. In general, Amdahl's law implies that the speedup attributable to any feature will be limited by its frequency. Our situation is even worse: the speedup due to lessening use of a non-limiting resource is zero even though that resource might be used as often as the limiting resource.

Four features of VIS and UltraSPARC-I stand out as providing clear performance gains. The superscalar instruction issue of UltraSPARC-I can potentially accelerate all programs; by using the floating point units for VIS processing, the potential for parallel issue is increased. The alignment and partitioned arithmetic operators operate in tandem to increase the amount of useful work performed in each instruction cycle. The unconditional clamping feature of the `fpack` instructions allows conditionals to be avoided, removing conditional branches which act as a barrier to some loop optimizations. Lastly, by opening up the floating point register set to imaging codes, greater parallelism is exposed.

5.1 Superscalarity

The gain in performance due to superscalarity is of course available to all programs. However, traditional imaging codes make heavy use of the integer units while leaving the floating point units nearly unused. By placing fixed point capabilities into the FPU, it becomes possible to issue an addition and a multiplication simultaneously, along with one or two integer operations for pointer indexing, loop control, and so forth.

Typical integer codes appear to have a limited ability to take advantage of superscalarity – there simply are not enough independent instructions available to keep very many functional units busy. This has led to some fairly gloomy assessments of the ultimate utility of superscalar design. Floating point codes fare much better, since they typically perform most of their computation in tight loops. Advanced schedulers may place instructions from a number of different loop iterations into a given issue slot in order to keep the processor busy with useful work. Imaging codes, although they traditionally use little actual floating point, share these characteristics and are amenable to the same sorts of scheduling algorithms.

5.2 Alignment and Partitioned Arithmetic

The alignment and partitioned arithmetic instructions are the most obvious source of speedups. In practice the need for data conversions, alignment, clamping, packing, and other overhead instructions cause the code to require more than simply one half or one fourth as many arithmetic instructions than standard imaging code, so in and of itself partitioned arithmetic does not guarantee large speedups. In particular, the time to reformat data into partitioned format and the time to restore it to its original format must always be minimized if the benefits of partitioned arithmetic are to be realized. For example, if the data reside in integer registers the time to move them into the floating point register file and back may swamp any savings due to partitioning. In such a case a new approach may be required, rather simply inserting partitioned arithmetic into existing routines.

5.3 Unconditional Clamping

The value of automatic clamping is difficult to quantify. By removing the need for a conditional in many loops, optimization is radically improved. Thus simply by removing the need to consider clamping, a more aggressive treatment of the loop as a whole is enabled.

Of course, conditionals are not the only way to implement clamping. Conditional moves, lookup tables, and clever sets of logical operations [Granlund92] are also valid techniques. The cost of these methods in a superscalar context will probably depend most strongly on the relative use of the functional units in the loop; if the load/store unit is heavily used, use of a lookup table will add cost, whereas in other loops the additional use would be free.

5.4 Use of Floating Point Registers

Superscalar scheduling techniques make heavy use of registers to store intermediate results. This is a general consequence of any technique that exposes parallelism. The ability of the compiler to schedule loops depending on integer registers is limited by the number of available registers for all but fairly simple loops. The compiler will be forced to adopt looser and looser schedules until the register pressure becomes manageable. Moving the computation to the floating point units opens up a large pool of registers that would otherwise be wasted. This is especially true for the v9 SPARC instruction set architecture, which devotes twice as many bits to the floating point register set as did v8.

6 Compilation Technology

VIS code, of course, may be written directly in assembly language. This gives the programmer full control over timing, register allocation, and the like. However, the need to observe the grouping rules and instruction/instruction latencies make this a very difficult task. In addition, all the usual difficulties of assembly-language coding apply. Given the resources at hand, an assembly-only XIL port would have had extremely limited functionality. Nonetheless, it was considered a serious option since there was no guarantee of adequate compiler support, and only finally rejected as the needed support materialized.

In this section we discuss the basic technique which is used to include VIS instructions in C code, inline assembly language templates. During the development of the VIS XIL port, a simulation environment was used which we discuss briefly.

The performance of VIS in practice is in large part determined by the quality of the code emitted by the compiler. The code generator relies on a machine model describing the instruction grouping rules and instruction/instruction latencies; this model allows for reasonable scheduling of straight-line code, but more importantly serves as a basis for more sophisticated loop scheduling techniques. We discuss the techniques which are used for instruction scheduling and register allocation with the aim of clarifying the scope of the compiler's contribution to performance. This knowledge has proved highly useful in the process of VIS code development, since it is often preferable to use a simple algorithm that will be well optimized rather than a more complex one.

6.1 Inline Templates

Since hand-coding in assembly language is unacceptably difficult, a technique for mixing C code with VIS instructions is required. Many compilers offer an `asm` keyword that allows a

```
.inline vis_fpadd16, 4
std     %o0,[%sp+0x48]  ! A scratch stack location
ldd     [%sp+0x48],%f2
std     %o2,[%sp+0x48]
ldd     [%sp+0x48],%f4
fpadd16 %f2,%f4,%f0
.end
```

Figure 34: An inline template for the fpadd16 instruction.

```
! val0 is in registers %f0-%f1
! val1 is in registers %f2-%f3
!  sum is in registers %f4-%f5
!
std     %f0,[scratchaddr0]
ldd     [scratchaddr0],%o0      ! Load into %o0-%o1
std     %f2,[scratchaddr1]
ldd     [scratchaddr1],%o2      ! Load into %o2-%o3
std     %o0,[scratchaddr2]
ldd     [scratchaddr2],%f1002
std     %o2,[%sp+0x48]
ldd     [%sp+0x48],%f1004
fpadd16 %f1002,%f1004,%f1000
fmovd   %f1000,%f4
```

Figure 35: Conceptual code generated by insertion of the template.

short sequence of assembly instructions to be introduced into a C function. Sun's SPARCWorks C compiler uses a variant of this technique in which assembly code is placed in a separate file with the suffix `.il`. An example of such a template is shown in Figure 34. The "4" in the first line refers to the number of input values – double-width values (`double` and `long long` arguments) are presented as an unaligned pair of integer registers. The result is left in registers `%f0` and `%f1` by convention.

Consider using this inline within the context of a C function. Given a source statement:

```
vis_d64 val0, val1, sum;
sum = vis_fpadd16(val0, val1);
```

the variables `val0`, `val1`, and `sum` may be thought of as residing in some double-precision registers. The inline template must be called with its (initial) arguments in the `%o` integer registers. This is only a convention, and we shall see that it does not affect the generated code. Figure 35 shows the code that would result from a near-literal inclusion of the inline within the context of the enclosing statement. The arguments to `vis_fpadd16` are copied through memory into integer registers `%o0`-`%o4`. Next, the template is inserted with its registers and scratch addresses renamed to some unique values. This is possible since the compiler uses virtual register names during code generation in any case. Finally, the result must be copied into its proper location.

If this code were to be run, there would of course be a large penalty for all this copying. Fortunately the redundant code is easily eliminated: the pattern `std–ldd–std–ldd` can be removed by simply identifying the initial and final registers in the sequence. In the example, `%f0` and `%f1002` are coalesced, as are registers `%f2` and `%f1004`. The final `fmovd` can also be eliminated by coalescing `%f1000` and `%f4`. All that is left is simply:

```
fpadd16 %f0,%f2,%f4
```

and no cost is incurred by the use of the inline calling convention.

6.1.1 Prototypes

It is imperative that proper ANSI function prototypes be used at all times. In this case, a correct prototype would be:

```
vis_d64 vis_fpadd16(vis_d64, vis_d64)
```

If no prototype were used, the implicit conversions between `int` and `double` would be employed, resulting in code containing `fitod` and `fdtoi` type-conversion instructions. These conversions will not maintain the fixed-point bit patterns used by VIS, but will instead treat the data as IEEE 754 floating point. Since we are using a floating point resource, namely the register file, to hold non-floating point data, we must always be careful to manipulate it using only VIS-aware operations, or operations that perform no conversions. Floating point loads, stores, and moves never alter the bit patterns of the data they handle, and so are safe to use.

Tables of VIS constants to be generated once and textually included in source code as array initializers should not be printed using the `stdio` library routine `printf()` using the `%lf` format conversion operator; even if a particular version of the C library allows a bit-for-bit round-trip of ordinary `double` values through `printf()` and `scanf()`, this will not necessarily hold true for today's `printf()` and some future version of `scanf()`. Worse, some VIS constants will not be valid floating point numbers and will print as `NaN` (not a number, i.e., an illegal bit pattern) or `Infinity` and will not be recoverable. The correct approach is to print the data in an integral format with a well-defined byte order; this will always be recoverable exactly.

In practice, the result of incorrect prototypes or variable declarations often results in no compile-time error messages due to the standard conversions defined by ANSI C. At runtime, data will typically be completely scrambled. However, an unintended conversion from float to double and back may leave data intact and thus remain unnoticed. This is easily detected by the extreme speed penalty for operations on the denormalized and `NaN` patterns being converted, which cause the processor to trap to software routines. A search through the assembly code for floating point conversion operators will also turn up these problems.

6.2 Simulating VIS Code

The VIS/XIL development process began prior to the tapeout of UltraSPARC-I. A set of simulation environments were constructed to allow development to proceed without hardware.

The initial simulators available to the XIL porting group, known as `siam` and `incas`, were implemented by the chip designers as modules of the `mpsas` simulator framework. Both implement a model of the UltraSPARC processor running in a nearly bare machine – one with no operating system services beyond a rudimentary initialization sequence. However, `siam` focuses on fast execution, while `incas` is "nearly cycle-accurate" (hence the "nca" of its name). Within `incas`, it is possible to examine the state of the processor pipelines during a

```
union vis_dreg_overlay {
  vis_d64 d64;
  vis_f32 f32[2];
  vis_u32 u32[2];
  vis_s32 s32[2];
  vis_u16 u16[4];
  vis_s16 s16[4];
  vis_u8  u8[8];
  vis_s8  s8[8];
  unsigned long long ull;
  struct {
    vis_ras u, l;
  } x;
};

vis_d64
vis_fpadd16 (vis_d64 frs1, vis_d64 frs2)

{
  union vis_dreg_overlay op1, op2, dest;

  op1.d64 = frs1;
  op2.d64 = frs2;

  dest.s16[0] = op1.s16[0] + op2.s16[0];
  dest.s16[1] = op1.s16[1] + op2.s16[1];
  dest.s16[2] = op1.s16[2] + op2.s16[2];
  dest.s16[3] = op1.s16[3] + op2.s16[3];

  return dest.d64;
}
```

Figure 36: A portion of the `vis_sim.c` file.

particular cycle, which is useful in understanding the practical effects of the grouping rules. Cycle accuracy is essential for any performance-oriented work.

A third simulation environment, produced within the XIL porting group, took the form of a C module, `vis_sim.c`. A portion of this file is shown in Figure 36. This module contains a function corresponding to each VIS inline template, as well as some `union` datatypes that allow access to the VIS data using regular integer instructions. When linked with code using VIS, a portable binary executable is generated that can run on any SPARC processor. Although such executables run at less than 1/100 the speed of the optimized UltraSPARC binaries, they allowed the use of existing hardware, compilers, and debugging tools during the period before those tools became VIS-aware. Even after the hardware was made available, it was sometimes in short supply. By providing an option in the project `makefile`s to produce simulated code, it was possible to work without interruption. Comparing the results of the simulated code and the production code also provided an ongoing way to locate compiler bugs, since the older, more stable version of the compiler would not otherwise have been able to accept VIS code.

6.3 Instruction Grouping Rules

The legal instruction groupings for UltraSPARC-I are quite complex. Here we attempt to provide a few simple rules that are of particular interest to the user of VIS.

A number of instructions are always dispatched singly; i.e., they cannot be grouped with any other instructions. In particular, block load and store and the `alignaddr` instruction fall into this category.

Instruction that make use of the integer units must reside in the first three slots of a group. Only one of the integer units contains a shifter, thus only a single shift instruction may be dispatched per cycle. Similarly, only one `array` or `edge` instruction may be dispatched in a given cycle.

Cycles will be broken if the same destination register (other than `%g0`) is used multiple times (i.e., a write-after-write hazard). Read-after-write hazards, in which the destination register of one instruction is used as a source in a later instruction, also force the breakage of a group. However, the result of most integer instructions may be forwarded to a store instruction in the same group.

When using the VIS comparison instructions, care must be taken to place at least two cycles of separation between the comparison and any instruction that uses the resulting bit mask. If this is not done, the second instruction may be dispatched prematurely and allowed to execute up to its writeback stage. It will then be canceled an restarted, resulting in a 9 cycle overall penalty.

6.4 Superscalar Code Scheduling

Code scheduling algorithms for superscalar machines have been heavily influenced by techniques from early hardware scheduling algorithms, microcode scheduling, and VLIW. Surveys may be found in [Lam90] and [Johnson91]. It has been appreciated for some time that simply widening a processor's instruction issue capacity does not result in a commensurate increase in throughput without aggressive scheduling. Even with such scheduling, many important integer codes have such high branch frequency that parallelism appears difficult or impossible to uncover. Fortunately, imaging codes have ample parallelism and are excellent candidates for straightforward basic block and loop scheduling techniques. More advanced techniques, such as *trace scheduling*, that look beyond branch boundaries, are probably unnecessary and possibly disadvantageous for the algorithms we are considering; they will not be discussed here.

6.4.1 Basic Block Scheduling

A *basic block* is a sequence of instructions that can be entered only from the top and exited only from the bottom. If one instruction in a basic block executes, they all must execute. This property, along with the fact that basic blocks are trivial to locate, has made them a popular focus of optimization for many years. As long as the correct result has been computed before the block exits the compiler may rearrange the intermediate computation freely (although exception behavior must still be considered).

Consider the computation described by the dependency graph in Figure 37. This graph corresponds to the code shown in Figure 33. The nodes of the graph are instructions (numbered for identification) and the edges represent *data dependencies* existing between pairs of instructions. A *flow dependency* is said to exist between a pair of instructions when the output register of the first instruction is used as an input register of the other. Two other

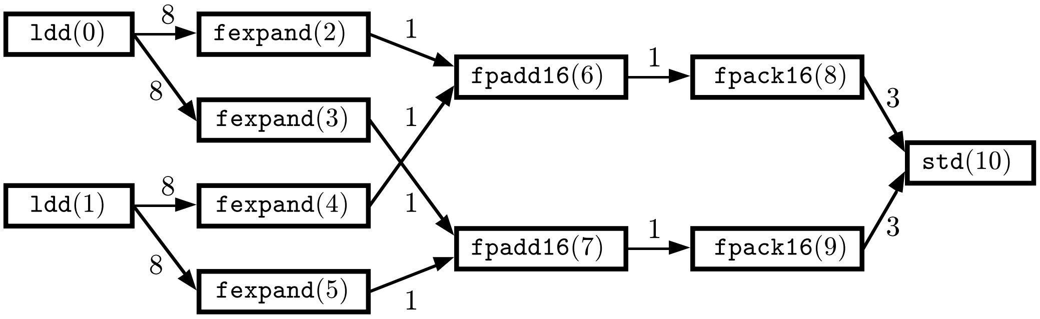

Figure 37: Dependency graph for a simple addition loop.

types of dependencies are commonly represented in dependency graphs: *anti-dependencies* in which an instruction overwrites an input register used by a previous instruction, and *output dependencies* in which two instructions write to the same output register. All three types of dependencies prevent the pair of instructions from being reordered. The latter two types are artifacts of the reuse of registers; if we posit an infinite set of virtual registers, reuse is no longer necessary and only flow dependencies can occur. There are also *control dependencies* that link conditional statements and the instructions they select between; since we are discussing basic blocks only, these dependencies are not relevant.

Each edge of the dependence graph may be labeled with the number of cycles of latency between the instructions it connects. For example, if the data to be loaded are in level 2 cache then 8 cycles of separation between the `ldd` and `fexpand` instructions will be required in order to avoid a stall. The other edges have latencies determined by the design of the various pipelined functional units and any bypasses that may exist between them. In the case at hand, all the edges between the computational instructions have latency 1. The results of the `fpack16` instructions are available to be written to memory after 3 cycles.

The critical path in a dependency graph is the path of greatest total latency. If a code sequence is to be executed in isolation, the critical path provides an upper bound on its peak performance. In the case at hand, there are several critical paths of length 13 extending from (either) initial `ldd` instruction to the final `std` instruction. Thus the store can be performed no earlier than cycle 13. The actual performance will usually be slower than that implied by the critical path since resource constraints are not considered. Assume a load is issued in cycle 0. The second load cannot be issued until cycle 1; the `fexpand` instructions that depend on it cannot be performed any sooner than cycles 9 and 10. The data for the final store will thus be ready no sooner than cycle 15, 2 cycles after the original critical path-based estimate.

Finding an optimal schedule given a dependency graph and a model of allowable instruction issues is in general NP-complete. Accordingly heuristics are used to achieve near-optimal schedules quickly. The most popular is *list scheduling*, first described by Fisher [Fisher79]. This is a greedy approach in which an instruction, once scheduled, is never moved. The dependency graph is constructed and instructions having no outstanding dependencies are identified as available for scheduling. Initially these are instructions whose dependencies have been satisfied by the code prior to the loop or by previous iterations.

An available instruction is assigned to the earliest issue slot at which its dependencies are satisfied and an appropriate functional unit is available. New instructions are added to the available list as their dependencies are satisfied. The key to achieving good schedules is the priority function that determines which of several available instructions is to be scheduled. Typically the priority of an instruction is based on the length of its chain of successors in order to make maximal progress along the critical path.

Any loop may be considered simply as a basic block that happens to be one of its own dynamic successors, and can be scheduled on that basis. There are, however, substantial benefits to giving loops special consideration in the scheduler, as we will see in the following sections.

6.4.2 Loop Unrolling

An obvious technique for increasing the throughput of a loop is to replicate the loop body. This results in a new loop that performs the same computation in a single iteration that the original loop performed in several iterations. This has at least two benefits. First, the time spent on loop overheads such as comparison, pointer updating, and branching is reduced in proportion to the unroll factor. On scalar processors, this is the main benefit and the original impetus for the technique. Second, the combined loop bodies form a larger basic block, with an attendant increase in opportunities for scheduling instructions in parallel. Observe that the critical path length for a set of independent iterations is the same as that of a single iteration; if some instructions must be performed in one iteration before the next one can commence, only those instructions lie on the overall critical path. In either case, we obtain a smaller lower bound on the time to complete the full set of iterations than that derived from simply multiplying the iteration time by the number of iterations. As a practical matter, by reducing the loop scheduling problem to one of scheduling basic blocks, we also leverage the existing optimization capability of the compiler.

As an imaging loop executes, it passes through a number of discrete iterations. At the beginning of each iteration, values are typically brought in from memory to be operated on; after a certain latency, some arithmetic expressions involving the data are evaluated. This evaluation typically involves some independent subexpressions and is thus amenable to scheduling across multiple units. Eventually the results are written back to main memory. The resulting pattern of resource usage begins at zero as the iteration begins, climbs to the loop's peak resource usage level where it plateaus for a while, and finally descends again to zero. The effect of loop unrolling is to lessen the number of occasions during the lifetime of the loop where the resource usage drops below its peak – the plateaus are extended and overall throughput is increased.

Another use for loop unrolling is suggested by the technique of *partial evaluation* (see section 9.2.1). If the number of loop iterations (its *trip count*) can be determined at compile time, the loop can be unrolled completely, and a near-optimal schedule for that number of iterations determined. Complete compile-time knowledge of the trip count is not realistic, but specialized loops for a selection of trip counts may be generated, with the final selection taking place at run time. For some algorithms such as texture mapping (section 7.8) of small triangles, trip counts of under 10 or so may account for the bulk of the run time. The usual reliance on asymptotically fast schedules breaks down since the overhead cannot be amortized over many iterations for these algorithms. Specializing for small trip counts would appear to offer a benefit in such cases.

Some loops contain values that are forwarded between adjacent iterations. For example,

Iteration:	0	1	2	3	4	5
Prologue	stage 0					
	stage 1	stage 0				
	stage 2	stage 1	stage 0			
Kernel	stage 3	stage 2	stage 1	stage 0		
		stage 3	stage 2	stage 1	stage 0	
			stage 3	stage 2	stage 1	stage 0
Epilogue				stage 3	stage 2	stage 1
					stage 3	stage 2
						stage 3

Figure 38: The execution of a 4-stage pipelined loop.

the loop in Figure 13 forwards the value `im1_data1`, copying it into the variable `im1_data0` for use by the following iteration. Suppose the loop is unrolled twice. The following code illustrates how the two variables may be used in alternation to eliminate the need for copying:

```
im1_data = align(im1_data0, im1_data1, 2);
im1_data0 = *im1_aligned++;
im1_data = align(im1_data1, im1_data0, 2);
im1_data1 = *im1_aligned++;
```

6.4.3 Software Pipelining

Software pipelining consists of grouping together instructions from different iterations into a single loop body. In this way, progress is made on several iterations at once using the resources at hand. For example, suppose we are computing a dot product. The individual products of the elements are computed in the multipliers; imagine there is some latency l before the results are ready to be processed by the adders. Rather than waiting, we can arrange the computation so that the additions from iteration $i - l$ are performed simultaneously with the multiplications of iteration i. Since there is no dependency between them, they can be issued and processed simultaneously.

Figure 38 shows the execution of 6 iterations of a 4-stage loop. The first three lines show the *prologue* of the loop in which the first three iterations are begun. The *kernel* of the loop comes next; three repetitions are shown but any number may be performed. This is the steady state of the loop, which repeats at some fixed *initiation interval* (II). The *epilogue* of the loop occupies the last three cycles and allows the iterations in progress during the last kernel iteration to complete.

6.4.4 Modulo Scheduling

The most effective software pipelining technique for superscalar processors to date (although originally developed in the context of microcoding and VLIW processing) appears to be *modulo scheduling* [Rau81] [Tirumalai96]. Modulo scheduling proceeds by fixing the initiation interval and attempting to find a satisfying schedule. This schedule will serve as the kernel; the prologue and epilogue are created separately without fixed time constraints. Processing begins with the minimum initiation interval, which is determined by examining the resource constraints of the loop (as in section 2.10). An additional bound on the II can be found by considering computation that must be performed in a given iteration before the following iteration can begin. For example, a loop that updates an accumulator will have such a restriction. This amounts to finding the maximum weighted path length over cycles in the dependency graph. Fortunately many imaging loops have iterations that are independent or nearly so, resulting in a minimum II determined solely by resource usage. If a schedule with the given II cannot be found, the II is incremented. Eventually a satisfying schedule must be found; however, at some point other loop scheduling techniques may yield better performance. In practice the modulo scheduler stops after trying a fixed number of IIs in the interest of compilation time.

Since the target II is known to the scheduler, its task is to allocate instructions to particular kernel cycles. A *modulo reservation table* with II entries is initialized to empty. Each entry can hold a group of instructions that is subject to the target processor's grouping rules, i.e., which can jointly occupy a processor issue slot. The instructions placed in entry i of the table will be executed in cycles $i, i + \mathrm{II}, i + 2\mathrm{II}, \ldots$ at run time (counting the first kernel cycle as 0). Conversely, if we wish a particular instruction to be executed in some cycle j at run time, it must be allocated in cycle j **mod** II of the table. This is known as the *modulo constraint*.

Let us schedule the code from Figure 37. Its minimum II is 6, since there are 6 instructions that require the graphics adder. For now we will ignore any additional instructions required to update pointers and loop indices, detect the loop exit, and branch back to the loop head. These instructions will be simple to schedule later.

Scheduling proceeds much as in list scheduling, with the addition of the modulo constraint. The only instructions free of dependencies are the two initial loads. We place them in slots 0 and 1. This satisfies the dependencies of the four `fexpand` instructions, which may now be scheduled beginning in cycles 8 and 9. The modulo constraint maps cycles 8 and 9 to slots 2 and 3 of the modulo reservation table; the requirement to avoid resource conflicts forces us to distribute the instructions across four cycles, say 8, 9, 10, and 11. The `fpadd16` instructions may be placed as soon as the appropriate pair of `fexpand` instructions have completed, but again the resource requirements interfere and we find that the earliest valid times are cycles 12 and 13, which correspond to slots 0 and 1 of the table. The `fpack16` depending on the earlier `fpadd16` may be placed in the next cycle, and the other one cycle later. Three cycles after the conclusion of the two `fpack16` instructions, in cycle 17, the `std` is scheduled. The resulting modulo reservation table is shown in Figure 39, and three iterations of the resulting loop are detailed in Figure 40. Cycles 12-17 show a complete copy of the kernel. The primes indicate the iteration to which each instruction belongs. The number of iterations executing simultaneously is given by dividing the length of a complete iteration by the kernel size and taking the ceiling; in this case, an iteration requires 18 cycles and the kernel occupies 6 cycles, so 3 iterations are processed together. This quantity will become significant when we attempt to assign registers, and is known as the *kernel unroll factor* (KUF).

The literature on modulo scheduling contains discussions of hardware schemes such as a *rotating register file* [Rau92] which allows the same kernel code to refer to a different set of registers during each iteration. On a more conventional processor such as UltraSPARC-I, the loop kernel must be unrolled KUF times, with each instance of the kernel code using a distinct set of virtual registers. Of course, the register allocator is free to assign these to physical registers as it sees fit (see section 6.5.1). The main disadvantages of this technique are increased code size and register pressure. It is also impractical to modulo schedule all but the most trivial loops by hand because of this KUF expansion.

```
0: ldd(0)   fpadd16(6)
1: ldd(1)   fpadd16(7)   fpack16(8)
2:          fexpand(2)   fpack16(9)
3:          fexpand(4)
4:          fexpand(3)
5: std(10)  fexpand(5)
```

Figure 39: The modulo-scheduled loop kernel.

```
 0: ldd                           15:         fexpand'
 1: ldd                           16:         fexpand
 2:                               17: std     fexpand'
 3:                               18:         fpadd16'
 4:                               19:         fpadd16'    fpack16'
 5:                               20:         fexpand''   fpack16'
 6: ldd'                          21:         fexpand''
 7: ldd'                          22:         fexpand''
 8:         fexpand               23: std'    fexpand''
 9:         fexpand               24:         fpadd16''
10:         fexpand               25:         fpadd16''   fpack16''
11:         fexpand               26:                     fpack16''
12: ldd''   fpadd16               27:
13: ldd''   fpadd16    fpack16    28:
14:         fexpand'   fpack16    29: std''
```

Figure 40: The first three loop iterations of the modulo scheduled loop.

A loop iteration may require values from one or more previous iterations. The difference in iteration numbers between the producer and consumer of such values is known as Ω, and is included as an annotation to the appropriate arc of the loop's dependency graph. For small values of Ω, it is possible to simply have the code for one iteration use source registers which are written by a previous iteration of the unrolled kernel. In this way values may be forwarded from iteration to iteration without additional copying or use of memory.

The modulo scheduling technique as we have described it, and as it is implemented in the current Sun compiler, does not allow any branching within the loop. This allows the scheduler full freedom to reorder instructions without concern for any incorrect state that might be generated following an early exit. In addition, the loop must have a trip count that can be determined upon loop entry. Use of a `for` loop with a simple continuation test is the simplest way to ensure this. The loop bound should be determined by a variable that is not altered within the loop; use of pointer dereferencing, as in the test `count < x->count`, may cause the compiler to fail to recognize the loop as pipelineable. It also appears that variables of type `int` should be used for the loop index in order for the loop to be recognized as pipelineable. Variations on modulo scheduling exist that allow early loop exists and other branching structures, but the author has not had experience with them.

Sun's modulo scheduler is invoked by default. However, it is desirable to have a way of determining whether it succeeded in scheduling a particular loop. The compiler option flags `-S -Qoption cg -ms_pipe,-Qms_pipe+D3` will produce an assembler output file with debugging information, including the achieved II for each scheduled loop. Occasionally a loop is determined to have a large minimum initiation interval due to pessimistic assumptions about aliasing between different pointers in the loop. The compiler provides a mechanism for informing the scheduler that no data written in a given iteration will be read by any of the next `n` iterations, and vice versa. The directive `#pragma pipeloop(n)` should be placed on the line prior to the beginning of the loop. A value of 0 for `n` means that no aliasing occurs. The results of such a directive when aliasing does in fact exist are undefined.

6.5 Register Allocation

Under the assumption that there are infinitely many registers, it is simple to use the software pipelining techniques described above to produce optimal schedules. In reality, of course, registers are a finite resource. It is important to understand how registers are allocated in a modern compiler in order to make reasonable estimates of the feasibility of successfully pipelining a given loop.

All but the simplest allocation problems are NP-complete. Thus we should expect some register allocation problems to require an exponential amount of work (assuming, of course, that $P \neq NP$ as is customary). Fortunately, actual instances of register allocation problems are rarely so intractable. Furthermore, a strictly optimal solution is not always required; it is acceptable to use more than the minimal number of registers as long as the actual number of available registers is not exceeded. The code may also be altered slightly if need be.

Early C compilers used registers mainly to cache intermediate values within an expression, keeping variables in fixed stack locations and updating them on each assignment. Programmers could use the `register` keyword to suggest that a particular variable was worthy of being kept in a register throughout a particular scope. Inclusion of a language structure to deal with variable allocation has had the effect of training programmers to minimize the number of variables in a function with the aim of having them all assigned to registers. We shall see that this strategy may produce suboptimal code in a compiler with a proper register allocator.

6.5.1 Register Coloring

Chaitin [Chaitin81] [Chaitin82] describes a general technique for allocating variables to registers using *graph coloring*. Graph coloring is the assignment of colors to the nodes of a graph such that no two nodes sharing an edge are assigned the same color. The *chromatic number* of a graph is the minimal number of colors sufficient to color it. The famous 4-color theorem (all maps may be colored with 4 colors so that no two countries sharing a border are colored alike) is a statement about graph coloring, namely that all planar graphs are 4-colorable.

In the case at hand, the input to the allocator is a piece of intermediate code that uses an unlimited number of *virtual registers*. An *interference graph* is defined such that the nodes correspond to these registers, with an edge between each pair of virtual registers that cannot be stored in the same physical register due to a conflict. Conflicts are found by identifying the *live ranges* of each virtual register; that is, the section of code in which they are in use. Two virtual registers conflict if their live ranges overlap. If the target machine has k usable registers, a k-coloring of the interference graph is attempted.

Graph coloring is NP-complete, and so is expected perform exponentially slowly on some inputs. Surprisingly, determining whether a graph is k-colorable requires only constant *average* time, where the average is taken over all possible graphs [Wilf86]. It is unclear what effect the actual distribution of graphs encountered in register allocation would have on this result, and the literature generally assumes the necessity of some heuristic approach.

If the allocator fails to find a k-coloring, the code is altered, either by *spilling* (saving and restoring) some values to memory around each use, by splitting the code into segments to be colored separately, or by recomputing the value in question (known as *rematerialization*). This process repeats until a k-coloring is found. The choice of heuristics to control the coloring process and to determine which values to spill has been the focus of much research, and is beyond the scope of this report.

What does the use of such a register allocator mean for the programmer? First, it renders the `register` keyword meaningless. Although it is possible to force some compilers to place a given variable in a register throughout its scope, and to exempt that register from consideration during the coloring phase, it is likely that the compiler's spill heuristic will be more accurate than the programmer's judgment. If the coloring succeeds, nothing has been gained by the `register` specification, and if it fails there will be spill code that might be costlier than that of spilling the `register` variable. Second, fewer variables are not necessarily better. By reusing the same variable for multiple purposes, the compiler may be fooled into allocating a single virtual register for that variable. This virtual register will have a longer live range than necessary, and so will increase the number of interference edges and possibly the chromatic number of the interference graph. More sophisticated compiler analysis techniques can avoid this by recognizing that the multiple uses of the variable may be assigned to different virtual registers.

The use of register pairs introduces an additional difficulty into the register allocation problem, since some values must be kept in aligned, adjacent registers instead of being free to be stored anywhere. If single- and double-precision registers are considered on equal terms during the allocation, the demand for registers will in effect be overestimated. Briggs et al. describe a technique for coloring register pairs [Briggs92] that can avoid some of these problems.

The SunPro SPARCCompiler 4.0, development versions of which were used for VIS library work, required a significant engineering investment to deal properly with the allocation demands of VIS code. In particular, the extensive use of constructs such as `vis_read_hi` and `vis_freg_pair`, which imply the use of register pairs, often resulted in failure to perform certain optimizations. The general solution involved better heuristics to determine when the use of single-precision virtual registers `%f`$(2n)$ and `%f`$(2n+1)$ followed by the use of a double-precision virtual register `%f`$(2n)$ was actually required to be mapped to a physical register pair or was merely a case of masquerading so-called "evil twins." Some production VIS XIL code is not modulo scheduled due to the existence of this problem in the compiler used to build the library.

7 VIS Applications

A number of distinct imaging algorithms have been coded to use the visual instruction set. Together they illustrate some of the potentials of VIS, as well as its practical complexities. In the following we will examine a number of algorithms that have been ported to VIS. The development of this code has been a team effort; however, the author had a direct influence over all the algorithms shown here. Some of these algorithms also appear (without analysis) in the VIS User's Guide [SME95b].

We discuss eight areas in which VIS has been used with success: clamped addition, alpha blending, table lookup, convolution, bicubic and bilinear resampling, color space conversion, and texture mapping.

7.1 Addition With Clamping

We saw above in section 3.2 how to implement clamped addition of two images in C. Of course, this example was somewhat contrived since the desired behavior corresponds exactly to that of VIS, but the function is nonetheless required by XIL and other libraries. The main purpose of this section is to describe the alignment and edge masking operations common to most simple VIS loops; modifying this loop to perform some other bytewise arithmetic or logical operation is trivial.

To add pixel values in VIS, the image data must be converted from 8- to 16-bit format. This can potentially be done in three ways:

- Use the `fexpand` instruction to convert a value x to $16x$ (i.e., insert 4 zero bits before and after the bits of x);
- Use the `fpmerge` and `fzero` instructions to intermingle zero bytes with the image bytes; or,
- Use the `fmul8x16`, `fmul8x16al`, or `fmul8x16au` instructions to multiply each value by some constant, usually a power of two.

The `fexpand` and `fpmerge` instructions are both processed by the graphics adder, so there is no distinction between them in terms of performance. Using a multiplication, even by $16 \cdot 256 = 4096$ (mimicking `fexpand`), may sometimes be advantageous in order to balance the resource usage of an otherwise FGA-heavy routine.

Assume for the moment that the input and output data are stored in doubleword-aligned buffers, i.e., arrays of `vis_d64`s. Then a simple loop suffices, shown in Figure 41.

The resource constraints on this loop are 3 loads and stores, 6 graphics adder instructions, and 2 graphics multiplier instructions (the `fpack16`s). Changing two of the `fexpand`s into `fmul8x16al` operations balances the use of the graphics units at 4 operations each. Thus this loop can potentially process 8 bytes in 4 clocks, or .5 clocks/byte.

Real images will not be conveniently aligned. Consider writing to a contiguous but unaligned set of `width` destination bytes starting at an address `dptr`; either single-byte stores or partial stores will be necessary to write some of the output. The partial store case illustrates the use of the `edge8` instruction to generate the masks. Before entering the loop, the address of the last byte is computed as `dlast = dptr + width - 1`; forgetting to subtract 1 is a common programming error. The `edge8` instruction is used with `dptr` and `dlast` to generate an initial mask. Note that the correct mask will be generated even if `width` is small enough that the entire image span is contained in a single doubleword. This mask will be used in the initial loop iteration. Once this mask has been generated, the pointer `dalign` is computed by masking off the lower 3 bits of `dptr`. This will be the address used by the first store.

After data are written to the memory at `dalign`, it is incremented by 8 and the `edge8` instruction is used again to generate the next mask. Although the `edge8` instruction generates a condition code that could be used to exit the loop, it is difficult to make use of it from C as well as undesirable from an optimization point of view, since the trip count becomes unpredictable as explained in section 6.4.3. Instead, the total number of loop iterations is computed in advance as:

```
void
add_images (vis_d64 src1[], vis_d64 src2[], vis_d64 dst[], int count)

{
  int i;
  vis_d64 s1, s2, ss1_hi, ss1_lo, ss2_hi, ss2_lo, dd_hi, dd_lo;

  vis_write_gsr(3 << 3); /* Shift right 7 - 3 = 4 places. */

  for (i = 0; i < count; ++i) {
    s1 = src1[i];
    s2 = src2[i];

    ss1_hi = vis_fexpand(vis_read_hi(s1));
    ss2_hi = vis_fexpand(vis_read_hi(s2));
    dd_hi = vis_fpadd16(ss1_hi, ss2_hi);

    ss1_lo = vis_fexpand(vis_read_lo(s1));
    ss2_lo = vis_fexpand(vis_read_lo(s2));
    dd_lo = vis_fpadd16(ss1_lo, ss2_lo);

    dst[i] = vis_freg_pair(vis_fpack16(dd_hi), vis_fpack16(dd_lo));
  }
}
```

Figure 41: Using VIS to add data from aligned arrays.

```
times = ((unsigned long) dlast >> 3) - ((unsigned long) dalign >> 3) + 1
```

The resulting function is showed in Figure 42.

All that remains is to deal with varying source alignments. Instead of the aligned pointers `src1` and `src2`, the source data will be presented as unaligned pointers `sptr1` and `sptr2`. A given span of eight source bytes beginning at an unaligned pointer `sptr1` may be processed by aligning `sptr1` to form `salign1`, and reading data from `*(salign1)` and `*(salign1 + 1)`. Alternatively, we can use array notation: `salign1[0]` and `salign1[1]`. These two data values are then realigned using an `faligndata` instruction; the `%gsr` offset must be set to the offset of `sptr1`. This process may be summed up as:

```
salign1 = vis_alignaddr(sptr1, 0);
sptr1 += 8;
s1a = salign1[0];
s1b = salign1[1];
s1 = vis_faligndata(s1a, s1b);
```

This alignment must take the destination alignment into account as well, since the first several bytes being written may actually precede `dptr` (and be masked away by the initial `edge_mask`). It is possible to deal with all cases in a straightforward manner by prepending each source with some “virtual” bytes, equal in number to the “virtual” (masked) destination bytes. This number is equal to `dptr - dalign`, which in turn is just the lower three bits of `dptr`. This offset needs to be subtracted from `sptr1` before beginning the source alignment process:

```
d_offset = (unsigned long) dptr & 0x3;
salign1 = vis_alignaddr(sptr1, -d_offset);
```

```
void
add_images (vis_d64 src1[], vis_d64 src2[], vis_u8 *dptr, int width)

{
   int i, times, edge_mask;
   vis_u8  *dlast;
   vis_d64 *dalign;
   vis_d64 s1, s2, ss1_hi, ss1_lo, ss2_hi, ss2_lo, dd_hi, dd_lo, result;

   vis_write_gsr(3 << 3); /* Shift right 7 - 3 = 4 places. */

   dlast = dptr + width - 1;
   edge_mask = vis_edge8(dptr, dlast);
   dalign = (vis_d64 *) ((unsigned long) dptr & ~0x3);

   times = ((unsigned long) dlast >> 3) - ((unsigned long) dalign >> 3) + 1;
   for (i = 0; i < times; ++i) {
     s1 = src1[i];
     s2 = src2[i];

     ss1_hi = vis_fexpand(vis_read_hi(s1));
     ss2_hi = vis_fexpand(vis_read_hi(s2));
     dd_hi = vis_fpadd16(ss1_hi, ss2_hi);

     ss1_lo = vis_fexpand(vis_read_lo(s1));
     ss2_lo = vis_fexpand(vis_read_lo(s2));
     dd_lo = vis_fpadd16(ss1_lo, ss2_lo);

     result = vis_freg_pair(vis_fpack16(dd_hi), vis_fpack16(dd_lo));
     vis_pst_8(result, dalign, edge_mask);
     ++dalign;
     edge_mask = vis_edge8(dalign, dlast);
   }
}
```

Figure 42: Addition using the `edge8` operation.

The second source is treated similarly.

The computation of `d_offset`, `salign1` and `salign2` may be performed outside of the loop. The `%gsr` must be set anew prior to each invocation of `faligndata`, however. Although the `alignaddr` instruction is the fastest way of accomplishing this, we note that the return value is not particularly interesting since in a given iteration `i` we will always read from `salign1[i]` and `salign1[i + 1]`. All that is really required is the offset of `sptr1 - d_offset`, which remains constant as `sptr1` is incremented by 8 bytes. Accordingly, this quantity may be computed once and stored in `s1_offset`; future uses of `alignaddr` will have a null pointer as their first argument. Figure 43 shows this process.

A further optimization is to recycle the value `s1b` for use as `s1a` in the following iteration:

```
s1b = salign1[i + 1];
(void) vis_alignaddr((void *) 0, s1_offset);
s1 = vis_faligndata(s1a, s1b);
s1a = s1b;
```

```
int s1_offset = ((unsigned long) sptr1 - d_offset) & 0x3;

for (i = 0; i < times; ++i) {
  s1a = salign1[i];
  s1b = salign1[i + 1];
  (void) vis_alignaddr((void *) 0, s1_offset);
  s1 = vis_faligndata(s1a, s1b);

  /* Perform the rest of the computation. */
}
```

Figure 43: Use of a constant source alignment factor.

This has the effect of reducing the load bandwidth back to one load per source per iteration. Copying `s1b` into `s1a` may even be free if the loop is to be unrolled or software pipelined, as explained in sections 6.4.2 and 6.4.3.

The above transformation has a further benefit: since the loads from `salign1` involve a linear sequence of addresses, it is simple for the compiler to apply loop optimizations that reorder the loads with respect to the other instructions of the loop body. In the case where the result of the `alignaddr` instruction is used, the compiler will treat the `alignaddr` as an opaque operation and accordingly the pointer will be available for dereferencing only after it has completed. This places a constraint on which portion of the loop may contain the load. In terms of the dependency graph of the loop, the loads have become descendents of the `alignaddr`, and not only the `faligndata` instruction. This will tend to increase the initiation interval of the loop, by restricting the flexibility of the scheduler.

Since the `faligndata` instructions require the use of the graphics adder, we now have 8 adder cycles (2 `faligndata`, 4 `fexpand` and 2 `fpadd16`) and 2 multiplier cycles (2 `fpack16`) in use. Thus it is optimal to convert 3 of the `fexpand` instructions into `fmul8x16au` instructions. This yields a minimum initiation interval of 7, since the 2 `alignaddr` instructions each occupy an entire cycle on their own. Thus the peak speed of this loop will be $7/8 = 0.875$ clocks/pixel. In practice the current modulo scheduler finds a schedule with an initiation interval of 10 cycles, for a peak performance of $10/8 = 1.25$ clocks/pixel. The final loop is shown in Figure 44.

It is possible to support child images with a subset of their parents' channels by allowing only the bytes corresponding to the desired channels to be written to the output. An additional mask is maintained and logically "and"ed with `edge_mask` in the partial store instruction. This mask is then rotated left 8 positions at the end of the loop for use by the next iteration. Initializing the mask is best done by a small precomputed table indexed by the image's pixel stride and the subset of bands to be written. This adds two shifts and three logical operations to the overall loop computation, which in the present case would not involve the loop's FGA/FGM resource constraint and would accordingly not affect performance. The rotation proceeds identically for images of any reasonable pixel stride, subject to the constraint that there exist some valid mask length x, $8 \leq x \leq 32$ which is a multiple of the stride. The bitmask will have x valid bits; the rotation is relative to these bits only. An expression to perform the rotation in terms of shifts and logical operations is:

```
mask = ((mask >> (x - 8)) & 0xff) | (mask << 8);
```

Another variation on the above loop is adding a set of constants, one per channel, to a single source. The constants may be stored in expanded format and substituted for the ss2_hi and ss2_lo in the arguments to the fpadd16 instructions. Naturally, all the code dealing with the second source is eliminated. The only difficulty is adding the proper constant for each band. For a 1-banded image, the constant may simply be expanded four times into a single constant, which is used for all additions. Images with 2 or 4 bands are handled similarly, replicating the constants twice or using them as-is, respectively. For 3-banded images (containing, say, RGB data), the lack of a common factor between 3 and 8 (the number of bytes processed in a loop iteration) forces the loop to be unrolled 3 times. Three constants are generated, containing the constants corresponding to $RGBR$, $GBRG$, and $BRGB$, respectively. Each constant will be used twice during the loop. A similar unrolling is required if some bands are to be masked out; three 8-byte masks are generated prior to the loop entry, each with a different bit pattern corresponding to $RGBRGBRG$, $BRGBRGBR$, and $GBRGBRGB$.

Timing data are presented in section 8.1.

7.2 Blending Using a Mask Image

The process of blending (or *compositing*) two images using an *alpha* image is used in imaging whenever a portion of an image is to be superimposed on another[1]. Simply selecting each output pixel from one of the two images results in harsh and unrealistic edges; more subtle edges are produced by accounting for the relative areas within each output pixel covered by each of the sources. Compositing is frequently used for this purpose within the context of interactive image editing, and so can benefit greatly from hardware performance enhancement. The handling of multiple channels and edges is identical to that of the addition loop and we will not dwell on it further.

The compositing function we wish to implement is given by the linear interpolation formula:

$$p_{\text{out}} = (1-\alpha)p_{\text{in}_1} + \alpha p_{\text{in}_2}$$

In practice, α is taken from an 8-bit image, and lies within the range $[0, 255]$, so we can rewrite this as:

$$\begin{aligned} p_{\text{out}} &= (1 - \frac{\alpha}{255})p_{\text{in}_1} + \frac{\alpha}{255}p_{\text{in}_2} \\ &= p_{\text{in}_1} + \frac{\alpha}{255}(p_{\text{in}_2} - p_{\text{in}_1}) \end{aligned}$$

It is tempting to approximate division by 255 by a division by 256, which can be replaced by a shift. Indeed, the VIS multiplier performs this shift implicitly. This suggests the code:

```
ss1    = vis_fexpand(src1);
ss2    = vis_fexpand(src2);
prod   = vis_fpsub16(vis_fmul8x16(alpha, ss2),
                     vis_fmul8x16(alpha, ss1));
blend  = vis_fpadd16(ss1, prod);
result = vis_fpack16(blend);
```

which does indeed produce an answer within 1 of the correct result.

A possible difficulty with this method is that it systematically underestimates α by dividing it by 256 rather than by 255. Repeated compositing may allow small errors to accumulate and ultimately exhibit visible error. Worst of all, an α value of 255 can result in

[1]The code example in this section is similar to code developed by Xiaoping Hu.

```
void
add_images (vis_u8 *sptr1, vis_u8 *sptr2, vis_u8 *dptr, int width)

{
  int i, times, edge_mask;
  int s1_offset, s2_offset, d_offset;
  vis_u8  *dlast;
  vis_d64 *s1align, *s2align, *dalign;
  vis_d64 ss1_hi, ss1_lo, ss2_hi, ss2_lo, dd_hi, dd_lo, result;
  vis_d64 s1a, s1b, s1, s2a, s2b, s2;
  vis_f32 sixteen;

  vis_write_gsr(3 << 3); /* Shift right 7 - 3 = 4 places. */
  *((short *) &sixteen) = 16 << 8; /* fmul8x16au shifts right by 8. */

  dlast = dptr + width - 1;
  edge_mask = vis_edge8(dptr, dlast);
  dalign = (vis_d64 *) ((unsigned long) dptr & ~0x7);
  d_offset = (unsigned long) dptr & 0x7;

  s1align = vis_alignaddr(sptr1, -d_offset);
  s1_offset = ((unsigned long) sptr1 - d_offset) & 0x7;
  s1a = s1align[0];

  s2align = vis_alignaddr(sptr2, -d_offset);
  s2_offset = ((unsigned long) sptr2 - d_offset) & 0x7;
  s2a = s2align[0];

  times = ((unsigned long) dlast >> 3) - ((unsigned long) dalign >> 3) + 1;

  for (i = 0; i < times; ++i) {
    s1b = s1align[i + 1];
    (void) vis_alignaddr((void *) 0, s1_offset);
    s1 = vis_faligndata(s1a, s1b);
    s1a = s1b;
    s2b = s2align[i + 1];
    (void) vis_alignaddr((void *) 0, s2_offset);
    s2 = vis_faligndata(s2a, s2b);
    s2a = s2b;
    ss1_hi = vis_fexpand(vis_read_hi(s1));
    ss2_hi = vis_fmul8x16au(vis_read_hi(s2), sixteen);
    dd_hi = vis_fpadd16(ss1_hi, ss2_hi);
    ss1_lo = vis_fmul8x16au(vis_read_lo(s1), sixteen);
    ss2_lo = vis_fmul8x16au(vis_read_lo(s2), sixteen);
    dd_lo = vis_fpadd16(ss1_lo, ss2_lo);
    result = vis_freg_pair(vis_fpack16(dd_hi), vis_fpack16(dd_lo));
    vis_pst_8(result, dalign, edge_mask);
    ++dalign;
    edge_mask = vis_edge8(dalign, dlast);
  }
}
```

Figure 44: The final VIS addition loop.

errors since $\lfloor(255 \cdot 255 + 128)/256\rfloor = 254$. A pure white pixel in the second input image (of value 255) lying under a mask α value of 255, which is meant to denote "no change," will be darkened to a value of 254. If the blend operation is to be used as part of a sequence, this may be a serious flaw; areas outside the area to be composited should not be changed. An application allowing the user to drag an object to be composited would not be able to simply composite the object with the background using the bounding box of the masked area but would have to use a hard selection mask as well. Even so, the overall computation will have more error than we might like. However, this low-precision loop is still quite useful, e.g., as a way of compositing image layers directly prior to display, since the errors will be purely visual and will not be magnified by further processing.

Timing data are presented in section 8.1.

7.3 16- to 8-Bit Table Lookup

Even a non-arithmetic operation such as table lookup can benefit from the use of VIS[2]. We define a 16- to 8-bit table lookup as follows:

```
vis_s16 *src;
vis_u8  *dst;
vis_u8  table[65536], *table_base;

table_base = &table[32768]; /* Middle of table. */
for (i = 0; i < count; ++i) {
  dst[i] = table_base[src[i]];
}
```

The source pixels are in the range $[-32768, 32767]$. This operation is useful for display of 16-bit images on an 8-bit frame buffer. A fixed mapping such as $x' = (x + 32768)/256$ does not allow for dynamic adjustment of the image contrast. If the image contains important details with values in the range $[1000, 2000]$ they will map to the range $[131, 135]$ – not enough to see much detail. Instead, a piecewise-linear mapping (a *window leveling* or *contrast-stretching* function) like the one shown in Figure 45 is frequently used.

Additional refinements such as a non-linear mapping to compensate for monitor gamma or lighting conditions may also be employed; using a lookup table allows all of these possibilities to be combined arbitrarily at a fixed cost.

This loop can potentially operate at a rate of 3 clocks per pixel on UltraSPARC-I, since there are 3 load/store unit operations involved. Although the loads from the table cannot be reduced, the source loads and destination stores can be combined. Ideally we could store 8 pixels at once, necessitating 16 bytes worth of loads. Although theoretically two loads would suffice, we will use four since the C compiler does not yet use v9 instructions to handle 64-bit integers. We thus obtain a limit of 13 loads and stores for 8 pixels, for a maximum throughput of 1.625 clocks/pixel, a speedup of $(3/1.625) \approx 1.8\times$.

In order to implement this idea, we require something more than the techniques we tried in section 3.2, which resulted in a large quantity of shifts, offsetting any benefits. Fortunately VIS provides an alternative way to join the output pixels together. We can use short loads (section 4.8) to load bytes from the table into double registers. A newly loaded value may be joined to a set of previously accumulated values by the statement:

```
accum = vis_faligndata(value, accum);
```

[2]The code examples in this section were developed collaboratively with Peter Farkas.

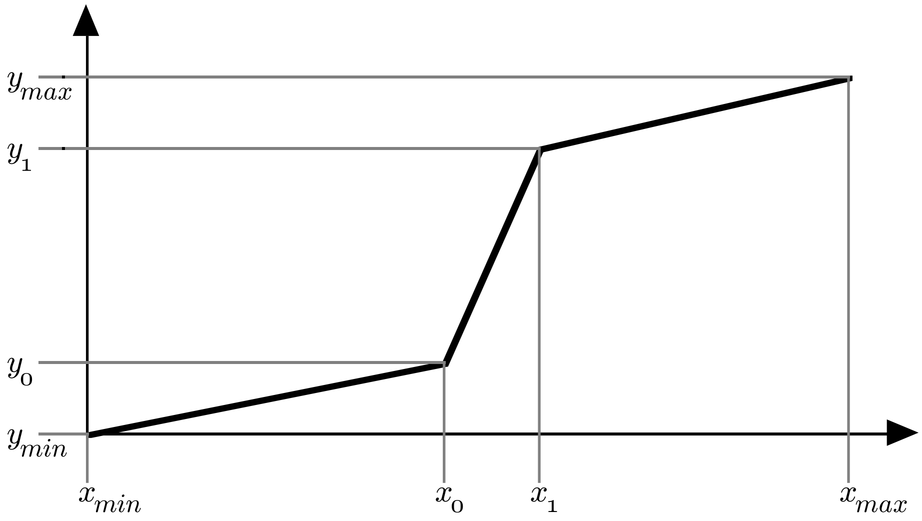

Figure 45: A piecewise-linear mapping to perform window leveling.

with a fixed %gsr alignment of 7.

Note that the values must be inserted in reverse:

```
vis_d64 byte0, byte1, /* ... */, byte7, *dst;
int offset0, offset1, /* ... */, offset7;

byte7 = vis_ld_u8_i(table, offset7);  /* Do the same for 6, ..., 1. */
byte0 = vis_ld_u8_i(table, offset0);

accum = vis_faligndata(byte7, accum); /* Do the same for 6, ..., 1. */
accum = vis_faligndata(byte0, accum);
*dst++ = accum;
```

Since there are more loads and stores than faligndata instructions, the latter should not add to the execution time. In addition, the fact that we are loading into the floating point register file increases the odds that the loop can be pipelined with minimal initiation interval, even with a large latency assumed for the loads. Similar approaches involving the use of fpmerge or fpack32 to intermingle the output bytes are also possible.

The input data remain to be aligned. Since we are loading 2-byte quantities in 4-byte groups, there are two possible cases of alignment. If the input is aligned we load and use four words; in the unaligned case we will initially load five words use half of the first and last words. The last word is recycled and four new words are read in each iteration. Figure 46 illustrates this process.

The double shifts are required in order to properly sign-extend the lower halves of word0, $\cdots$, word4. Since there are 12 shifts and 13 loads and stores, the loop may be scheduled with an initiation interval of 13.

Timing data are presented in section 8.2.

```
switch(align) {
case 0:
  offset0 = word0 >> 16;        offset1 = (word0 << 16) >> 16;
  offset2 = word1 >> 16;        offset3 = (word1 << 16) >> 16;
  offset4 = word2 >> 16;        offset5 = (word2 << 16) >> 16;
  offset6 = word3 >> 16;        offset7 = (word3 << 16) >> 16;
  /* Perform table lookup. */ break;
case 1:
  offset0 = (word0 << 16) >> 16;  offset1 = word1 >> 16;
  offset2 = (word1 << 16) >> 16;  offset3 = word2 >> 16;
  offset4 = (word2 << 16) >> 16;  offset5 = word3 >> 16;
  offset6 = (word3 << 16) >> 16;  offset7 = word4 >> 16;
  /* Perform table lookup. */ break;
}
```

Figure 46: Aligning the sources for 16- to 8-bit lookup.

7.4 Convolution With Small Kernels

Convolution, with its heavy demand for both addition and multiplication, provides a good test case for VIS[3]. Consider convolution with a general 3×3 kernel, conceptually requiring 9 multiplications and 8 additions. Utilizing partitioned arithmetic, we require 18 multiplications, 16 additions, and 2 `fpack16` operations to process eight bytes of output (ignoring reads for the moment). The FGM thus dominates, yielding a theoretical performance of 2.5 clocks/byte. Things are not so rosy, however – the source row will be require some alignment before it can be used. Perhaps more than one alignment will be required since the additions will require their sources to be relatively aligned. An upper bound on this process is one alignment per kernel element for each eight bytes of output, since we can simply grab spans of bytes from the source and align them without regard to duplication. This adds 9 FGA operations to the mix. It also adds a number of `alignaddr` instructions, each occupying a full cycle.

A straightforward approach to implementing general $n \times n$ convolution with VIS is to read n^2 (overlapping) spans of image data, multiplying each one by the appropriate coefficient, summing the products, and storing the result. This approach naturally takes advantage of the parallel addition and multiplication capabilities of VIS to produce a span of output pixels simultaneously. The total VIS computation required to apply an $n \times n$ kernel ($3 \leq n \leq 7$) to an 8-pixel span is $2n^2$ `fmul8x16`, $2(n^2 - 1)$ `fpadd16`, and 2 `fpack16` operations. The factors of two derive from the fact that $8 = 2 \cdot 4$ values are being processed using 4-way operations.

We can assume that the destination span is doubleword-aligned, performing a small non-VIS startup loop otherwise. The input data will require realignment using the `alignaddr` and `faligndata` instructions. A simple approach to reading the input is to calculate the desired (unaligned) address for each load and to use the `alignaddr` instruction to set the `%gsr` and to produce and aligned address. Two doublewords are then read starting at this address, and the results are processed by an `faligndata` instruction. The problems with this approach are threefold. First, it requires n^2 `alignaddr` instructions, each occupying a full cycle. Second, the overlapping of the source data reads are increased even further;

[3]The code examples in this section were inspired by code written by Steve Howell, Ray Roth, and Jaijiv Prabhakaran. Algorithms to deal with symmetry were suggested by Alex Mou.

a total of $2n^2$ loads are needed in order to acquire only $n^2 + 7n$ unique bytes ($8 + n - 1$ bytes from each of n scanlines), and no values are reused between loop iterations. Third, the addresses of the loads cannot be known until after the `alignaddr` instruction has been issued (see section 7.1), increasing the height of the loop's dependency graph.

A better way to deal with the inputs is to read a new doubleword from each of the n scanlines with each loop iteration, using n loads, and align it to the destination with n `alignaddr` and n `faligndata` instructions. The buffered data may be used as-is for the multiplications with the entries along the left edge of the kernel but must be realigned using fixed offsets of $1, \cdots, n-1$ for each scanline; the `%gsr` is set to each value once and n alignments are performed. The result is a total of $n+(n-1) = 2n-1$ `alignaddr` instructions and $n + n(n-1) = n^2$ `faligndata` instructions. We save $2n^2 - n$ loads and $n^2 - 2n + 1$ `alignaddr` instructions at the cost of maintaining some buffered values. Since performance will be determined mainly by the $2n^2 - 2$ `fpadd16` instructions, the n^2 `faligndata`, and the $2n - 1$ alignaddr instructions, the theoretical performance is $3n^2 + 2n - 3$ cycles per 8 pixels. At 167 MHz, this yields a theoretical peak speed of 44.5, 16.2, and 8.4 million convolutions per second for kernels of width 3, 5, and 7 respectively.

Figure 47 illustrates the alignment process described above for a 3×3 kernel. The variables `offset0`, `offset1`, and `offset2` contain the offsets of the initial pixels of the three source rows contributing to the convolution, and `salign0`, `salign1`, and `salign2` contain pointers to the rows that have been aligned to an 8-byte boundary. Note that in the context of an application, where the layout of images can be controlled, forcing the scanline stride of the source image to be a multiple of 8 will ensure that all vertically neighboring pixels share the same alignment; in the example at hand, this would eliminate the need for two of the `alignaddr` instructions. In the context of a library where general image layouts are to be accommodated, this optimization may not be possible. We do not use this optimization in the VIS XIL code timed below.

The first section of the loop sets the `%gsr` and aligns previously read values `sa0` and `sb0` to produce a new value `s00_next`, which is properly aligned for multiplication by the upper-left kernel entry. The raw value `sb0` is copied into `sa0` for use during the following iteration, with `sb0` acquiring a new value from memory. The next two blocks perform the same operation for the other two input rows.

The values `s00` and `s00_next` (and their counterparts for the following scanlines) must now be combined to form input spans suitable for multiplication by the middle and right columns of the kernel, using constant alignment factors of 1 and 2. This produces values `s01`, `s11`, etc. Finally `s00_next` is copied into `s00` and similarly for the other rows. The remainder of the loop multiplies each of the spans by a corresponding fixed-point kernel element and sums the partial results. The final value is written to the destination.

The convolution algorithm described thus far does not take advantage of any special features that the kernel may possess. For example, some common image-processing kernels contain many zero elements. For these elements, it is unnecessary to produce the aligned spans and to perform the corresponding multiplication and addition.

Another common special case is symmetry. Consider a kernel symmetric along its vertical axis:

$$k = \begin{bmatrix} a & b & a \\ c & d & c \\ e & f & e \end{bmatrix}$$

One way to speed up convolution with such a kernel is to note that we compute sums of

```
for (i = 0; i < times; ++i) {
  (void) vis_alignaddr((void *) 0, offset0);
  s00_next = vis_faligndata(sa0, sb0); sa0 = sb0; sb0 = salign0[i + 2];
  (void) vis_alignaddr((void *) 0, offset1);
  s10_next = vis_faligndata(sa1, sb1); sa1 = sb1; sb1 = salign1[i + 2];
  (void) vis_alignaddr((void *) 0, offset2);
  s20_next = vis_faligndata(sa2, sb2); sa2 = sb2; sb2 = salign2[i + 2];

  (void) vis_alignaddr((void *) 0, 1);
  s01 = vis_faligndata(s00, s00_next);
  s11 = vis_faligndata(s10, s10_next);
  s21 = vis_faligndata(s20, s20_next);
  (void) vis_alignaddr((void *) 0, 2);
  s02 = vis_faligndata(s00, s00_next);
  s12 = vis_faligndata(s10, s10_next);
  s22 = vis_faligndata(s20, s20_next);
  s00 = s00_next; s10 = s10_next; s20 = s20_next;

  /* Multiply s_ij by k_ij, sum, and store to destination. */
}
```

Figure 47: Aligning sources during a 3×3 convolution.

the form $a \cdot p_{i-1} + b \cdot p_i + a \cdot p_{i+1}$, which may be rewritten as $a \cdot (p_{i-1} + p_{i+1}) + b \cdot p_i$. This transformation eliminates a multiplication at the cost of an addition. Such a trade-off appears attractive at first, but in fact the loop as written is already bound by the instruction issue limitation of the graphics adder (since `faligndata` is processed there). Furthermore, raw pixel values may not be added without first performing expansion, negating any gain.

Another possible way to take advantage of symmetry is to notice that certain values are computed multiple times as the kernel moves across the source data. For example, with the symmetrical kernel described above, the value $a \cdot p_{i+1}$ will be computed on behalf of the convolution centered at pixel p_i and well as that centered at pixel p_{i+2}; if this partial product could be forwarded between iterations, a multiplication would be saved. Note that a kernel with both horizontal and vertical symmetry contains at most three distinct values, so in principle only three multiplications per source pixel are required. However, the fact that we are processing 8-pixel spans means that the forwarded value will not be meaningful for the next loop iteration, so we cannot take direct advantage of this approach either.

Vertical symmetry may be utilized within the context of the scheme outlined above. Since the first and last rows of the kernel are the same, the sum of products involving `s00`, `s01`, and `s02` may be reused two rows later. Since the overall alignment scheme requires horizontal motion through the source in order to properly amortize the cost of the readahead, the sum of the products from the first row will have to be stored in a buffer that will be reused two row in the future. Since this buffer is internal to the convolution code, no additional alignment will be required.

Separable convolutions are those which may be rewritten as the product of two one-dimensional convolutions:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} \cdot \begin{bmatrix} d & e & f \end{bmatrix} = \begin{bmatrix} ad & ae & af \\ bd & be & bf \\ cd & ce & cf \end{bmatrix}$$

For example, the kernel generated by $[1\ 2\ 1]^T \cdot [1\ 2\ 1]$ is a useful, nearly rotationally symmetric low-pass filter.

Each row of the input is multiplied by the n elements of the row vector, and the products are added using $n-1$ additions; the results are themselves multiplied by the elements of the column vector and added together. In principle, a separable convolution thus requires only $2n$ multiplications and $2n-2$ additions, as opposed to the n^2 multiplications and n^2-1 additions for the general case.

The basic algorithm for separable convolution is to perform a single new convolution with the horizontal kernel per row; the results of the previous n such convolutions are then convolved with the vertical kernel. It is easiest to move vertically through the data, although buffering is also possible. In the case of VIS there are some complications, since the result of a product of 8-bit pixel data and a 16-bit kernel coefficient will be in 16-bit form. Multiplying this by a kernel element will require two multiplications and an addition (using `fmul8sux16` and `fmul8ulx16`). The total number of multiplications is therefore $3n$. The number of additions similarly rises to $3n-3$.

Convolution by a separable kernel with a symmetrical vertical part may partake in both sorts of optimizations described, since the result of the horizontal convolution may be multiplied immediately by the repeated vertical kernel element prior to buffering. When the buffer entry is reused this multiplication will not have to be performed again.

Some performance figures for the general 3×3 case are given in section 8.3.

7.5 Resizing Using Bicubic Interpolation

The task of image resampling ("zooming" or "shrinking") is ideally suited to VIS acceleration. It is related to convolution, the main difference being that the filter kernels are applied to a set of source positions dependent on both scaling and translation factors for each axis. For general information about image resampling, see [Wolberg90]. This section will illustrate the use of multiple stages and buffering to implement a complex algorithm using VIS.

The main work to be performed for each destination pixel is a two-dimensional interpolation to compute the image value at a backward-mapped source location. This interpolation may sample the single source pixel whose center is closest to the backwards-mapped location, in which case it is known as *nearest-neighbor* interpolation; it may sample a four-pixel neighborhood using horizontal and vertical distances to the pixel centers for weighting (*bilinear* interpolation); or it may sample a sixteen-pixel neighborhood using weights derived from a cubic polynomial (*bicubic* interpolation). Various polynomial filters may be used, generally combining a windowed $\sin(x)/x$ function and some sort of sharpening. The filters used are *separable*; that is, they may be computed equivalently as a single two-dimensional convolution or two one-dimensional convolutions. This has the effect of reducing the effective kernel size from n^2 to $2n$ for filters of width n. In practice, the results of the first convolution are reduced down to 8 bits for processing by the second convolution (with some loss of precision). This makes the two convolution steps symmetric in terms of their input and output data formats, reducing the resampling problem to two independent phases, one horizontal and one vertical. The process is illustrated in Figure 48. Figure 49 compares the effects of nearest-neighbor, bilinear, and bicubic sampling on a low-resolution synthetic bitmap of the letter 'A'; the bitmap was scaled by a factor of two using nearest-neighbor, bilinear, or bicubic resampling and then nearest-neighbor resampled in order to show the individual pixels. It should be noted that for natural scenes, which tend to have few sharp

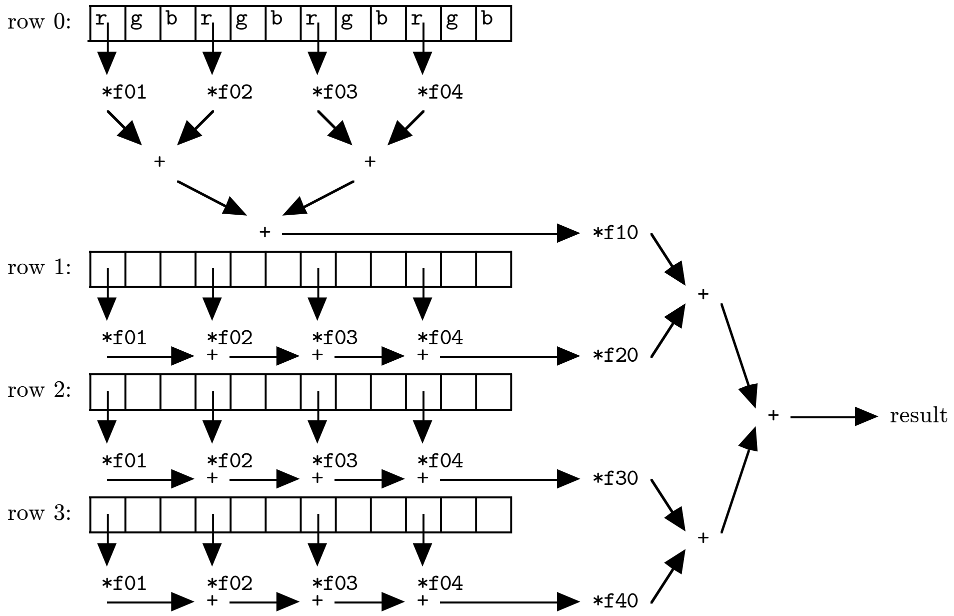

Figure 48: Separable resampling using a 4×4 neighborhood.

edges, the artifacts of these techniques are less noticeable. Figure 50 shows the effects of resampling a photographic image with a large scale factor. Note the directional artifacts around bright areas in the bilinearly resampled image; this is the primary reason why bilinear resampling is frequently considered visually unacceptable. The bicubically resampled image also shows better definition and greater extremes of light and dark.

Consider the vertical phase first, even though it will occur after the horizontal pass during the actual resampling process. A weighted sum of several scanlines is performed, and the results packed and stored to the destination. Assuming the source and destination images have identical widths and pixel formats, the processing proceeds in lockstep, consuming one byte from each source scanline for each byte of the destination. Each byte of a particular source is multiplied by a filter coefficient that remains constant over the destination scanline. Figure 51 illustrates this process in pseudo-code.

Since the filter product is independent of j, it can be trivially vectorized. All that remains is the determination of the filter coefficients. Ward [Ward89] suggests precomputing coefficients for a fixed number of "bins," i.e., quantized subpixel positions. As long as the scale factor is small in comparison to the number of bins, the error can be made comparable to that introduced by rounding. This technique reduces the filter computation to a one-time polynomial evaluation plus a single per-scanline lookup. The selection of source scanlines is obvious except at edges, where at least three reasonable strategies exist: refuse to write output scanlines for which not all source lines are available; clamp source indices to the extremes of the image, in effect replicating the bordering pixels of the source data; or treat values outside the source as constant, say zero. These schemes are simple to implement in the context of a two-pass algorithm since the selection of source scanlines is performed explicitly. The source scanline pointers may simply be duplicated in order to implement replication, or a pointer to an array of zeros provided to implement zero-padding.

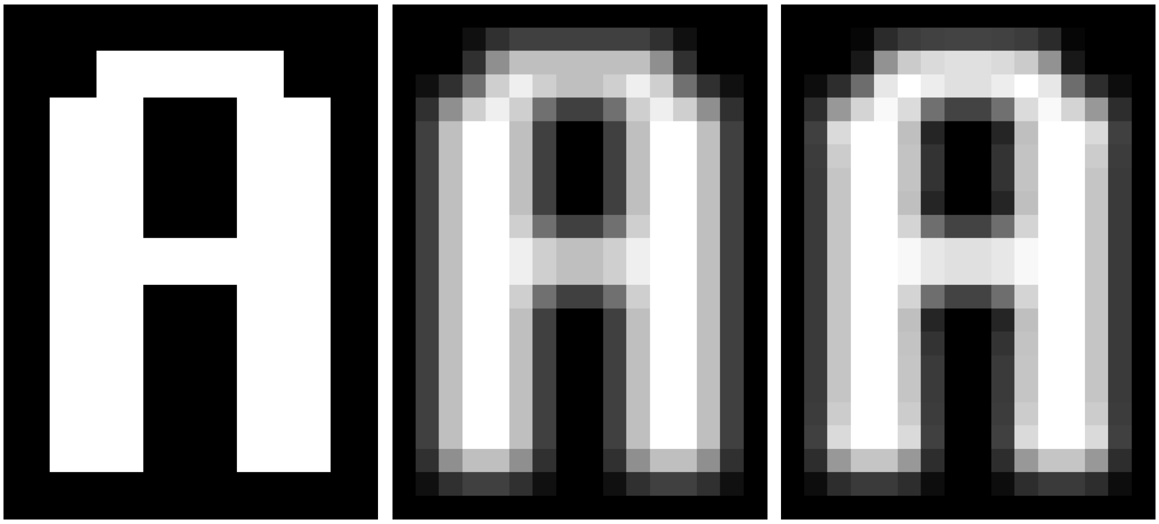

Figure 49: A comparison of three types of resampling for a synthetic image.

Figure 50: A comparison of three types of resampling for a natural image.

The determination of the source position for each output row may also be performed before processing on the image begins. Four tables, containing horizontal and vertical positions and filters, are computed by a utility routine and the main resampling routine is driven by these tables. This allows the routine to be interfaced to different applications with slightly different semantics by writing a new interface routine that computes the positions and the filter coefficients. This approach has been used successfully to provide conformance to both XIL and Adobe Photoshop scaling semantics.

A VIS implementation for the case where the filter width is four is shown in Figures 52-53. The source rows are assumed to be aligned and padded on both ends to allow reading 8 bytes before and 16 bytes after the actual source data. Since they are the result of a prior horizontal resampling pass, this can be guaranteed. A user-supplied channel mask is used to mask output writes along with the usual edge masking. The filter data are presented with 2 integer and 14 fractional bits. When multiplied by the 8-bit pixel data, the result has 10 integral and 6 fractional bits. Accordingly, a `%gsr` shift value of 1 is used, shifting the data right by $6 = 7 - 1$ places.

The performance of the vertical resampling loop is limited by the use of the graphics adder, which executes 12 operations (4 `faligndata` and 8 `fpadd16`). The graphics multiplier executes 10 operations (4 `fmul8x16au`, 4 `fmul8x16al`, and 2 `fpack16`), and the load/store unit performs 4 loads and 1 store. This suggests that expanding the vertical pass to generate

w = width of the vertical filter
c = width of the source and destination images in bytes

For each output scanline:
  Calculate reverse-mapped vertical source position
  Calculate filter coefficients f_i, $0 \leq i < w$ based on subpixel position
  Determine which source scanlines s_i are required, $0 \leq i < w$

  For each byte j of the output, $0 \leq j < c$:
    Compute $\sum_{0 \leq i < w} (f_i \cdot s_{i,j})$ and clamp the result

Figure 51: Pseudocode for vertical resampling.

more than one destination row at a time (reusing some source loads) will not be worthwhile. Suppose we attempt to generate two output rows from the same set of four input rows. We will still execute 4 loads, but now we require 2 stores, 24 FGA, and 20 FGM operations. The loop produces twice as much output but takes twice as long, producing no net benefit. In general we are best off writing simple loops that can be optimized thoroughly rather than trying to use brute force to increase the work per iteration of the loop.

The horizontal pass is complicated by the desire to deal with different input and output formats without excessive code duplication. The various end conditions must also be handled; unlike the vertical case, for which it suffices to substitute pointers, the horizontal case requires either special-casing the processing around the edges or else copying data into an intermediate buffer, since it would be illegal to write replicated values or zeros around the edges – even if the original values are replaced, segmentation violations are possible. Fortunately, an intermediate buffer solves both problems at once.

Consider applying a filter to a span of source pixels. The actual work performed at each byte differs considerably depending on the image formats in question and which bands are to be scaled. In addition, it is unclear how to take advantage of parallelism. In order to circumvent these problems, the source image data is *transposed* bytewise as it is copied into `ibuf`, an array of `vis_d64`s. Figures 54 and 55 show eight rows of 3-banded image data before and after such a transposition.

The input transposition process is illustrated in Figures 56-60. We wish to create a buffer of 8-byte entries, one for each byte along the width of the source image, such that entry i of the buffer holds the data from the i^{th} byte of 8 adjacent source rows. The first 8 entries of the buffer contain data from the first 8 bytes of each row, the second 8 entries contain data from the second 8 bytes, and so forth. Figure 56 shows two source rows being copied into the initial buffer `ibuf`, skipping eight entries after each group of eight bytes. The transposition will take place in 8×8 blocks in order to take advantage of locality and to use VIS instructions. If we were to read 8 bytes from each source row and transpose the resulting block we would have to perform alignment for each source for every block. Reading an arbitrary 8-byte span in isolation requires two loads, an `alignaddr` instruction, and a `faligndata` instruction. Since the `alignaddr` instruction, which sets the alignment field of the `%gsr`, occupies a cycle by itself, it is preferable to avoid performing it with such frequency. Instead, we set the `%gsr` once per source, align it, and copy the 8-byte aligned groups into every eighth entry of the buffer. This requires one `alignaddr` per row, as well as

```
void
resampleV_4 (vis_ras dptr,                /* Pointer to destination row. */
             int width,                   /* Destination width in pixels, */
             int bpp,                     /* Bytes per pixel. */
             vis_d64 *row0, vis_d64 *row1, /* Source rows, aligned & padded. */
             vis_d64 *row2, vis_d64 *row3,
             vis_f32 f01, vis_f32 f23,    /* Filter coefficients. */
             int mask                     /* Channel selection mask. */)
{
  int i, times, offset, align, edge_mask, masklen;
  vis_ras dptr_last;
  vis_d64 sumhi, sumlo, result, half;
  vis_d64 r0, r0a, r0b, r1, r1a, r1b, r2, r2a, r2b, r3, r3a, r3b;
  vis_d64 t0hi, t1hi, t0lo, t1lo, t2hi, t2lo, t3hi, t3lo;

  *((vis_s16 *) &half)     = *((vis_s16 *) &half + 1) =
  *((vis_s16 *) &half + 2) = *((vis_s16 *) &half + 3) = 32; /* For rounding. */

  dptr_last = (vis_ras) ((vis_u8 *) dptr + width*bpp - 1); /* Last byte. */
  edge_mask = vis_edge8(dptr, dptr_last); /*  Left edge mask. */

  offset = ((vis_u32) dptr & 0x7); /* Compute alignment. */
  dptr = (vis_ras) ((vis_u32) dptr & ~0x7); /* Align dptr. */
  align = (8 - offset) & 0x7;
  vis_write_gsr(1 << 3); /* Use 14 fractional bits. */
  (void) vis_alignaddr((vis_ras) 0, align); /* Set %gsr alignment. */

  /* Realign mask to correspond to realigned destination pointer. */
  masklen = bpp*((8 + bpp - 1)/bpp); /* Multiple of 8 greater than bpp. */
  mask = ((mask >> (masklen - align)) & 0xff) | (mask << align);
  mask &= (1 << masklen) - 1;

  /* If dptr is not aligned, we begin reading before the actual source data. */
  if (align != 0) {
    --row0; --row1; --row2; --row3;
  }

  /* Read initial source data. */
  r0a = row0[0]; r0b = row0[1]; r1a = row1[0]; r1b = row1[1];
  r2a = row2[0]; r2b = row2[1]; r3a = row3[0]; r3b = row3[1];

  times = ((vis_u32) dptr_last >> 3) - ((vis_u32) dptr >> 3) + 1;
```

Figure 52: Prologue for vertical resampling with a filter width of 4.

```
for (i = 0; i < times; ++i) {
  /* Read and align input rows. */
  r0 = vis_faligndata(r0a, r0b);
  r0a = r0b; r0b = row0[i + 2];
  r1 = vis_faligndata(r1a, r1b);
  r1a = r1b; r1b = row1[i + 2];
  r2 = vis_faligndata(r2a, r2b);
  r2a = r2b; r2b = row2[i + 2];
  r3 = vis_faligndata(r3a, r3b);
  r3a = r3b; r3b = row3[i + 2];

  /* Compute filter products. */
  t0hi = vis_fmul8x16au(vis_read_hi(r0), f01);
  t1hi = vis_fmul8x16al(vis_read_hi(r1), f01);
  t2hi = vis_fmul8x16au(vis_read_hi(r2), f23);
  t3hi = vis_fmul8x16al(vis_read_hi(r3), f23);
  t0lo = vis_fmul8x16au(vis_read_lo(r0), f01);
  t1lo = vis_fmul8x16al(vis_read_lo(r1), f01);
  t2lo = vis_fmul8x16au(vis_read_lo(r2), f23);
  t3lo = vis_fmul8x16al(vis_read_lo(r3), f23);

  /* Sum up the products, adding a rounding factor. */
  sumhi = vis_fpadd16(half,  t0hi);
  sumhi = vis_fpadd16(sumhi, t1hi);
  sumhi = vis_fpadd16(sumhi, t2hi);
  sumhi = vis_fpadd16(sumhi, t3hi);
  sumlo = vis_fpadd16(half,  t0lo);
  sumlo = vis_fpadd16(sumlo, t1lo);
  sumlo = vis_fpadd16(sumlo, t2lo);
  sumlo = vis_fpadd16(sumlo, t3lo);

  /* Pack and store result using edge mask and channel mask. */
  result = vis_freg_pair(vis_fpack16(sumhi), vis_fpack16(sumlo));
  vis_pst_8(result, (vis_ras) dptr, edge_mask & mask);

  /* Increment dptr, compute new edge mask. */
  dptr = (vis_ras) ((vis_d64 *) dptr + 1);
  edge_mask = vis_edge8(dptr, dptr_last);

  /* Rotate channel mask left 8 positions. */
  mask = ((mask >> (masklen - 8)) & 0xff) | (mask << 8);
}
```

Figure 53: Main loop for vertical resampling with a filter width of 4.

$$
\begin{aligned}
\texttt{row}[0] &= r_{00}g_{00}b_{00}r_{01}g_{01}b_{01}\ldots r_{0(n-1)}g_{0(n-1)}b_{0(n-1)} \\
\texttt{row}[1] &= r_{10}g_{10}b_{10}r_{11}g_{11}b_{11}\ldots r_{1(n-1)}g_{1(n-1)}b_{1(n-1)} \\
&\vdots \\
\texttt{row}[7] &= r_{70}g_{70}b_{70}r_{71}g_{71}b_{71}\ldots r_{7(n-1)}g_{7(n-1)}b_{7(n-1)}
\end{aligned}
$$

Figure 54: The first 8 source rows before transposition into `ibuf`.

$$
\begin{aligned}
\texttt{ibuf}[0] &= r_{00}r_{10}r_{20}\ldots r_{70} \\
\texttt{ibuf}[1] &= g_{00}g_{10}g_{20}\ldots g_{70} \\
\texttt{ibuf}[2] &= b_{00}b_{10}b_{20}\ldots b_{70} \\
\texttt{ibuf}[3] &= r_{01}r_{11}r_{21}\ldots r_{71} \\
\texttt{ibuf}[4] &= g_{01}g_{11}g_{21}\ldots g_{71} \\
\texttt{ibuf}[5] &= b_{01}b_{11}b_{21}\ldots b_{71} \\
&\vdots \\
\texttt{ibuf}[3(n-1)] &= r_{0(n-1)}r_{1(n-1)}r_{2(n-1)}\cdots r_{7(n-1)} \\
\texttt{ibuf}[3(n-1)+1] &= g_{0(n-1)}g_{1(n-1)}g_{2(n-1)}\cdots g_{7(n-1)} \\
\texttt{ibuf}[3(n-1)+2] &= b_{0(n-1)}b_{1(n-1)}b_{2(n-1)}\ldots b_{7(n-1)}
\end{aligned}
$$

Figure 55: Data in `ibuf` after bytewise transposition.

one load and one `faligndata` instruction per 8 source bytes. The function `copy_span_skip` shown in Figure 57 illustrates this process. In addition, this function exhibits the technique of performing one loop iteration fewer than necessary in order to avoid excessive readahead. This is necessary here since the source data come from a user-supplied image whose bounds must not be exceeded in any way. Removing the iteration requires that extra code be added in the case that the number of iterations collapses to zero. Lastly, a final alignment without readahead must be performed following the loop.

The horizontal resampling of the transposed data is shown in Figure 61. The source data enter in the buffer `ibuf` and are sampled according to the table `src_col`. The filter format is the same as for the vertical pass. Since we process a band at a time, it is possible to rearrange the order of the bands without extra cost; the `src_band`, `dst_band`, and `num_bands` arguments specify the mapping between the bands of the two images. Figure 62 illustrates the contents of `mbuf` and `obuf` during this process; each entry of `mbuf` contains a column of values each belonging to the same channel. Four entries are combined using a function $f_i(a, b, c, d)$ representing a weighted average $f_i0a + f_i1b + f_i2c + f_i3d$. The source pixels for destination pixel i are determined by a backwards-mapping function $p(i)$ corresponding to the `src_col` array. The coefficients f_i come from the input arrays `f01s` and `f23s`. Both arrays are computed once per image, as described earlier.

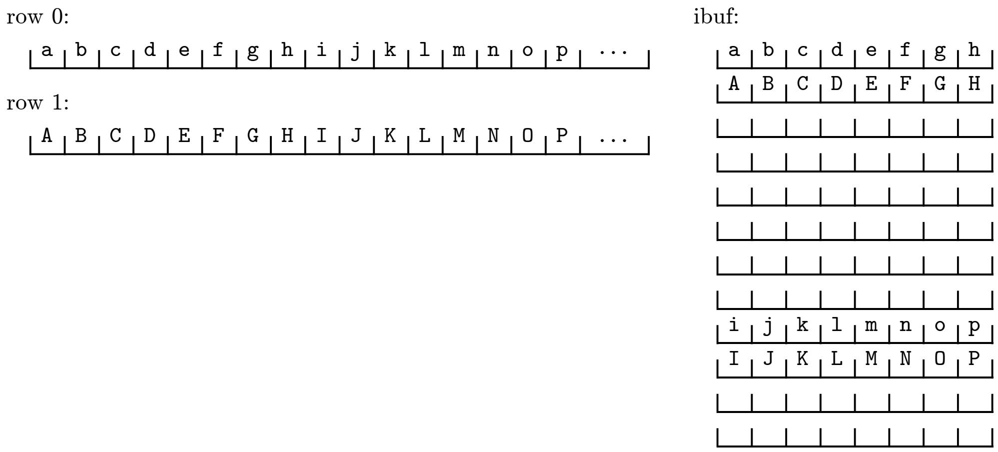

Figure 56: Copying two source rows using `copy_span_skip`.

```
void
copy_span_skip (vis_d64 *dst, vis_ras src, int count, int skip)

{
  int i, doubles = (count + 7)/8;
  vis_d64 s0, s1, *salign;

  salign = vis_alignaddr(src, 0);

  /* Be careful about very short source counts. */
  if (doubles == 1 && salign == 0) {
    *dst = *salign;
    return;
  }

  s0 = *salign; s1 = *++salign;

  for (i = 0; i < doubles - 1; ++i) {
    *dst = vis_faligndata(s0, s1); s0 = s1; s1 = *++salign;
    dst += skip;
  }
  *dst = vis_faligndata(s0, s1); /* Write last double. */
}
```

Figure 57: Copying source data into every eighth entry of a buffer.

$$
\begin{aligned}
p_i &= a_{i0}a_{i1}a_{i2}a_{i3}a_{i4}a_{i5}a_{i6}a_{i7} \\
m_{04} = h(p_0) \odot h(p_4) &= a_{00}a_{40}a_{01}a_{41}a_{02}a_{42}a_{03}a_{43} \\
m_{26} = h(p_2) \odot h(p_6) &= a_{20}a_{60}a_{21}a_{61}a_{22}a_{62}a_{23}a_{63} \\
m_{15} = h(p_1) \odot h(p_5) &= a_{10}a_{50}a_{11}a_{51}a_{12}a_{52}a_{13}a_{53} \\
m_{37} = h(p_3) \odot h(p_7) &= a_{30}a_{70}a_{31}a_{71}a_{32}a_{72}a_{33}a_{73} \\
m_{0426} = h(m_04) \odot h(m_26) &= a_{00}a_{20}a_{40}a_{60}a_{01}a_{21}a_{41}a_{61} \\
m_{1537} = h(m_15) \odot h(m_37) &= a_{10}a_{30}a_{50}a_{70}a_{11}a_{31}a_{51}a_{71} \\
\\
o_0 = h(m_{0426}) \odot h(m_{1537}) &= a_{00}a_{10}a_{20}a_{30}a_{40}a_{50}a_{60}a_{70} \\
o_1 = l(m_{0426}) \odot l(m_{1537}) &= a_{00}a_{11}a_{21}a_{31}a_{41}a_{51}a_{61}a_{71}
\end{aligned}
$$

Figure 58: The process of transposition.

a_0	b_0	c_0	d_0	e_0	f_0	g_0	h_0
a_1	b_1	c_1	d_1	e_1	f_1	g_1	h_1
a_2	b_2	c_2	d_2	e_2	f_2	g_2	h_2
a_3	b_3	c_3	d_3	e_3	f_3	g_3	h_3
a_4	b_4	c_4	d_4	e_4	f_4	g_4	h_4
a_5	b_5	c_5	d_5	e_5	f_5	g_5	h_5
a_6	b_6	c_6	d_6	e_6	f_6	g_6	h_6
a_7	b_7	c_7	d_7	e_7	f_7	g_7	h_7
i_0	j_0	k_0	l_0	m_0	n_0	o_0	p_0
i_1	j_1	k_1	l_1	m_1	n_1	o_1	p_1
i_2	j_2	k_2	l_2	m_2	n_2	o_2	p_2
i_3	j_3	k_3	l_3	m_3	n_3	o_3	p_3
i_4	j_4	k_4	l_4	m_4	n_4	o_4	p_4
i_5	j_5	k_5	l_5	m_5	n_5	o_5	p_5
i_6	j_6	k_6	l_6	m_6	n_6	o_6	p_6
i_7	j_7	k_7	l_7	m_7	n_7	o_7	p_7

$\Rightarrow$

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7
c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7
d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7
e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7
f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
g_0	g_1	g_2	g_3	g_4	g_5	g_6	g_7
h_0	h_1	h_2	h_3	h_4	h_5	h_6	h_7
i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_7
j_0	j_1	j_2	j_3	j_4	j_5	j_6	j_7
k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7
l_0	l_1	l_2	l_3	l_4	l_5	l_6	l_7
m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_7
o_0	o_1	o_2	o_3	o_4	o_5	o_6	o_7
p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7

Figure 59: Transposing data from `ibuf` into `mbuf` by blocks.

```
void
transpose_in (vis_u8 *rh[8], vis_d64 *ibuf, int width, int height)

/* Transpose a portion of a source image up to 8 scanlines tall.
   ibuf must contain space for 8*floor((src->w + 7)/8) vis_d64s. */

{
  vis_d64 p0, p1, p2, p3, p4, p5, p6, p7;
  vis_d64 *buf, m04, m26, m15, m37, m0426, m1537;
  int i, blocks = (width + 7)/8;

  for (i = 0; i < min(height, 8); ++i) /* Copy sources into entries of ibuf. */
   copy_span_skip(ibuf + i, rh[i], width, 8);

  for (i = 0; i < blocks; ++i) { /* Transpose blocks of 8 using 24 fpmerges. */
    p0 = ibuf[8*i];     p1 = ibuf[8*i + 1]; p2 = ibuf[8*i + 2];
    p3 = ibuf[8*i + 3]; p4 = ibuf[8*i + 4]; p5 = ibuf[8*i + 5];
    p6 = ibuf[8*i + 6]; p7 = ibuf[8*i + 7];

    m04 = vis_fpmerge(vis_read_hi(p0), vis_read_hi(p4));
    m26 = vis_fpmerge(vis_read_hi(p2), vis_read_hi(p6));
    m15 = vis_fpmerge(vis_read_hi(p1), vis_read_hi(p5));
    m37 = vis_fpmerge(vis_read_hi(p3), vis_read_hi(p7));
    m0426 = vis_fpmerge(vis_read_hi(m04), vis_read_hi(m26));
    m1537 = vis_fpmerge(vis_read_hi(m15), vis_read_hi(m37));
    ibuf[8*i]     = vis_fpmerge(vis_read_hi(m0426), vis_read_hi(m1537));
    ibuf[8*i + 1] = vis_fpmerge(vis_read_lo(m0426), vis_read_lo(m1537));
    m0426 = vis_fpmerge(vis_read_lo(m04), vis_read_lo(m26));
    m1537 = vis_fpmerge(vis_read_lo(m15), vis_read_lo(m37));
    ibuf[8*i + 2] = vis_fpmerge(vis_read_hi(m0426), vis_read_hi(m1537));
    ibuf[8*i + 3] = vis_fpmerge(vis_read_lo(m0426), vis_read_lo(m1537));

    m04 = vis_fpmerge(vis_read_lo(p0), vis_read_lo(p4));
    m26 = vis_fpmerge(vis_read_lo(p2), vis_read_lo(p6));
    m15 = vis_fpmerge(vis_read_lo(p1), vis_read_lo(p5));
    m37 = vis_fpmerge(vis_read_lo(p3), vis_read_lo(p7));
    m0426 = vis_fpmerge(vis_read_hi(m04), vis_read_hi(m26));
    m1537 = vis_fpmerge(vis_read_hi(m15), vis_read_hi(m37));
    ibuf[8*i + 4] = vis_fpmerge(vis_read_hi(m0426), vis_read_hi(m1537));
    ibuf[8*i + 5] = vis_fpmerge(vis_read_lo(m0426), vis_read_lo(m1537));
    m0426 = vis_fpmerge(vis_read_lo(m04), vis_read_lo(m26));
    m1537 = vis_fpmerge(vis_read_lo(m15), vis_read_lo(m37));
    ibuf[8*i + 6] = vis_fpmerge(vis_read_hi(m0426), vis_read_hi(m1537));
    ibuf[8*i + 7] = vis_fpmerge(vis_read_lo(m0426), vis_read_lo(m1537));
  }
}
```

Figure 60: Transposing the strided input data in blocks of 8.

```
void
resampleH_4 (vis_d64 *ibuf, vis_d64 *obuf,   /* Transposed buffers. */
             vis_f32 f01s[], vis_f32 f23s[], /* Filter coefficients. */
             int src_col[],                  /* Source columns to resample. */
             int src_band[],                 /* Source channels to resample. */
             int dst_band[],                 /* Destination channels. */
             int num_bands,                  /* Number of channels to work on. */
             int sbpp, int dbpp,             /* Bytes per pixel. */
             int width)                      /* Dest width in pixels. */

{
  int col, band, p;
  vis_d64 pix0, pix1, pix2, pix3, acc_hi, acc_lo, half;
  vis_f32 f01, f23;

  vis_write_gsr(1 << 3);
  *((vis_s16 *) &half)     = *((vis_s16 *) &half + 1) =
  *((vis_s16 *) &half + 2) = *((vis_s16 *) &half + 3) = 32;

  for (band = 0; band < numbands; ++band) {
    ibuf += src_band[band]; obuf += dst_band[band];

    for (p = 0; p < width; ++p) {
      col = src_col[p];
      pix0 = ibuf[col];          pix1 = ibuf[col[p] + sbpp];
      pix2 = ibuf[col + 2*sbpp]; pix3 = ibuf[col[p] + 3*sbpp];
      f01 = f01s[p]; f23 = f23s[p];

      acc_hi = vis_fpadd16(half,  vis_fmul8x16au(vis_read_hi(pix0), f01));
      acc_hi = vis_fpadd16(acc_hi, vis_fmul8x16al(vis_read_hi(pix1), f01));
      acc_hi = vis_fpadd16(acc_hi, vis_fmul8x16au(vis_read_hi(pix2), f23));
      acc_hi = vis_fpadd16(acc_hi, vis_fmul8x16al(vis_read_hi(pix3), f23));

      acc_lo = vis_fpadd16(half,   vis_fmul8x16au(vis_read_lo(pix0), f01));
      acc_lo = vis_fpadd16(acc_lo, vis_fmul8x16al(vis_read_lo(pix1), f01));
      acc_lo = vis_fpadd16(acc_lo, vis_fmul8x16au(vis_read_lo(pix2), f23));
      acc_lo = vis_fpadd16(acc_lo, vis_fmul8x16al(vis_read_lo(pix3), f23));

      obuf[p*dbpp] = vis_freg_pair(vis_fpack16(acc_hi), vis_fpack16(acc_lo));
    }

    /* Restore ibuf, obuf pointers to their true positions. */
    ibuf -= src_band[band]; obuf -= dst_band[band];
  }
}
```

Figure 61: Loop for horizontal resampling with a filter width of 4.

$\ldots$		$\ldots$
$r_{p(i),0}\ r_{p(i),1}\ \cdots\ r_{p(i),7}$		$f_i(r_{p(i),0}, r_{p(i)+1,0}, r_{p(i)+2,0}, r_{p(i)+3,0})$
$g_{p(i),0}\ g_{p(i),1}\ \cdots\ g_{p(i),7}$		$f_i(g_{p(i),0}, g_{p(i)+1,0}, g_{p(i)+2,0}, g_{p(i)+3,0})$
$b_{p(i),0}\ b_{p(i),1}\ \cdots\ b_{p(i),7}$	$\Rightarrow$	$f_i(b_{p(i),0}, b_{p(i)+1,0}, b_{p(i)+2,0}, b_{p(i)+3,0})$
$r_{p(i)+1,0}\ r_{p(i)+1,1}\ \cdots\ r_{p(i)+1,7}$		$f_{i+1}(r_{p(i+1),0}, r_{p(i+1)+1,0}, r_{p(i+1)+2,0}, r_{p(i+1)+3,0})$
$g_{p(i)+1,0}\ g_{p(i)+1,1}\ \cdots\ g_{p(i)+1,7}$		$f_{i+1}(g_{p(i+1),0}, g_{p(i+1)+1,0}, g_{p(i+1)+2,0}, g_{p(i+1)+3,0})$
$b_{p(i)+1,0}\ b_{p(i)+1,1}\ \cdots\ b_{p(i)+1,7}$		$f_{i+1}(b_{p(i+1),0}, b_{p(i+1)+1,0}, b_{p(i+1)+2,0}, b_{p(i+1)+3,0})$
$\ldots$		$\ldots$

Figure 62: Resampling channels from `mbuf` into `obuf`.

The horizontal resampling loop exemplifies the principle of eliminating conditionals. Even though we may read the same entry of `ibuf` multiple times, it would be slower to check for this case than to simply reread the data. The resource constraint on this loop is the use of the graphics multiplier, since there are 10 multiplications, as compared with 8 additions and 5 loads and stores. In the case of smaller filter widths, the loads and stores will become more significant. Specialized code for integer bilinear scales, which have predictable patterns of source reads and can be rearranged to reuse the result of a multiplication of a pixel by a given filter coefficient, does produce some increase in performance.

The input image data are acquired into a buffer 8 scanlines at a time. Only rows that are required by the vertical resampling pass are read. The input rows are transposed in place, horizontally resampled into a second buffer, and transposed into a third buffer. This process is performed twice, resulting in two 8-scanline buffers. Vertical resampling is performed using data from these buffers; when the scanlines contained in the first buffer are no longer in use, it is refilled. This buffering scheme increases the locality of memory references for large images, preventing excessive cache misses. It also would provide a natural way to divide computation across several processors since the filling and draining of the buffers is a somewhat asynchronous process.

Now that the data have been copied into every eighth entry of the buffer, the buffer may be transposed in place a block at a time. The function `transpose_in` shown in Figure 60 calls `copy_span_skip` up to 8 times and then transposes the buffer block by block. Part of the transposition process itself is illustrated in Figure 58. The notations $h(x)$ and $l(x)$ are used to denote the high and low portions of a double register, respectively. The remaining six outputs follow a similar pattern. Figure 59 illustrates the effects of this transposition. Note how each entry of `mbuf` comprises bytes from a single channel and column of the source, and how successive entries represent successive bytes.

Performance data are provided in section 8.4.

7.6 Bilinear Scaling by Two

Image resampling libraries typically provide some special-cased code for common resizing cases. Especially important is bilinear scaling by a factor of two, which is commonly used to increase the output size of MPEG or other video streams. Since the resulting pixels are computable as simple combinations of four neighboring pixels, this has the potential to run very quickly. VIS implementations for the case of 1- and 4-banded images have been written. The 1-banded case makes use of the `fpmerge` instruction to transform pixel data of the form $p_0p_1p_2p_3\ldots$ into $p_0p_0p_1p_1p_2p_2\ldots$ and $p_0p_1p_1p_2p_2p_3\ldots$. Adding these together with proper scaling yields a stream of outputs:

$$p_0, \frac{p_0 + p_1}{2}, p_1, \frac{p_1 + p_2}{2}, \ldots$$

Averaging these horizontally resampled scanlines vertically, using a buffering scheme to cache previous values, allows the computation of:

$$\frac{p_0 + p_0'}{2}, \frac{p_0 + p_1 + p_0' + p_1'}{4}, \frac{p_1 + p_1'}{2}, \frac{p_1 + p_2 + p_1' + p_2'}{4}, \ldots$$

where the p' values are taken from the buffer. This process is able to scale a 512×512 1-banded image onto a 1024×1024 window at around 77 frames per second, corresponding to a computation rate of 2.16 clocks per pixel of output. The speed is ultimately limited by the frame buffer pixel write rate. Four cases are needed, depending on the whether the upper-left hand pixel samples the upper-left corner, center, top edge center, or left edge center of a source pixel.

A portion of the C code implementing this approach is shown in Figure 63. In the interest of space we abbreviate `vis_read_hi` and `vis_read_lo` as `v_r_hi` and `v_r_lo`. The constant `f1_4` is equal to $256/4 = 64$ and so has the effect of a multiplication by $1/4$. A comparison with a generic but efficient C implementation is presented in section 8.5.

7.7 Conversion From YUV to RGB Color Space

Conversion from subsampled YUV to RGB color space is frequently the most time consuming portion of a software MPEG decoder. We describe an algorithm that uses a combination of table lookup and partitioned arithmetic to accelerate this process[4]. Color space conversion is sometimes provided as a hardware feature, see for example [Lee95].

In order to reduce the quantity of data to be transmitted and stored, the MPEG standard defines a number of subsampled image formats. In these formats, luminance information (the Y channel) is represented at higher resolution than chroma (U and V channels). This makes efficient use of the available bandwidth since the visual system itself samples these quantities at different resolutions. We will discuss the so-called 4:2:0 case, in which there are four Y samples for each (U, V) sample pair.

The transformation between YUV and RGB color spaces is given by:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.1644 & 0 & 1.5966 \\ 1.1644 & -0.3920 & -0.8132 \\ 1.1644 & 2.0184 & 0 \end{bmatrix} \cdot \begin{bmatrix} Y - 16 \\ U - 128 \\ V - 128 \end{bmatrix}$$

where $Y \in [16, 240]$ and $U, V \in [16, 235]$.

Any linear transformation $y = \mathbf{M} \cdot x$ may be expanded as:

$$y_i = \sum_j \mathbf{M}_{i,j} \cdot x_j$$

This suggests the obvious algorithm in which each element of y is computed as a dot product between x and the appropriate row of $\mathbf{M}$. An alternative approach is to multiply a column

[4]The algorithm described in this section is due to C. Zhou [Zhou95] and has been submitted by Sun for patent protection.

```
soff = ((unsigned long) src) & 0x7; sa = (vis_d64 *) ((unsigned long) src & ~0x7);
prev = sa[1]; vis_alignaddr(0, salign); p0_p7 = vis_faligndata(sa[0], prev);

for (i = 0; i < times; ++i) {
  next = sa[i + 2];    /* Read ahead horizontally, create p8_p15 and p1_p8. */
  vis_alignaddr((void *) 0, soff); p8_p15 = vis_faligndata(prev, next);
  vis_alignaddr((void *) 0, 1);    p1_p8 = vis_faligndata(p0_p7, p8_p15);

  p0p0_p3p3 = vis_fpmerge(v_r_hi(p0_p7), v_r_hi(p0_p7));
  p0p1_p3p4 = vis_fpmerge(v_r_hi(p0_p7), v_r_hi(p1_p8)); /* Merge together  */
  p4p4_p7p7 = vis_fpmerge(v_r_lo(p0_p7), v_r_lo(p0_p7)); /* spans of bytes. */
  p4p5_p7p8 = vis_fpmerge(v_r_lo(p0_p7), v_r_lo(p1_p8));

  p0p0p1p1_2 = vis_fmul8x16al(v_r_hi(p0p0_p3p3), f1_4);
  p0p1p1p2_2 = vis_fmul8x16al(v_r_hi(p0p1_p3p4), f1_4);
  p2p2p3p3_2 = vis_fmul8x16al(v_r_lo(p0p0_p3p3), f1_4);
  p2p3p3p4_2 = vis_fmul8x16al(v_r_lo(p0p1_p3p4), f1_4);    /* Scale by 1/4. */
  p4p4p5p5_2 = vis_fmul8x16al(v_r_hi(p4p4_p7p7), f1_4);
  p4p5p5p6_2 = vis_fmul8x16al(v_r_hi(p4p5_p7p8), f1_4);
  p6p6p7p7_2 = vis_fmul8x16al(v_r_lo(p4p4_p7p7), f1_4);
  p6p7p7p8_2 = vis_fmul8x16al(v_r_lo(p4p5_p7p8), f1_4);

  p0_01_1_12 = vis_fpadd16(p0p0p1p1_2, p0p1p1p2_2);
  p2_23_3_34 = vis_fpadd16(p2p2p3p3_2, p2p3p3p4_2);     /* Add horizontally */
  p4_45_5_56 = vis_fpadd16(p4p4p5p5_2, p4p5p5p6_2);     /* shifted values.  */
  p6_67_7_78 = vis_fpadd16(p6p6p7p7_2, p6p7p7p8_2);

  interp0 = vis_fpadd16(buf0[i], p0_01_1_12);
  interp1 = vis_fpadd16(buf1[i], p2_23_3_34);           /* Compute vertical */
  interp2 = vis_fpadd16(buf2[i], p4_45_5_56);           /* interpolants.    */
  interp3 = vis_fpadd16(buf3[i], p6_67_7_78);

  buf0[i] = p0_01_1_12; buf1[i] = p2_23_3_34;             /* Buffer values  */
  buf2[i] = p4_45_5_56; buf3[i] = p6_67_7_78;             /* for later use. */

  /* Pack and store the first output row. */
  result0 = vis_freg_pair(vis_fpack16(p0_01_1_12), vis_fpack16(p2_23_3_34));
  result1 = vis_freg_pair(vis_fpack16(p4_45_5_56), vis_fpack16(p6_67_7_78));
  *((vis_d64 *) dptr0++) = result0; *((vis_d64 *) dptr0++) = result1;

  p0_01_1_12 = vis_fpadd16(p0_01_1_12, p0_01_1_12);
  p2_23_3_34 = vis_fpadd16(p2_23_3_34, p2_23_3_34);              /* Double. */
  p4_45_5_56 = vis_fpadd16(p4_45_5_56, p4_45_5_56);
  p6_67_7_78 = vis_fpadd16(p6_67_7_78, p6_67_7_78);

  /* Pack and store the second output row. */
  result2 = vis_freg_pair(vis_fpack16(p0_01_1_12), vis_fpack16(p2_23_3_34));
  result3 = vis_freg_pair(vis_fpack16(p4_45_5_56), vis_fpack16(p6_67_7_78));
  *((vis_d64 *) dptr1++) = result2; *((vis_d64 *) dptr1++) = result3;
  prev = next; p0_p7 = p8_p15;                            /* Cycle sources. */
}
```

Figure 63: Bilinear resampling by two.

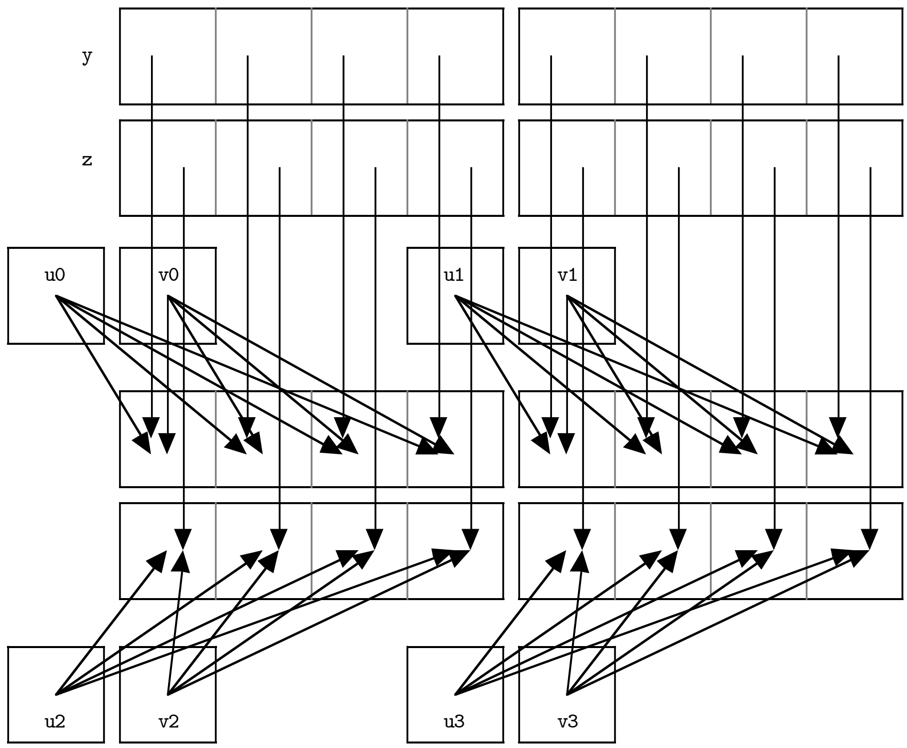

Figure 64: The input data for a single loop iteration.

of $\mathbf{M}$ by a particular element x_i, producing a column vector. The sum of these products is y:

$$y = \sum_j \mathbf{M}_{*,j} \cdot x_j$$

Since the x_i are integers in the small range $[0, 255]$, it is possible to precompute all of these column products for a given matrix $\mathbf{M}$. This table will contain 256 entries for each of Y, U, and V, so it is not excessively large. Each entry consists of three 16-bit signed fixed-point products and so requires 6 bytes of storage. In practice we require 8-byte entries since accesses must be aligned and since VIS will operate on groups of 4 anyway. The frame buffer also requires a padding byte. Adding these entries together and converting the data to 8-bit format requires two `fpadd16` and one `fpack16` instruction.

As with any table lookup approach, the source data must be acquired into integer registers. We read 4 bytes each of U and V information, and 16 bytes of Y that are taken apart using shifts and logical operations. Since the color conversion phase takes its input from a previous decoder phase that has also been ported to VIS, we can arrange for the data to be properly aligned. The input data are illustrated in Figure 64.

Figure 65 illustrates the conversion process. Four bytes of `u` and `v` information are read, along with eight bytes of `y` data (in the variables `y` and `z`). The top two values from `y` and `z` are used as table offsets, along with the first `u` and `v` values. An expression such as `((y >> 13) & 0x7f8)` is equivalent to `((y >> 16) & 0xff) << 3`, i.e., extracting the second byte of `y` and shifting it left 3 places in order to form a proper index into `y_table`. By doing this we save a shift.

```
for (i = 0; i < width; ++i) {
  u = U[i]; v = V[i]; /* Read u, v once for the entire loop iteration. */
  y = yptr[2*i]; z = zptr[2*i]; /* Read first four bytes of y, z. */
  y0_yuv = *((vis_d64 *) (y_table +  (y >> 21)));          /* Use bytes 0-1   */
  y1_yuv = *((vis_d64 *) (y_table + ((y >> 13) & 0x7f8))); /* of y, z, byte 0 */
  z0_yuv = *((vis_d64 *) (y_table +  (z >> 21)));          /* of u, v.        */
  z1_yuv = *((vis_d64 *) (y_table + ((z >> 13) & 0x7f8)));
  u0_yuv = *((vis_d64 *) (u_table +  (u >> 21)));
  v0_yuv = *((vis_d64 *) (v_table +  (v >> 21)));

  uplusv = vis_fpadd16(u0_yuv, v0_yuv);
  sumhi = vis_fpadd16(y0_yuv, uplusv); sumlo = vis_fpadd16(y1_yuv, uplusv);
  vis_std_fpack16(sumhi, sumlo, rgb++);           /* Store two values to rgb. */
  sumhi = vis_fpadd16(z0_yuv, uplusv); sumlo = vis_fpadd16(z1_yuv, uplusv);
  vis_std_fpack16(sumhi, sumlo, rgbnext++);   /* Store two values to rgbnext. */

  /* Do the same using bytes 2-3 of y and z and the byte 2 of u, v. */
  y = yptr[2*i + 1]; z = zptr[2*i + 1]; /* Read four more bytes of y, z. */
  /* Do the same using bytes 0-1 of y and z and the byte 3 of u, v. */
  /* Do the same using bytes 0-1 of y and z and the byte 4 of u, v. */
}
```

Figure 65: The main loop for YUV to RGB color conversion.

Next, the results of the `u` and `v` table lookups are added together, and their sum is added to the four `y` and `z` table lookups to produce four (X, B, G, R) quadruples (the X byte is unused), which are packed into 8-bit format and stored. The inline template **`vis_std_fpack16`** encapsulates the action of packing two values to a register pair and storing them atomically in order to avoid problems with the compiler's handling of complex register pair operations.

The complete loop, processing 16 pixels, requires 2 source reads for U and V, 4 source reads for Y, 24 table reads, and 8 stores, a total of 38 LSU operations. Thus the maximum data rate is 2.375 clocks/pixel. In practice, cache misses and loop pipelining overhead appear to cause the loop to run substantially more slowly. The speedup for this function is difficult to estimate since it has no direct counterpart in XIL and no research has been done to determine the best algorithm for a non-VIS UltraSPARC implementation.

7.8 Gouraud Shading and Texture Mapping

VIS has applications in 3D rendering as well as imaging[5]. Like the i860, VIS may be used to calculate interpolated shading values along a triangle span, an important primitive. The partitioned comparison and partial store operations may also be used to implement a software z-buffer. This potential has not been pursued at SMCC to date, since a much more efficient solution for z-buffering and shading has been implemented in the form of Creator, a low-cost 24-bit frame buffer. This frame buffer uses a new DRAM technology [Deering94] that eliminates the need for a read-modify-write cycle for z comparisons. In the following example follows we assume the use of such a frame buffer, in which a 32-bit color and a 32-bit z (depth) value may be presented to the frame buffer as a single unit. The frame

[5]The code examples in this section were inspired by code written by Grace Wang.

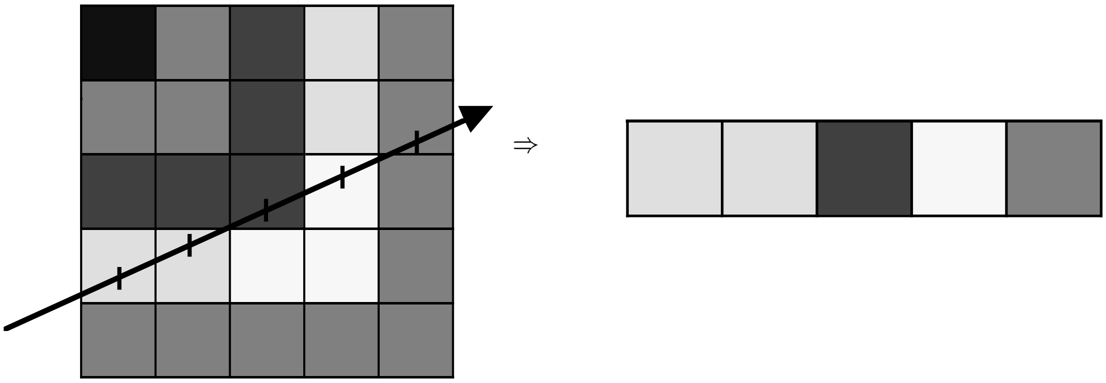

Figure 66: Texture mapping a five-pixel span.

buffer is assumed to reject the write silently if the z value exceeds the previous value at the same screen location.

One area of 3D graphics where VIS has been used is texture mapping. By omitting direct support for texture mapping from the frame buffer, its cost is greatly reduced. Since most users of graphics hardware currently make light use of texture mapping, this appears to be a valid trade-off. As texture mapping becomes more popular this evaluation is of course subject to change.

Texture mapping comes in many forms and we will only deal with a few simple cases. The object is to draw a horizontal span of pixels with values taken from a straight (not necessarily horizontal) line in a texture image. In the general case this amounts to an affine mapping between the source and destination coordinate spaces. The texture samples are taken at points with a fixed spacing $(\Delta x, \Delta y)$ within the texture map. Initially we will simply take the value of the pixel within the texture image (the *texel*) whose center is closest to the sample point (*nearest neighbor* interpolation). This will cause severe aliasing that may be avoided by taking a weighted average of texels near the sample point. We will also ignore edge conditions. An example of acquiring texel values for a span of 5 pixels is shown in Figure 66.

The current position and increment of the sample within the texture map are each represented by a pair of integer variables with a set number `FRAC` of fractional bits. The texture image must have a power-of-two scanline stride; the y position is shifted left and added to the x position (shifted left two places to account for the fact that the texture has 4 bytes per pixel) to yield the texture offset. The z value is also interpolated linearly across the span, and is fed to the frame buffer at each point. Although integer arithmetic could be used to maintain the z value, it is more convenient to maintain it as a true floating point value since we must send it to the frame buffer along with the texture value in one atomic store operation. We define a new inline `vis_fstoi` that allows access to the SPARC `fstoi` instruction to convert the z value from a float to an integer (within a floating point register). Figure 67 shows a version of this function.

A few minor optimizations may be made to this function to reduce the number of shifts. First, `ixpos` may be shifted right by `FRAC - 2` and logically "and"ed with `~0x3` as long as FRAC is greater than 2. A similar optimization may be performed for `iypos`.

Instead of simply copying the texture values onto the screen, we may modulate them

```
void
unlit_texture (float fxpos, float fypos, float fdx, float fdy,
               float fz, float fdz,
               vis_d64 *fb_addr, vis_f32 *texture,
               int y_shift, int count)

{
  int ixpos, iypos, idx, idy, i;
  vis_f32 value;

  ixpos = (int) (fxpos*(1 << FRAC)); iypos = (int) (fypos*(1 << FRAC));
  idx   = (int) (fdx*(1 << FRAC));   idy   = (int) (fdy*(1 << FRAC));

  for (i = 0; i < count; ++i) {
    value = *(texture + ((iypos >> FRAC) << y_shift) + ((ixpos >> FRAC) << 2));
    *fb_addr++ = vis_freg_pair(value, vis_fstoi(fz));
    ixpos += idx; iypos += idy; fz += fdz;
  }
}
```

Figure 67: Unlit texture mapping.

with other linearly interpolated values. It is simple to implement Gouraud (linearly interpolated) shading [Foley90]; a `vis_d64` is maintained to represent a color in 16-bit fixed-point format. This color is initialized at the top vertex, and interpolated according to a delta along the triangle edges. A horizontal color delta is also computed. These deltas correspond to the partial derivatives of the plane connecting the three vertex colors in RGB space and so are constant over each triangle. This is useful since the divisions required to obtain them will be relatively expensive. Within a span, an `fpadd16` and an `fpack16` suffice for updating the value. Alternatively, the 64-bit color may be used as a coefficient modulating the texture value by means of the `fmul8x16` instruction. Finally, a specular component with the color of the light source may be interpolated and added to the modulated texture value prior to packing (although not physically correct, the use of a pure light-source colored highlight is a common approximation in computer graphics).

For higher-quality texturing, bilinear and trilinear interpolation are frequently used. Bilinear interpolation involves reading four texture values surrounding the sample point and averaging them according to the point's horizontal and vertical subtexel coordinates. The easiest way to generate the necessary coefficients, as in the case of resampling (section 7.5), is to index a table of precomputed width 2 filters. The lower-order bits of the texture coordinate are used as the index. The filters are best stored as complementary pairs of coefficients $(f, 1-f)$ which may be accessed using the `fmul8x16au` and `fmul8x16al` instructions. Two multiplies and two additions result in two horizontally interpolated texture values, which must then be averaged according to the vertical filter. Either two 16×16-bit multiplications (costing two multiply instructions each) may be used, or else the horizontal interpolates may be packed and multiplied using 8×16 multiplication instructions. The latter approach uses two fewer additions since the results of the multiplications do not need to be added together.

The main instruction requirements for unlit texture mapping with bilinear interpolation (ignoring the z interpolation) are thus: 4 loads to acquire the texels and 2 loads to acquire the coefficients; 4 multiplications, 2 additions, and 2 packs for horizontal interpolation; 2 multiplications, 1 addition, and 1 pack for vertical interpolation; and a store to the frame buffer. The total is 7 LSU, 9 FGM, and 3 FGA instructions. At 167 MHz, this corresponds to peak performance of 18.5 million 4-element bilinear interpolations per second.

Trilinear interpolation, which introduces an interpolation between different levels in a hierarchy of subsampled versions of the texture (a *MIP-map*), takes 9 FGM operations for each of the two levels, plus an additional 3 to combine them (the decision of which MIP-map levels to use is decided once per span). Thus the peak performance of trilinear interpolation is determined by the need for 21 FGM operations, yielding 7.9 million 4-element interpolations per second.

If it is possible for the interpolated source texture coordinates to exceed the bounds of the rectangular texture map itself, some sort of clipping must be performed. Clipping is probably best done outside the main texture mapping loop in order to avoid the need for conditionals (or conditional moves, lookups, or any other alternative implementation). If the source positions of the endpoints of the span lie within the texture, the interpolated values must as well, so it is safe to proceed after clipping the endpoints without further checks. The need to clip may sometimes be avoided entirely if the entire backwards-mapped figure can be shown not to cross a texture boundary, e.g., if the vertices of a polygon all map within the texture. In any case clipping need not cause any slowdown of the inner loop.

8 Timing Comparisons

The above examples (except for $4:2:0$ color conversion and texture mapping) perform functions that are available as part of SunSoft's XIL imaging and video library. XIL provides both generic C implementations of its functions, as well as hooks to allow them to be reimplemented on a specific device. The generic code (known as the "memory" driver since it manipulates image data in main memory as opposed to a specialized device; strictly speaking, the XIL VIS driver is a memory driver as well) will serve as our base reference. Although this is an imperfect benchmark, it does represent the performance a typical customer would have seen on the current generation of Sun hardware were VIS not implemented. Note that this code is generally written for portability and genericity across image formats, and is additionally not scheduled for UltraSPARC; it is certainly possible to write more efficient C code in the context of an application with fixed image formats and target processor.

Several other potential code bases are available to us for comparison. The XIL memory code running on an UltraSPARC processor represents real-world library performance, but is not highly optimized. The VIS XIL port is substantially faster, but for historical reasons is not always fully optimized according to the principles outlined in this report. We may also wish to time generic code which attempts to make optimal use of the non-VIS portion of the processor (e.g., through use of the modulo scheduler). Finally, we will time a custom VIS routine based on the code fragments shown in section 7 when its performance meaningfully exceeds that of the XIL implementation. We make no attempt to draw comparisons with any non-UltraSPARC platforms.

All the timings were performed on a Sun Ultra 1 workstation with a 167 MHz processor and 96 megabytes of RAM. A simple test harness function was constructed that times the operation of interest on a pair (or triple) of images of a given size. The images were initialized to uniformly-distributed random values – no attempt was made to simulate "real images." The sizes were varied between 100 and 1000 pixels in both the horizontal and vertical directions in order to observe any size-dependent effects such as loop overhead, cache aliasing, and so forth. The test program was then run, calling each implementation in turn, producing a measurement of megapixels processed per second. The resulting data were fitted with an interpolating cubic spline, and plotted with up to five distinguished contour levels (depending on readability) by the freely-distributed `GNUPLOT` plotting package.

The same compiler, SPARCWorks SPARCCompiler 4.0 was used throughout, with the options `-fast -O4 -xchip=ultra -xarch=v8plusa`. These options produce code optimized using the compiler's UltraSPARC machine model, and allow non-SPARC v9 instructions, including VIS, in the generated code.

8.1 Addition and Alpha Blending

The three-dimensional graphs produced by the XIL memory, XIL VIS, and custom VIS implementations of addition share a common trait – at least within the domain of images smaller than 1000×1000, the total image size is the crucial factor in determining performance and not the exact image dimensions. This can be seen by observing the shape of the isocontours, which appear to follow curves of constant product, i.e., $y = k/x$. In particular, the code produced by the modulo scheduler does not appear to be excessively dependent on high trip counts.

The XIL memory and XIL VIS addition performance graphs (Figures 68-69) show a strong oscillation. This is a result of two factors. First, they are not modulo scheduled; loads and uses are not always separated enough to mitigate the effects of cache misses. As the alignment of source lines in memory changes, these cache misses will occur at different load instructions, some of which will result in longer stalls than others. In the case of the memory routine, each loop iteration processes a single pixel; at the top edge of the graph, each iteration requires approximately $167/13 \approx 12.8$ cycles; at the bottom edge, roughly $167/9 \approx 18.5$ cycles are used. Consider the first two pronounced troughs, comprising image dimensions of 129-157 and 182-203 pixels. In both cases, the added quantity of source data is around 16K (since $2 \cdot (157^2 - 129^2) = 16016$ and $2 \cdot (203^2 - 182^2) = 16170$), and this pattern continues, suggesting the 16K internal cache as the source of the oscillation.

The non-modulo scheduled VIS code (Figure 69) follows the same oscillating pattern; in addition we see the effects of overhead (pointer arithmetic, edge masking, etc.) for small images in the form of a descending "tail" in the left corner of the plot.

The second, more significant factor causing oscillation is XIL's policy of allocating image memory starting at page boundaries, increasing cache line aliasing. A second set of experiments was performed, in which each image was allocated at a random offset within a page. This had the effect of reducing contention for cache lines. Figures 70 and 71 show the results of this change. The plots display some random variation, but overall show none of the oscillation seen previously.

The modulo-scheduled VIS code (Figure 72) displays a much greater insensitivity to the precise cache dimensions, even though no special steps were taken to randomize the pointer offsets or scanline strides. For very small images (under 200 pixels on a side), the low loop trip counts cause some performance fluctuation. The next major change in performance occurs slightly above 400 pixels; the external 512kb cache is theoretically capable of just holding three such images, since $\sqrt{(512 \cdot 1024)/3} \approx 418$. Still, even the lowered performance exceeds the peak performance of the non-modulo scheduled loop. At the peak performance of around 130 megapixels/second, each 8-pixel loop iteration takes approximately 10.3 cycles; at the sustained, large image performance of around 60 megapixels/second, 22.3 cycles are consumed, a difference of 12 cycles. This suggests that the external cache is being missed (with a 22 cycle penalty, since 8 of the 30 cycles are covered by the modulo scheduler) every two iterations or so. This is reasonable, since a cache miss that brings in 32 bytes of data from one source image during one iteration guarantees that the next three iterations will find their data in cache. Since there are two source images, on average one half of the iterations will feature a cache miss.

The timings of all five methods are compared directly by taking diagonal cross-sections of the previous plots, in effect considering square images only. The results are displayed in Figure 73.

The alpha blending timings (Figures 74-77) tell a similar story to those for addition. The relevant cutoff point for filling the external cache is now $\sqrt{(512 \cdot 1024)/4} \approx 362$. The same 16K cyclic behavior is present; the 8-pixel cycle count at the small image performance of roughly 75 megapixels/second is 17.8 cycles; the large image performance of roughly 40 megapixels/second represents a 33.4 cycle loop iteration. Assuming that each source image faults around every fourth iteration, we would expect to spend $22/4 = 5.5$ additional cycles for each of the three images, or a total of 16.5 cycles per iteration, which is not far from the "eyeballed" difference of 15.6. The same trick of randomizing the data pointers would work here as well; we preserve the oscillation in order to show its universality.

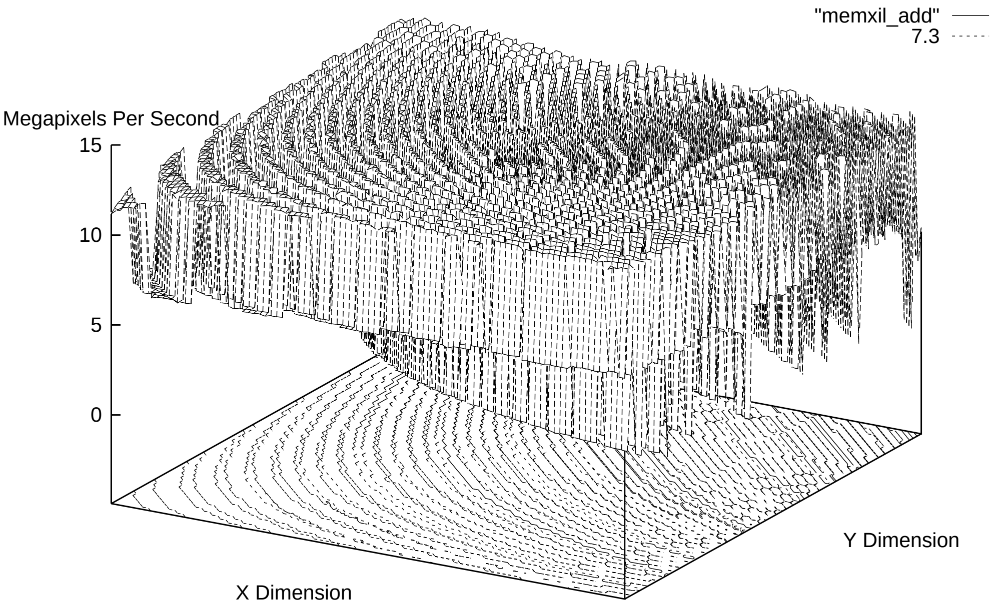

Figure 68: Times for the XIL memory implementation of addition.

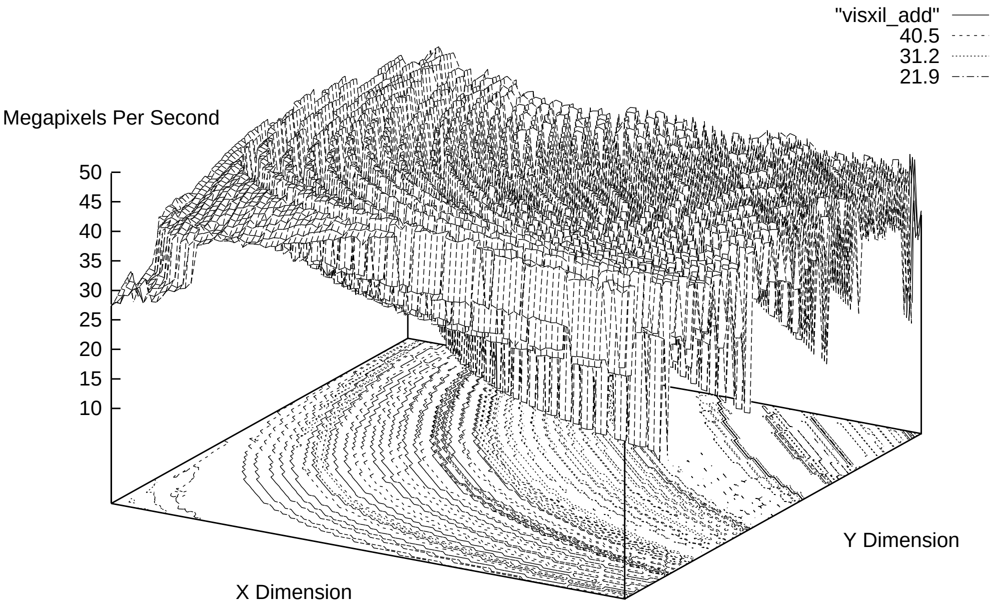

Figure 69: Times for the XIL VIS implementation of addition.

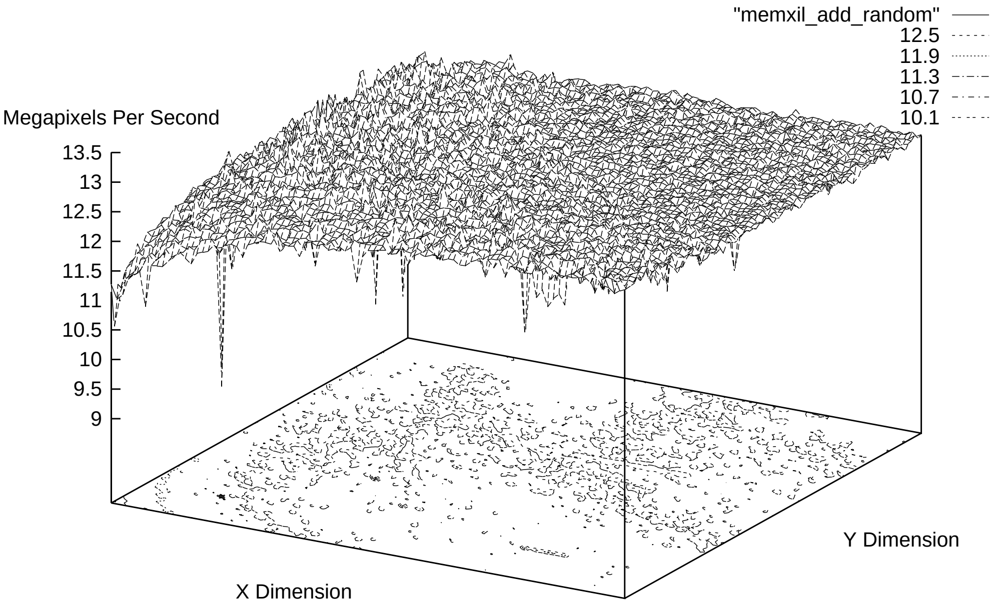

Figure 70: Times for the (randomized) XIL memory implementation of addition.

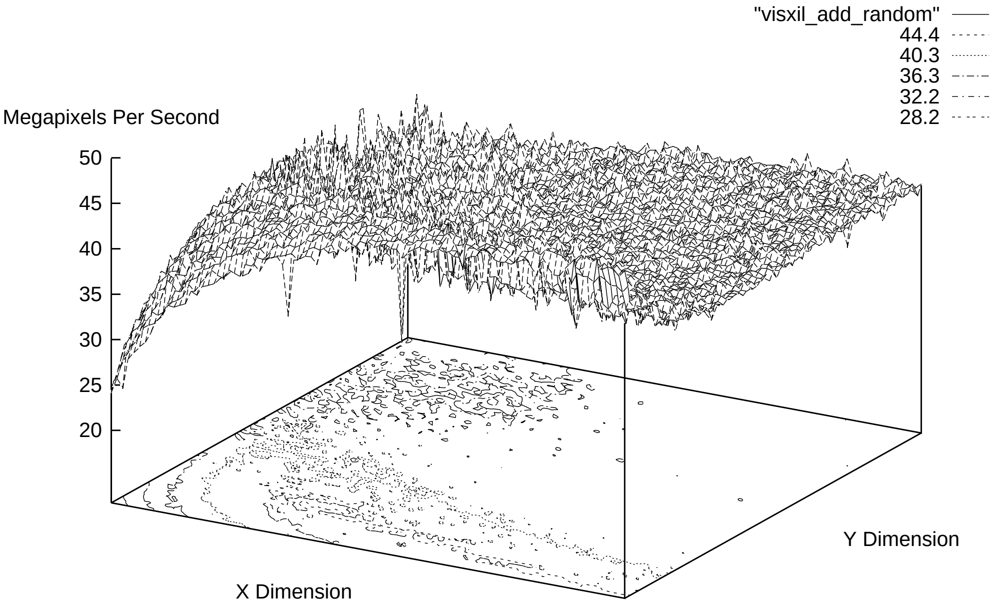

Figure 71: Times for the (randomized) XIL VIS implementation of addition.

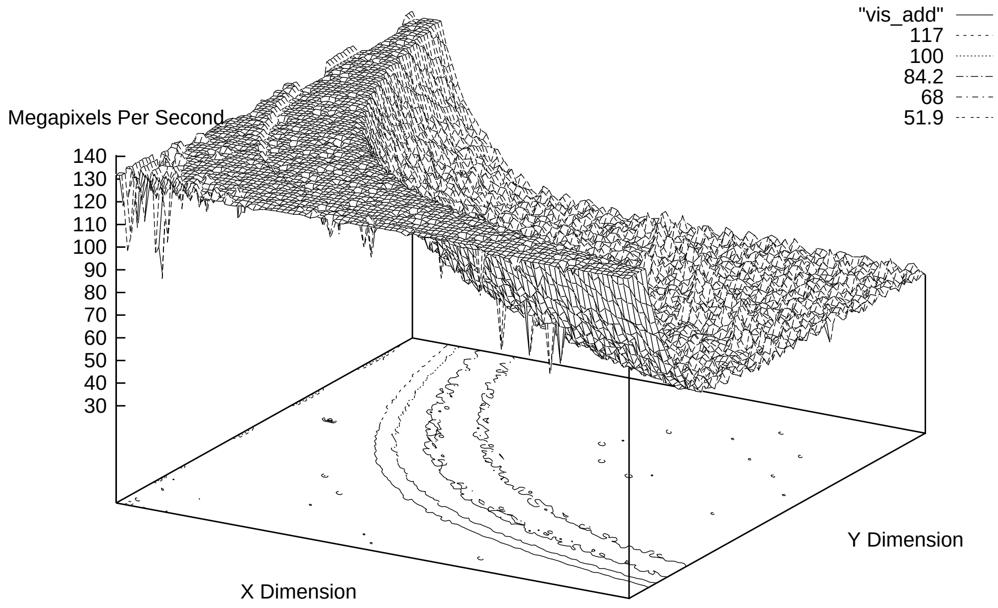

Figure 72: Times for the (non-XIL) VIS implementation of addition.

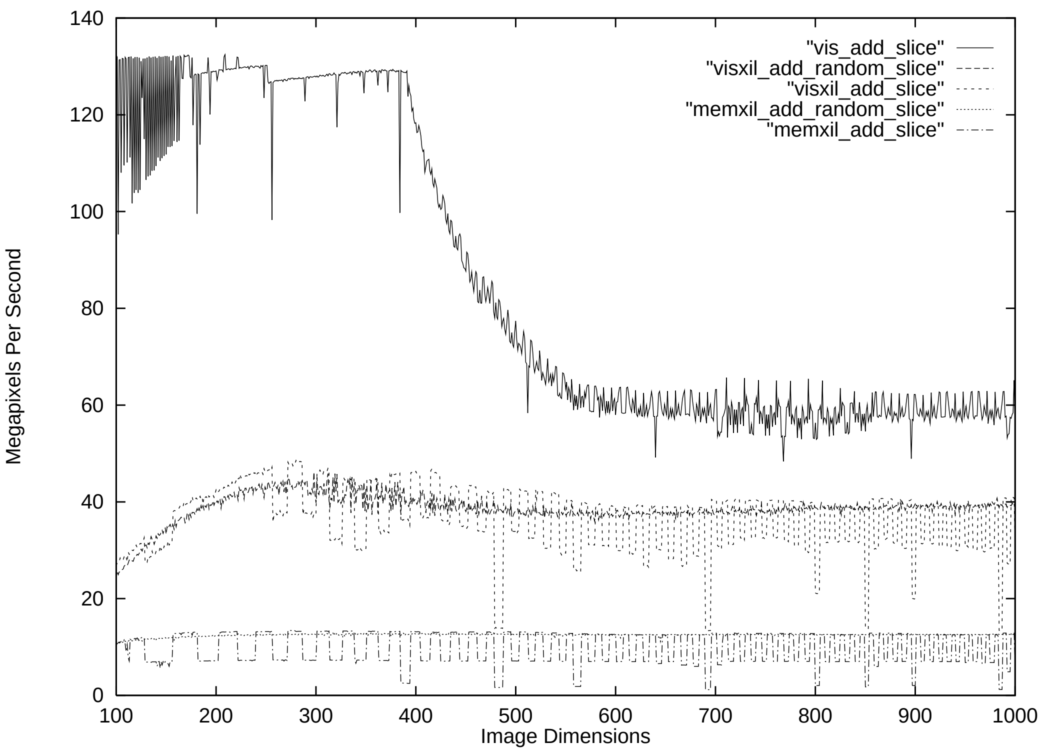

Figure 73: Comparison of five implementations of addition (Figures 68-72).

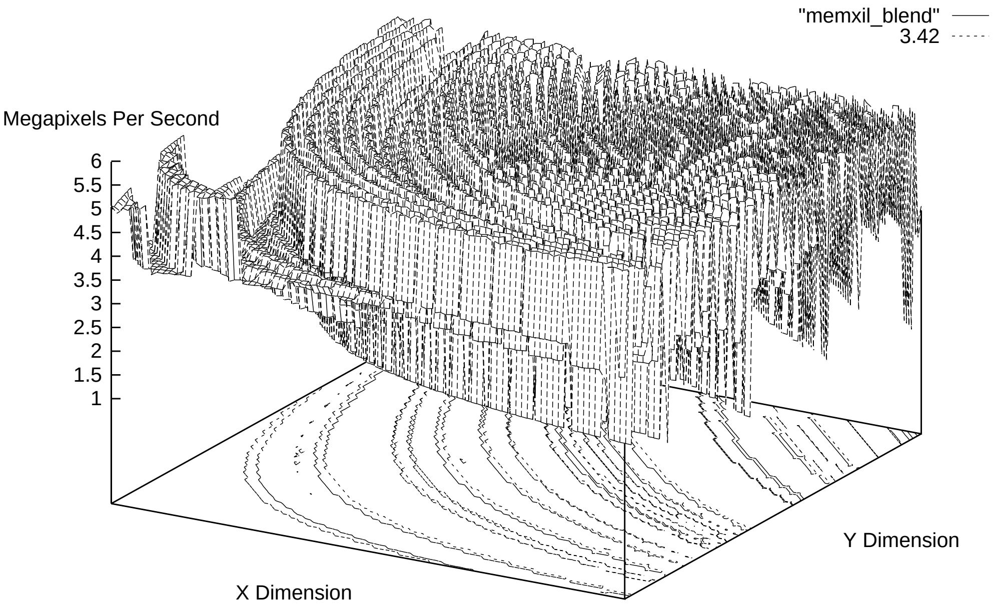

Figure 74: Times for the XIL memory implementation of alpha blending.

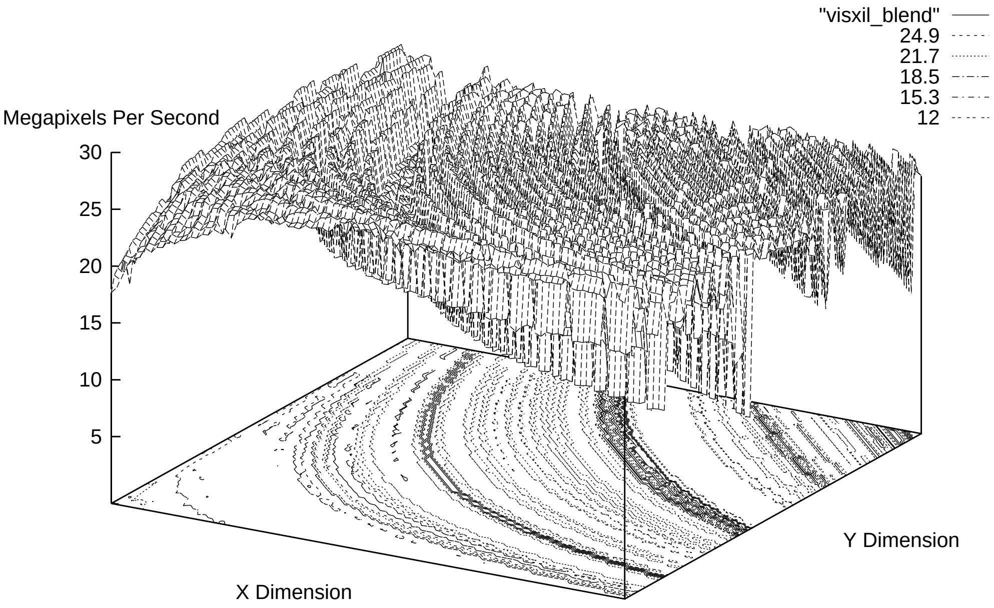

Figure 75: Times for the XIL VIS implementation of alpha blending.

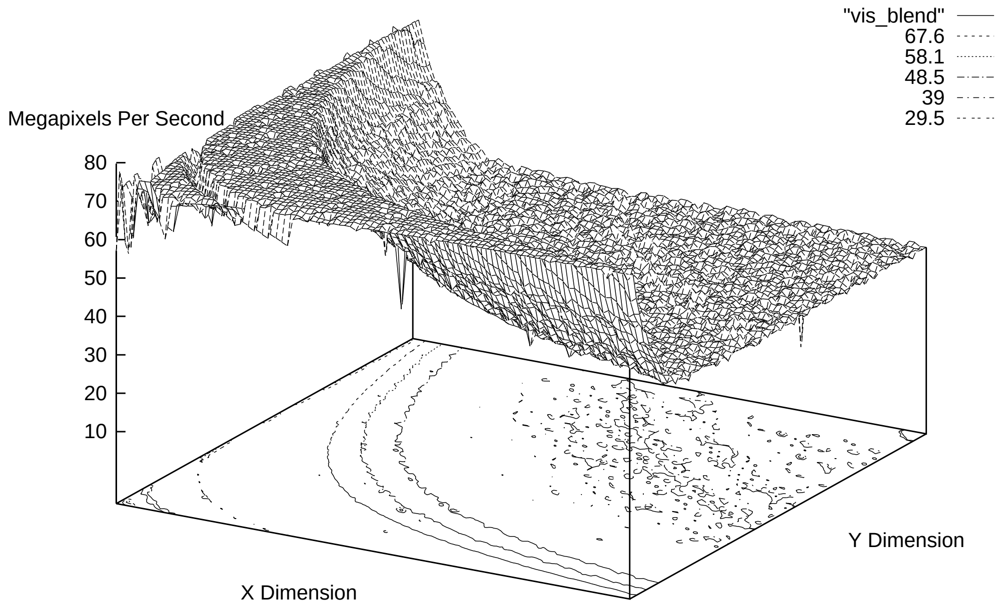

Figure 76: Times for the VIS implementation of alpha blending.

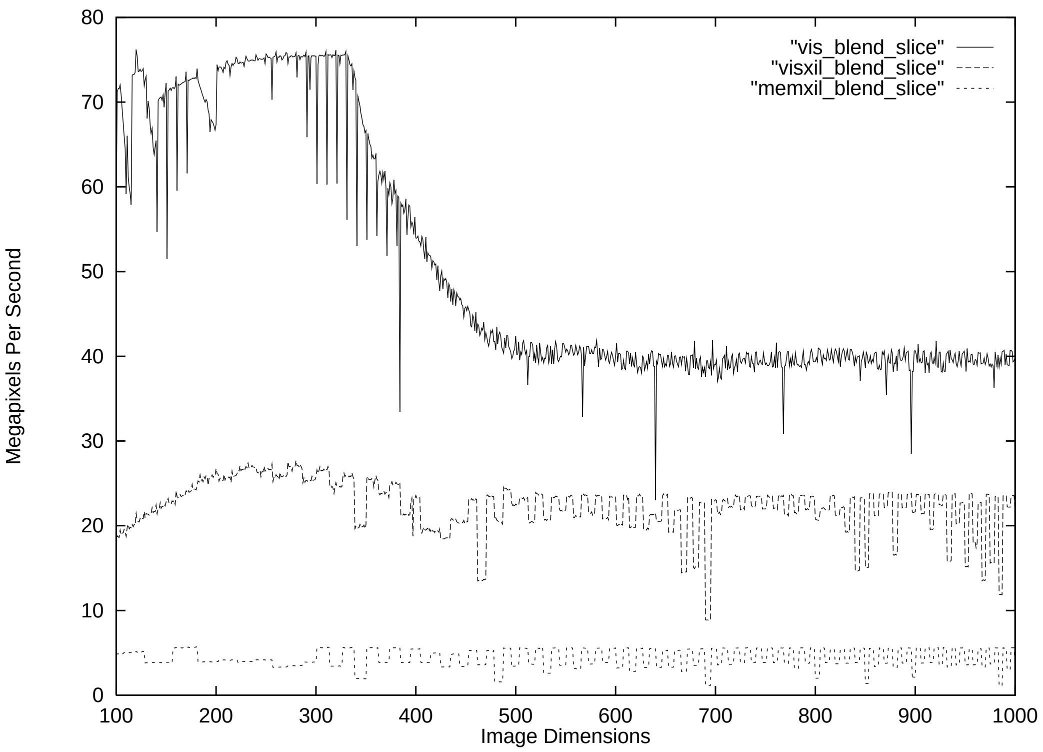

Figure 77: Comparison of three implementations of alpha blending (Figures 74-76).

8.2 16- to 8-Bit Table Lookup

The lookup function is sensitive to the lookup table size. Three sets of data were collected, in which the source values were restricted to $[-512, 511]$, $[-8192, 8191]$, and $[-32767, 32767]$ (the full range). The largest and smallest ranges are displayed in the three-dimensional plots, while the intermediate range may be seen in the cross-sectional plots. The random source data form a worst case for table lookup; real images will exhibit locality.

The first experiment, showing the performance of the XIL memory implementation is shown in Figures 78-79. This implementation appears to be mainly instruction-bound, due to its general nature and lack of processor-specific scheduling. For very small images, loop overhead slows its performance noticeably; or larger images, it performs at a nearly constant rate. This rate is, however, dependent on the lookup table size. The cache miss cycles occur with sufficient uniformity to produce a smooth performance decrease.

The second two experiments, shown in Figures 80-81 and Figures 82-83, are of identical C code for the specific case of lookup at hand, scheduled for UltraSPARC. The code is essentially the same as the initial example of section 7.3, which reads and writes a pixel at a time. The second set of experiments were compiled with the additional flag:

```
-Qoption cg -ms_pipe,-Qms_pipe+non_float_loop_ld=8
```

This flag informs the modulo scheduler to use a delay of 8 between loads and their uses even for integer loops; integer loads would otherwise be assigned a delay of 2 cycles by default.

The fourth experiment uses the VIS code developed in section 7.3. We see from the contours of Figures 84-85 that the times are not quite symmetric with respect to the x and y dimensions of the images; on the left side of the plots, representing "thin" images, performance drops off somewhat. This is due to the presence of low trip counts, the effects of which are made pronounced by the routine's large dependence on the high-latency LSU. In other words, the loops perform a great deal of loads but little computation, so the initial wait for data during the prologue has a noticeable effect on the loop timing as a whole.

In general the modulo-scheduled codes, VIS or non-VIS, display the same characteristic "bowl" shape seen in the previous examples. For images that fit in cache, near-theoretical performance is achieved, while for larger images the cache misses add a constant number of cycles to each group of iterations. The presence of a lookup table adds an additional factor, in that arbitrary cache lines may need to brought in at any time, possibly overwriting image data that will be needed in the future and vice versa. Only very small images can totally avoid overwriting entries of the 64K table, as seen in the initial high-performing "necks" of Figures 81, 83, and to a lesser extent, Figure 85.

The cross-sections of the four experiments are shown together, in Figures 86-89. The differences between the performance for table sizes of 1K and 64K must be due to the contention for cache lines between image and table data. The 1K table uses a negligible fraction of the external cache, while the 64K table uses one 1/8 of it. Assume, therefore, that a given source read has a 1/8 chance of missing due to a table entry occupying the same line as a source pixel. In the case of the memory code with a load delay of 8, this would translate into $(30 - 8)/8 = 2.75$ cache miss cycles per source read. However, we observe a sustained performance loss of $167/31 - 167/38 \approx 1$ cycle per iteration between the flat portions of the top and bottom curves in Figure 88. Accounting for this factor of 2.75 would require a better model of the interactions between image and table data. When the table size is increased, the VIS performance drops from around $167 * 8/55 \approx 24$ cycles to $167 * 8/44 \approx 30$ cycles per iteration for images that do not fit in cache. This suggests an extra cache miss every $(30 - 8)/6 \approx 3.66$ iterations.

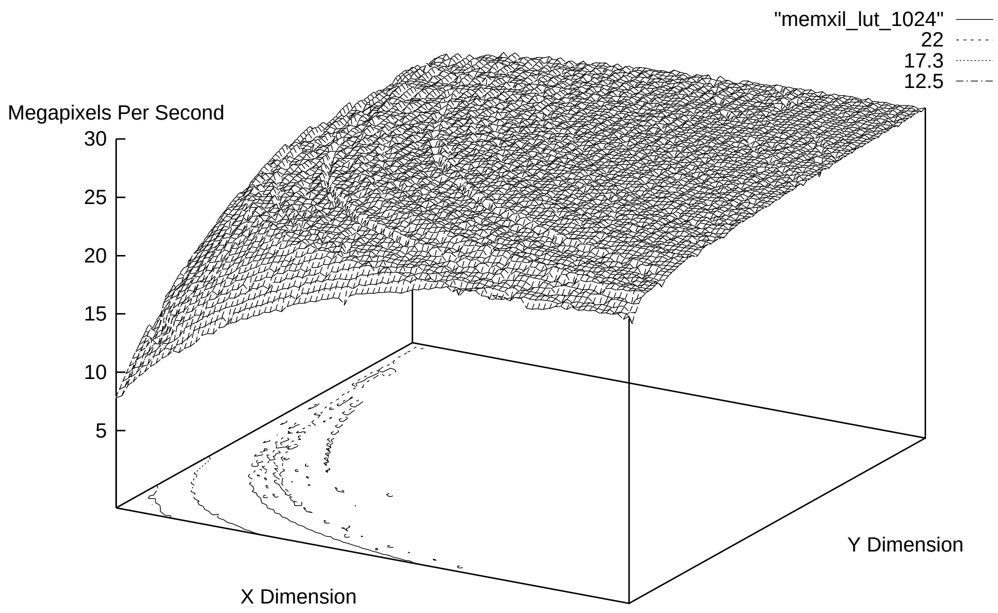

Figure 78: Times for the XIL memory implementation of lookup, range=$[-512, 511]$.

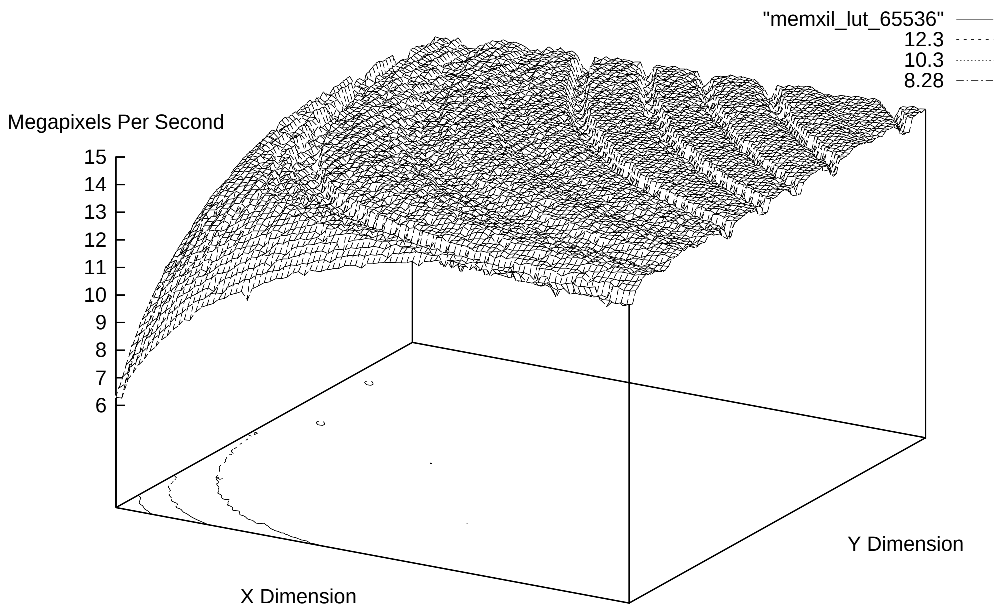

Figure 79: Times for the XIL memory implementation of lookup, range=$[-32768, 32767]$.

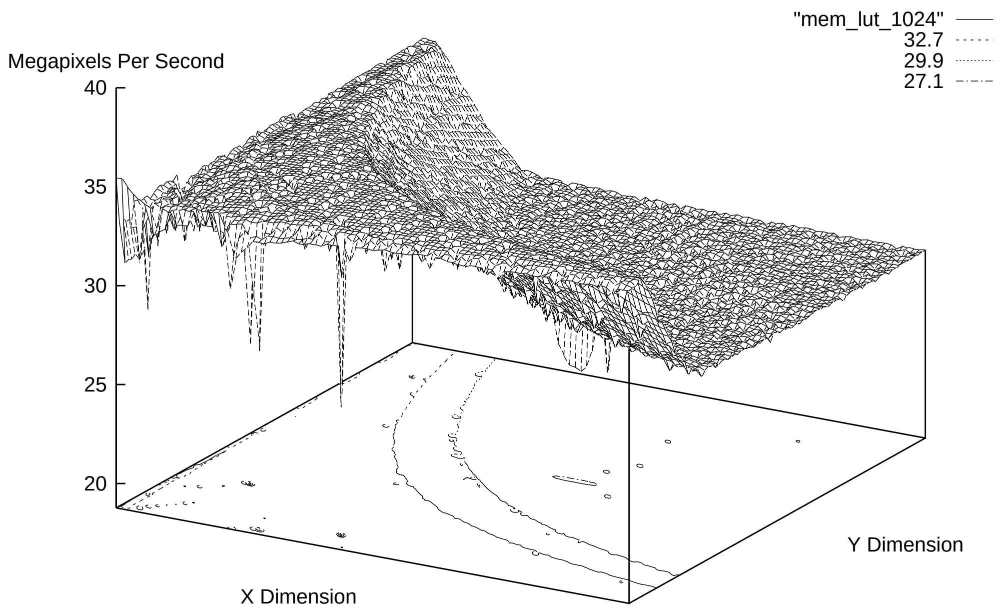

Figure 80: Times for the memory (2) implementation of lookup, range=$[-512, 511]$.

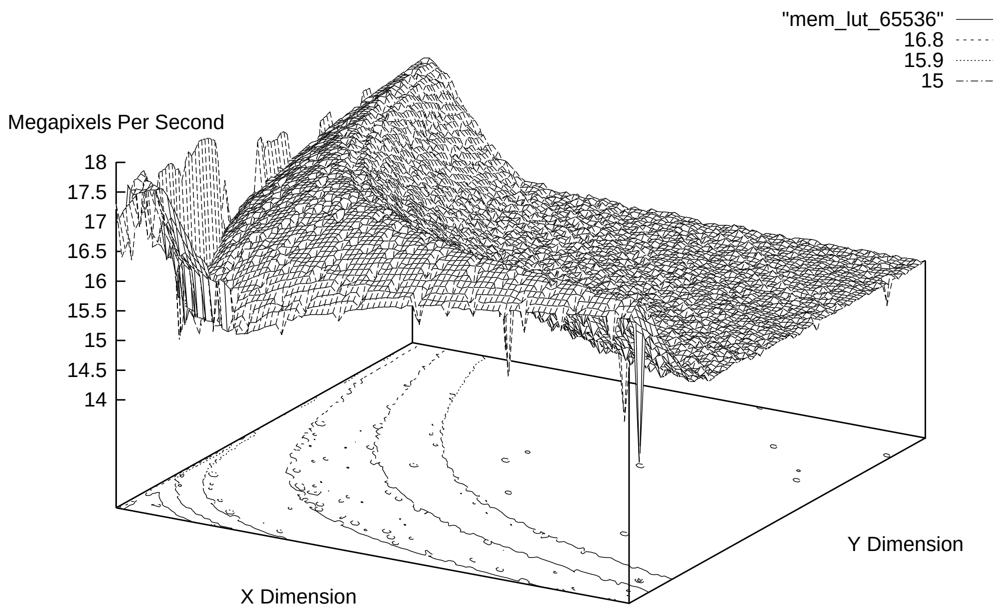

Figure 81: Times for the memory (2) implementation of lookup, range=$[-32768, 32767]$.

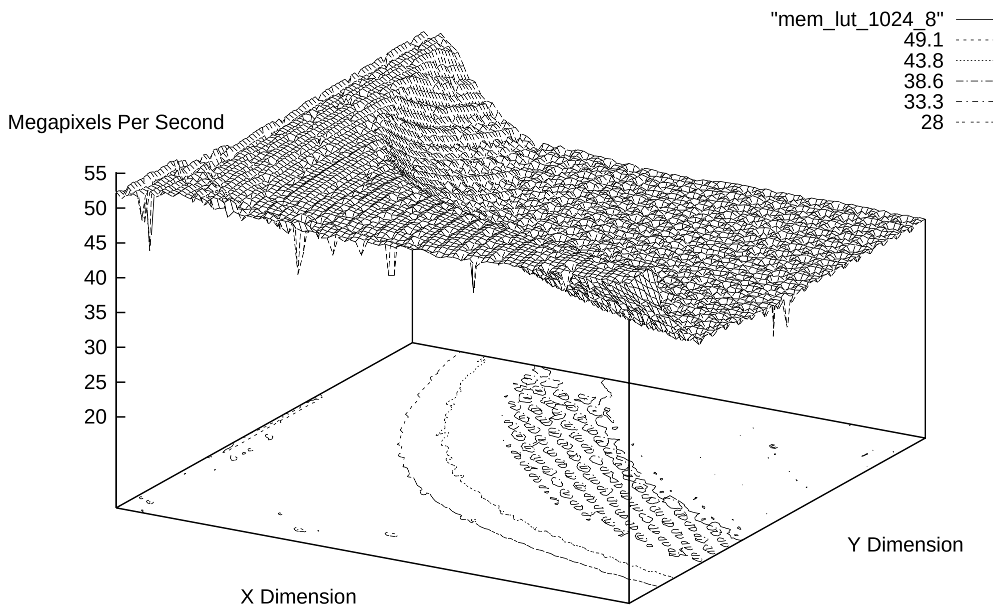

Figure 82: Times for the memory (8) implementation of lookup, range=$[-512, 511]$.

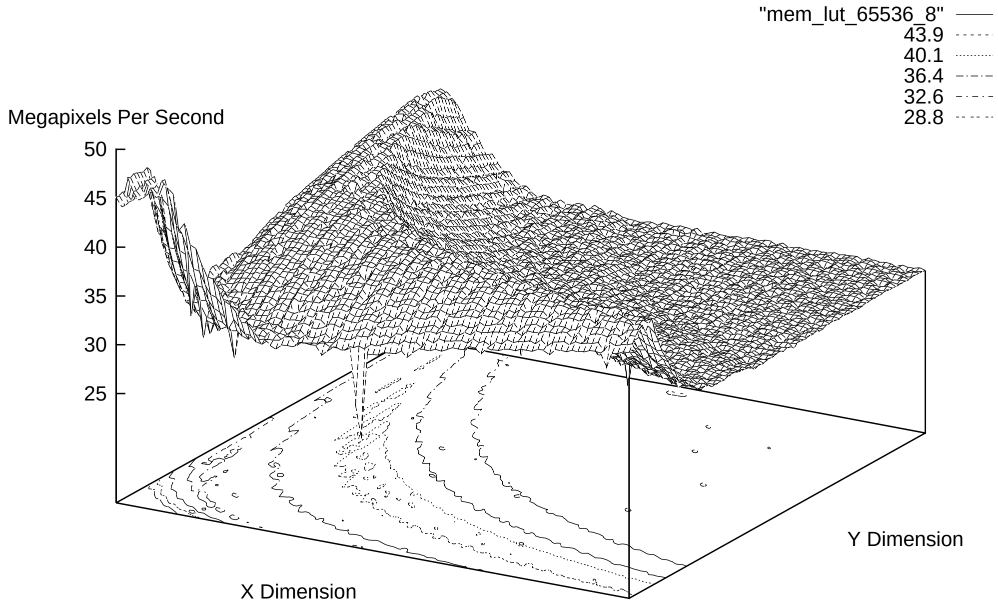

Figure 83: Times for the memory (8) implementation of lookup, range=$[-32768, 32767]$.

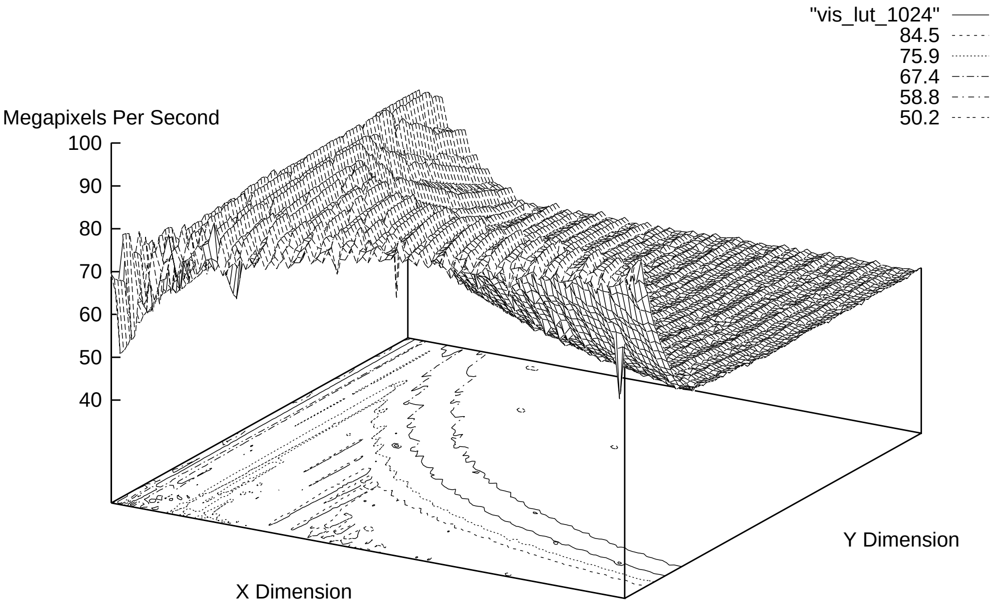

Figure 84: Times for the VIS implementation of lookup, range=$[-512, 511]$.

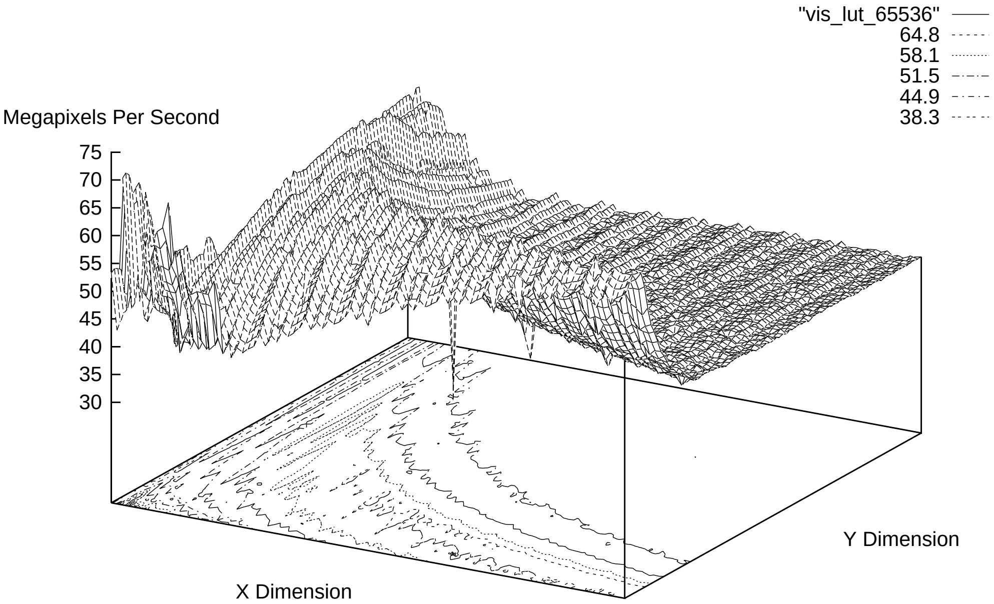

Figure 85: Times for the VIS implementation of lookup, range=$[-32768, 32767]$.

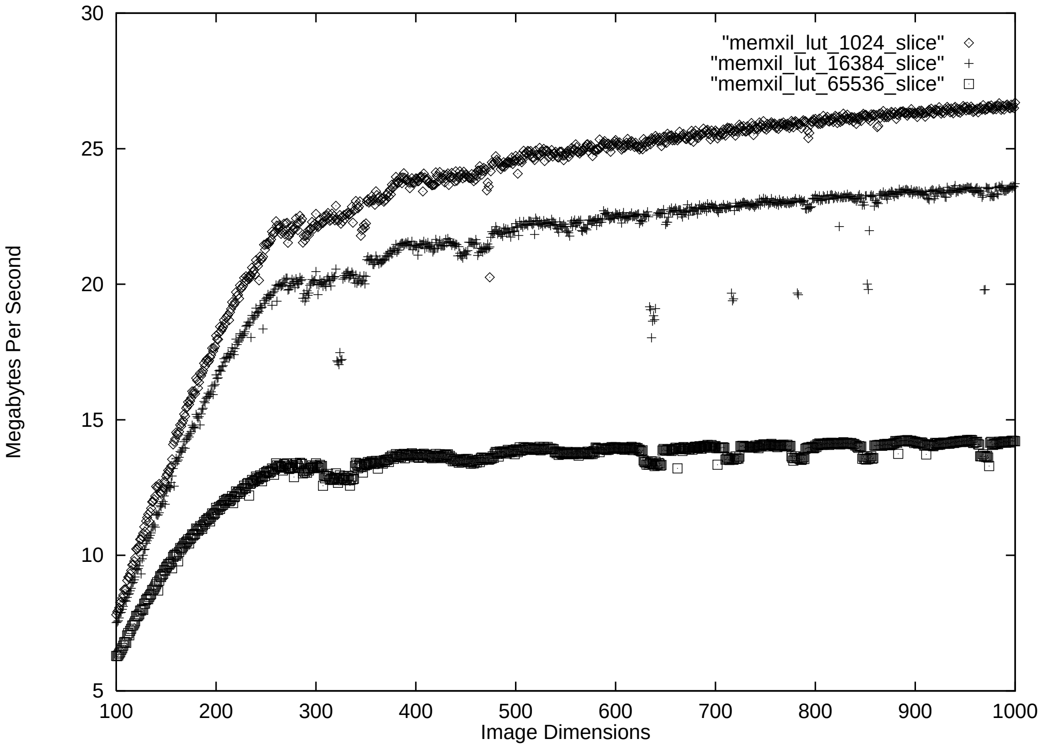

Figure 86: Relative speeds of XIL memory lookup for varying table sizes (Figures 78-79).

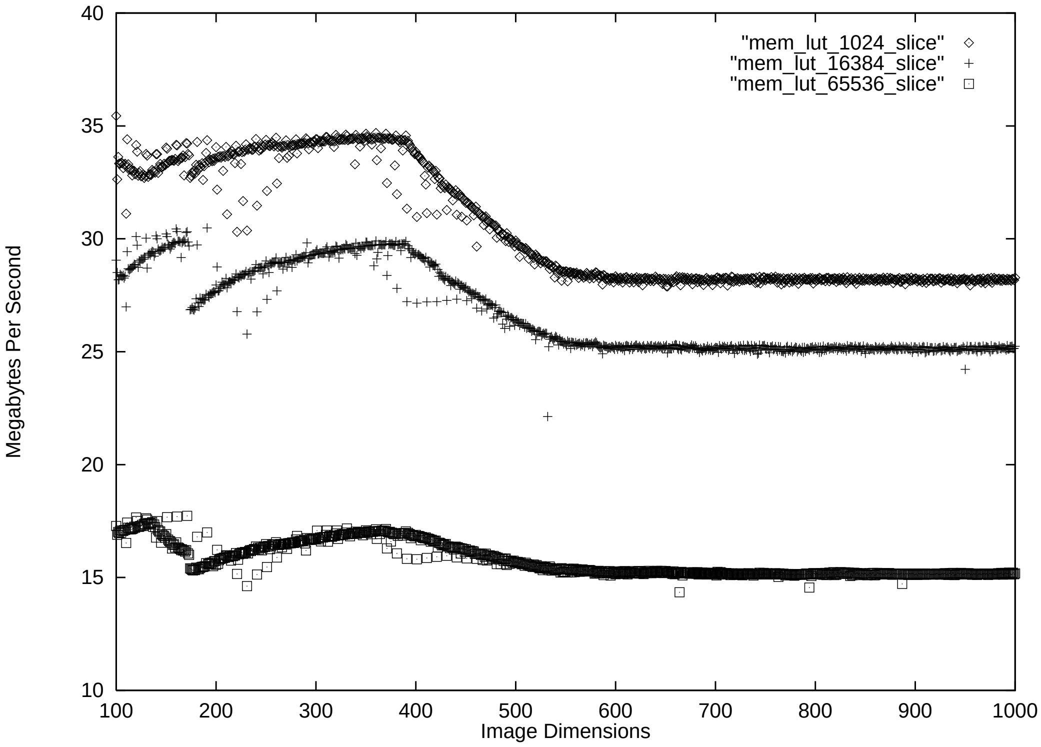

Figure 87: Relative speeds of memory (2) lookup for varying table sizes (Figures 80-81).

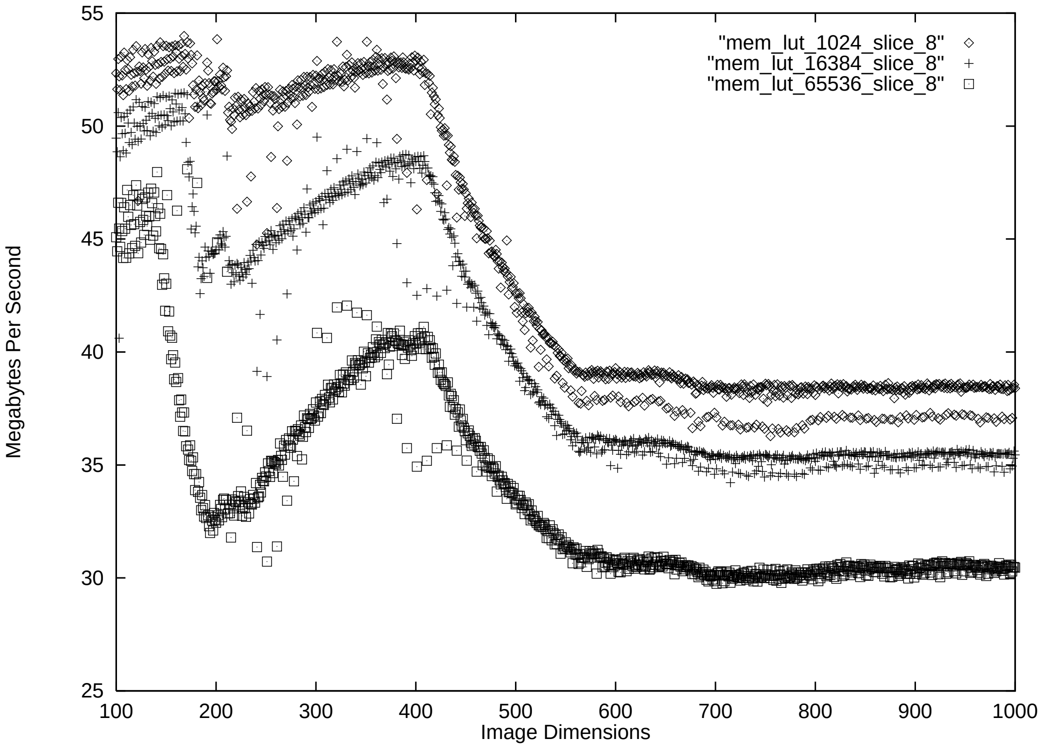

Figure 88: Relative speeds of memory (8) lookup for varying table sizes (Figures 82-83).

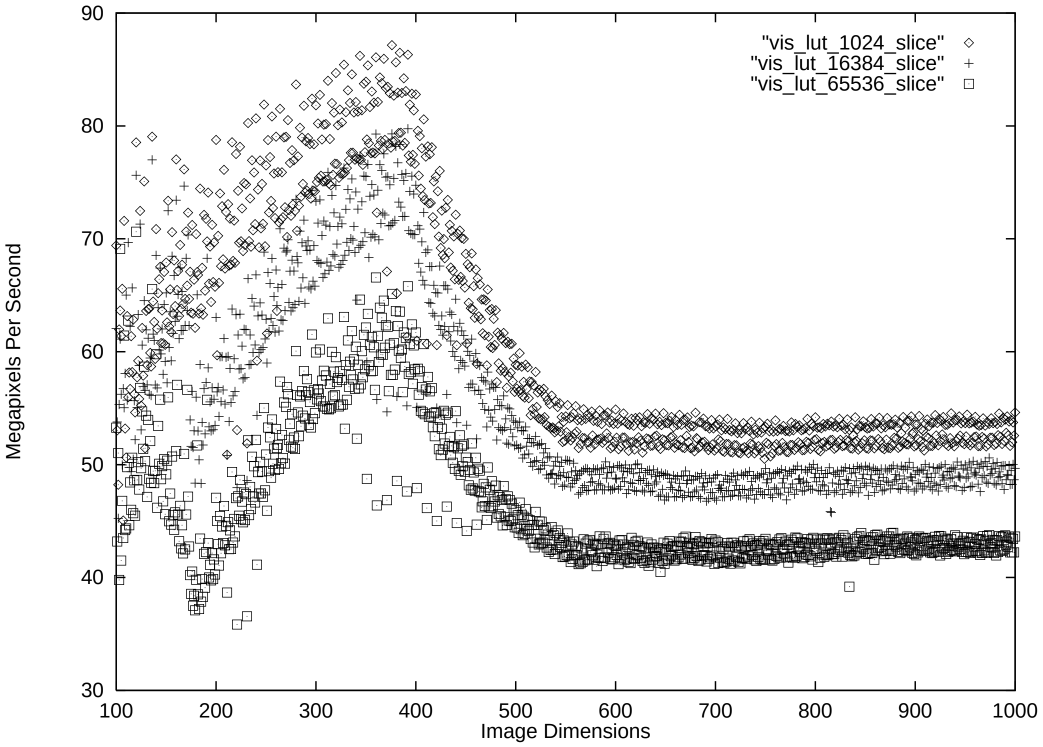

Figure 89: Relative speeds of VIS lookup for varying table sizes (Figures 84-85).

8.3 Convolution With Small Kernels

In this section we time a simple 3×3 convolution routine which does not use symmetry, separability, or any information about any special form the kernel might possess. This algorithm rereads each input line three times during three contiguous horizontal passes. Thus any cache miss on the source image will be amortized over three iterations, and accordingly will have a less dramatic effect than we saw for the previous, computationally simple, examples. This observation is borne out as we observe the memory and VIS XIL timing data, shown in Figures 90-91. The non-XIL VIS implementation did not run substantially faster than its XIL counterpart, and so is not displayed here.

Both plots show almost no dependence on the vertical image size, instead depending on the loop trip counts and the amount of horizontal data reuse. The horizontal oscillation displayed by the VIS code is due to the fact that as the image width increases, a full iteration is required for every eight additional pixels; dividing the stairstep function of time by the linear function of width yields the observed period-8 sawtooth pattern. The other obvious feature of both plots is the canyon-like depressions around 700^2 pixels. These appear to be caused by cache aliasing as the input image size almost matches the external cache size and begins to alias with itself.

Although the VIS loop is not modulo scheduled, probably because of excessive size and register pressure, it does not display the extreme oscillation of the simpler algorithms described in the previous sections. This is due to the large amount of computation relative to the number of loads and stores, reducing the relative impact of cache misses.

One strategy to increase convolution performance would be to break the loop up into a series of smaller, pipelineable stages. The results of each stage will be available in the internal cache for the next stage, so memory bandwidth *per se* need not be a source of performance degradation. If the amount of data passed between stages is excessively large, the number of load and store instructions will limit performance; this should be taken into account when deciding where to place the breaks between each pair of stages. Separable convolution would lend itself naturally to such an approach.

The practical results of such an approach may be seen below in section 8.4, as it is similar in spirit to the two-pass resampling discussed in section 7.5. Both the horizontal and vertical passes are modulo scheduled separately and interact through a relatively small buffer containing only a few scanlines worth of data. This multipass approach succeeded in producing an optimal schedule for each pipeline phase, with no additional arithmetic instructions. The extra loads and stores are irrelevant since they do not limit instruction execution and never cause cache misses. There is thus essentially no additional overhead beyond the cost of the loop prologues and epilogues.

The large number of instructions involved in convolution presents a difficult problem in instruction scheduling. The input data must be realigned before they can be handed to the multiplier. As the multiplier results become available, they may be accumulated by the adder in many different orders. The current paradigm of hard-coded VIS inlines implicitly requires the programmer to choose a particular dependency structure for the alignments, multiplications and additions that make up the computation. The compiler's role is then simply to schedule the tree of dependencies it has been given. A truly optimal approach would seem to require a search over at least a portion of the space of possible trees, which is impractical with current tools, or else a theoretical understanding of which dependency structures will result in the best final output. We will revisit this problem of algorithmic overspecification in a variety of guises in section 9, below.

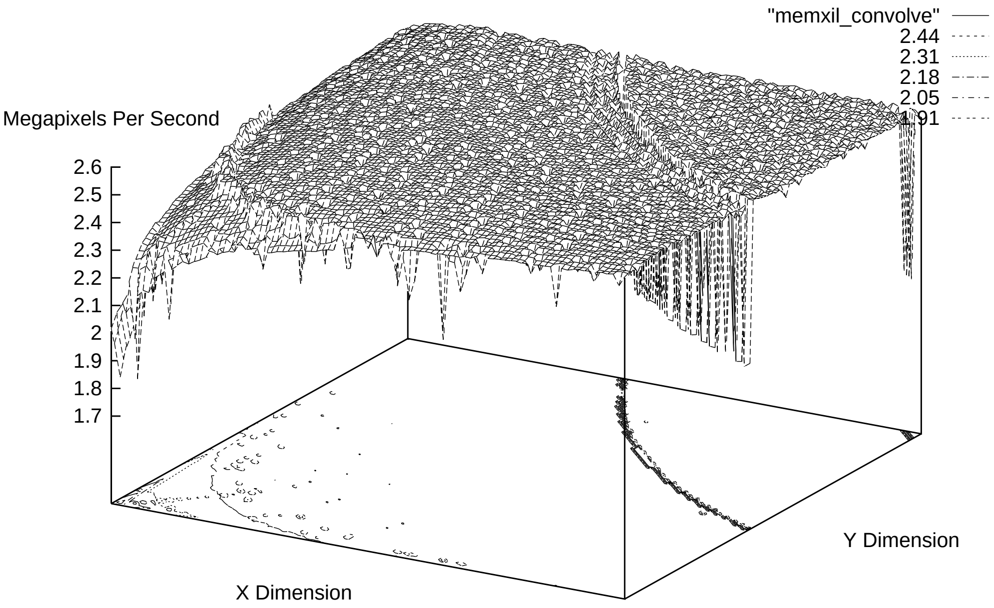

Figure 90: Times for the XIL memory implementation of convolution.

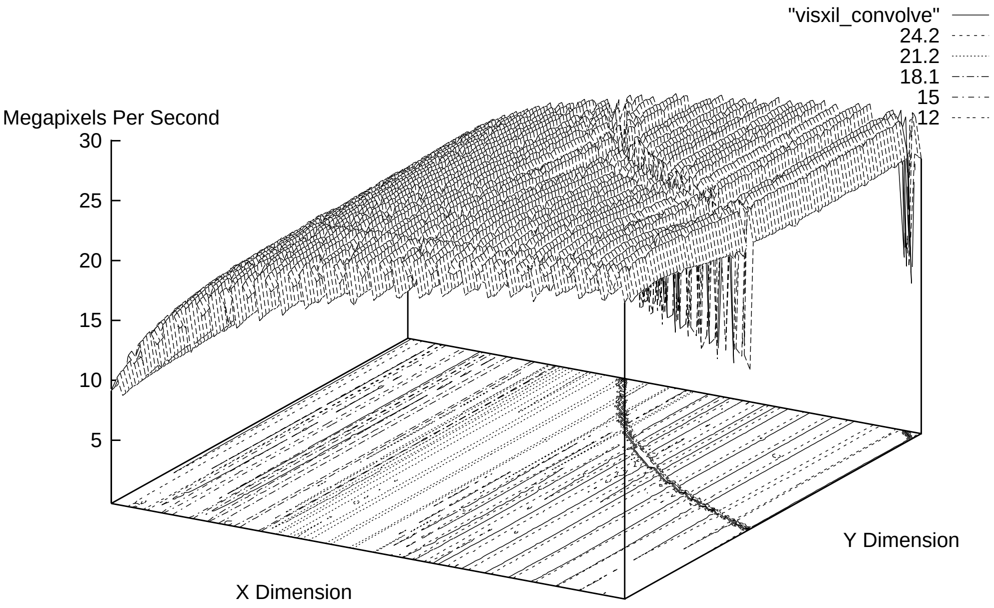

Figure 91: Times for the XIL VIS implementation of convolution.

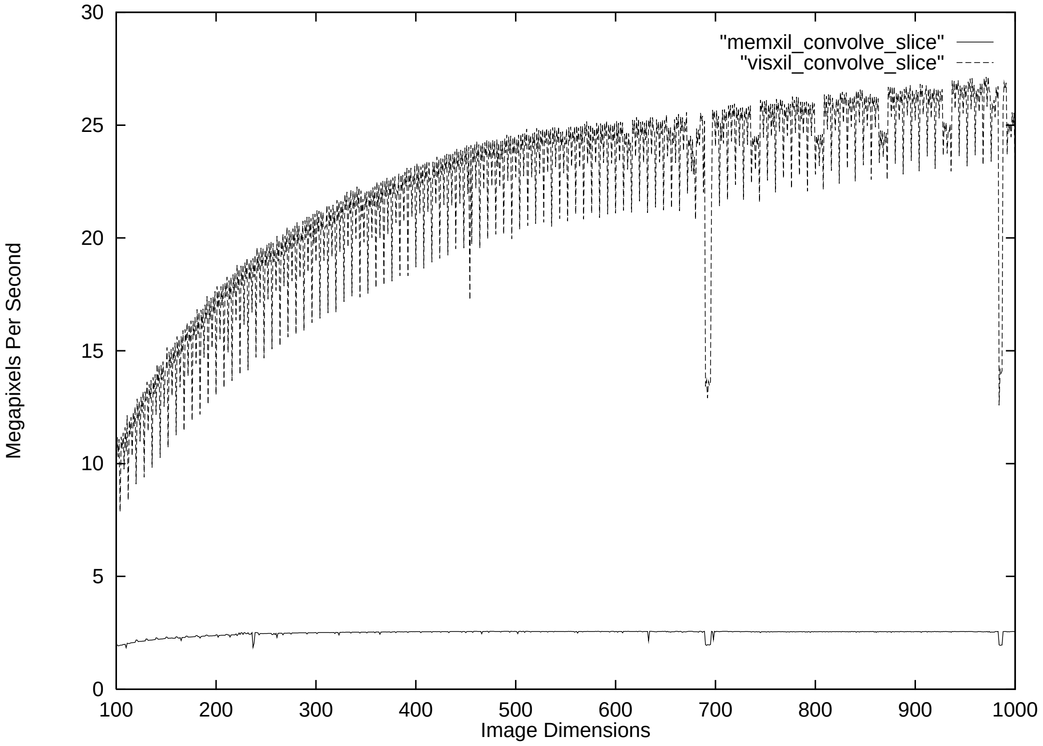

Figure 92: Comparative performance of two implementations of convolution (Figures 90-91).

8.4 Resizing Using Bicubic Interpolation

Figures 93 and 94 show the actual speeds of memory and VIS image resizing (using nearest neighbor, bilinear, and bicubic interpolation) as implemented in XIL. Rather than varying the image sizes as before, here we vary the magnification in order to highlight a different aspect of performance. The input image was large enough (1536×1024) to more than fill the 640×480 output image even at 50% magnification. The speeds are given in terms of output megapixels/second, where each pixel contains three bands. The VIS nearest-neighbor case uses a similar algorithm to the one described in section 7.5; horizontal resampling is accomplished simply by copying entries from `mbuf` into `obuf` and vertical resampling uses the system `memcpy` routine (which uses block loads and stores, as described in section 4.8) to copy each scanline into its place within the destination. The nearest neighbor and bilinear algorithms contain some special-case code: VIS at 100%, and memory at 200%. The 200% memory code operates at 29.8 megapixels/second for nearest-neighbor and 9.8 megapixels/second for bilinear, well outside the range displayed in Figure 93. These special cases will be ignored for the purposes of the discussion in this section.

Bilinear resampling is mathematically identical to the bicubic case except for the use of filters of width 2 throughout; the memory code makes use of the fact that the filters are simple triangles, whereas the VIS code is designed to use arbitrary coefficient tables. The general upward trend in all the VIS curves is due to the lesser quantity of input data that must be read and horizontally resampled to generate each output frame. The memory code displays a more constant profile, suggesting that it is compute bound throughout the range of scale factors displayed here. This constancy is also consistent with the use of a one-pass approach, which does not reuse any previous results.

Figure 95 shows the ratio of VIS performance to that of the memory implementation for each form of interpolation. We observe performance increases of up to 10.5× for nearest-neighbor, even though no filtering is performed – pixels are simply copied. The advantage comes from factors including:

- Reading and writing multiple bytes at a time.

  The source data is read in a strictly horizontal fashion into the input buffers, and as mentioned above the output writes use block loads and stores to transfer data from the horizontally resampled buffers to the output image very rapidly.

- Reuse of horizontally resampled scanlines.

  The work of horizontal resampling is performed at most once per input scanline, and the results are reused (trivially, in this case) by the vertical pass. This optimization pays off increasingly for higher magnifications.

- Preselection of input rows and columns to avoid branching.

  The XIL memory code uses a Bresenham scheme [Foley90] to determine which input row and column to sample from at each output pixel. The VIS XIL code makes this decision once for each row and column, placing the results into tables. These tables may be used during the actual resampling to select input pixels and filter coefficients without the need for conditionals.

- Fixed data formats.

  By writing code specifically for a given input and output format, loop nesting is reduced and opportunities for parallelism are increased (see section 9.2.1 for more on this idea). The XIL VIS code, as discussed above, can accept arbitrary formats but converts them into a fixed internal format as part of the buffering process. By contrast, the XIL memory code deals with arbitrary band formats, and must operate on data one byte at a time even within its innermost loops.

- Modulo Scheduling.

  The benefits of modulo scheduling are identical here to the previous examples.

We thus see that even a routine that might be categorized as computationally light may benefit significantly from a good choice of algorithm, proper consideration of system behavior, and use of compiler optimizations.

The speedups for bilinear and bicubic resampling at 300% magnification are 31× and 85×, respectively. The same factors described above in the nearest neighbor case still apply (with the exception of the use of block load and store), but additionally the use of partitioned arithmetic and clamping come into play.

As the magnification increases, the VIS resampling routines achieve an asymptotic performance determined mainly by the vertical resampling rate, since the amount of source data to be read and horizontally resampled becomes insignificant. The nearest-neighbor algorithm achieves an asymptotic rate of around 57 megapixels/second, bilinear achieves around 29 megapixels/second, and bicubic achieves around 25 megapixels/second.

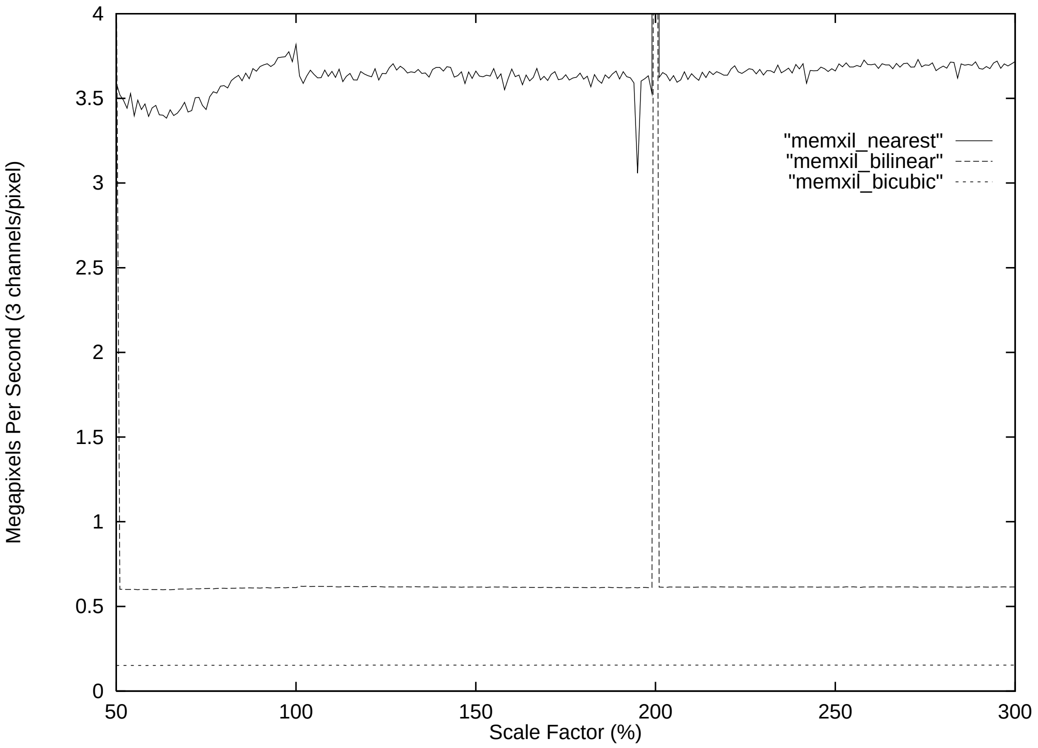

Figure 93: XIL memory resize speed as a function of magnification.

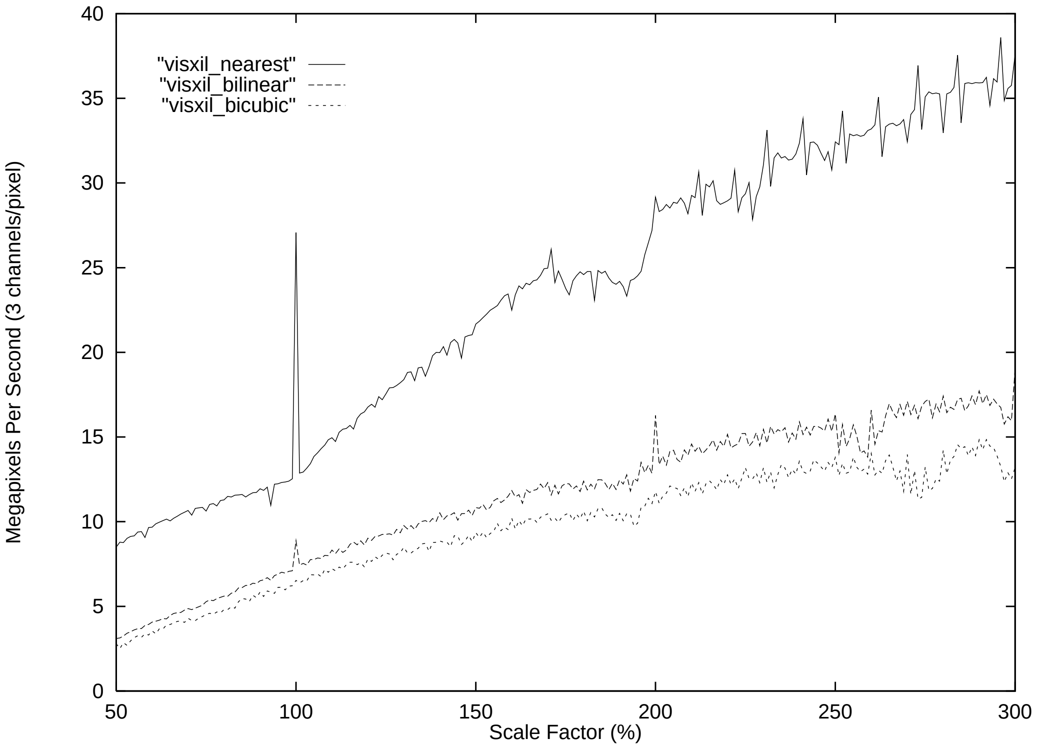

Figure 94: XIL VIS resize speed as a function of magnification.

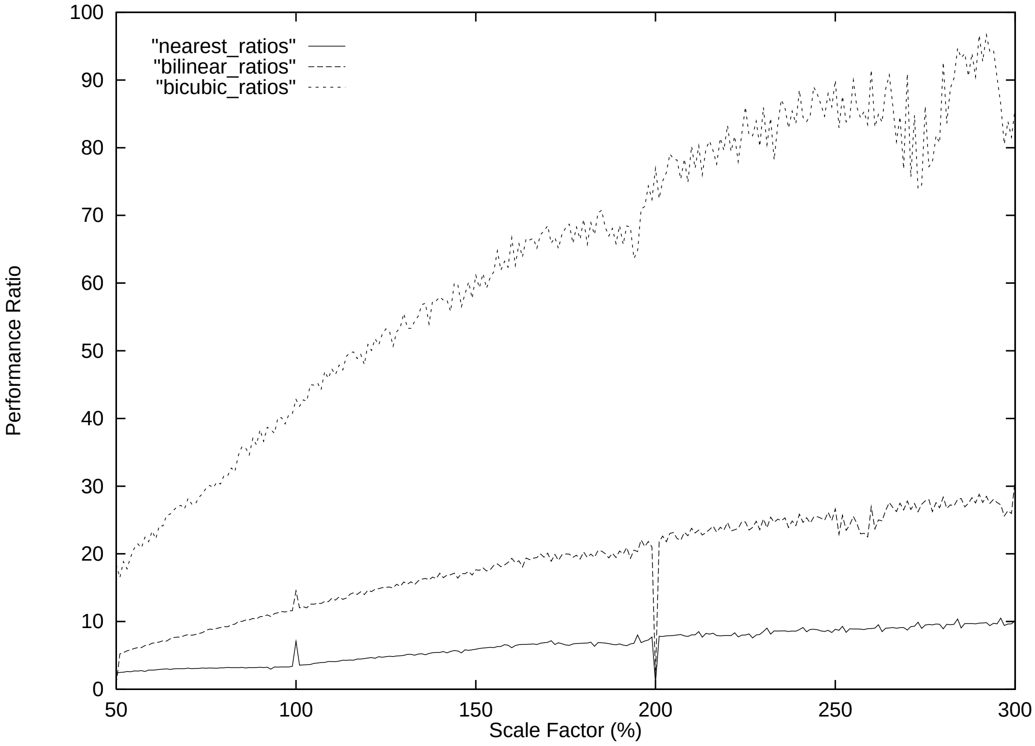

Figure 95: Ratios of VIS to memory resize performance.

8.5 Bilinear Scaling by Two

In order to test the bilinear scale function at a scale factor of two, a Sun-developed medical imaging testbed known as `qdxtool` (written by John Zimmerman) was used. This program displays a repeating sequence of images and optionally performs scaling, convolution, and contrast adjustment. A 136-frame sequence of 512×512 grayscale (one-banded) images was zoomed up to 1024×1024 and displayed. The XIL memory code's observed peak performance was 23.12 megapixels/second, although this includes the cost of copying its output to the frame buffer. This code is quite well written, and targets exactly the case at hand, as noted in the previous section. The frame buffer copy uses a routine equivalent to `memcpy` (see section 4.8), and normally operates at around 150 megabytes/second. Therefore we estimate that the true bilinear scaling performance was around $1/(1/23.12 - 1/150) = 27.33$ megapixels/second. This is roughly consistent with the speed of 9.8 megapixels/second observed on a three-banded image in the previous section, implying a one-banded performance of $9.8 \cdot 3 = 29.4$ megapixels/second.

The VIS XIL code, which follows the outline described in section 7.6, operated at 72.17 megapixels/second (writing directly to the frame buffer). We thus estimate a speedup of $2.64\times$.

8.6 Summary of Timing Results

Table 3 shows the results of the previous sections in tabular form. Each operation is represented by its average speed and standard deviation across the entire set of sample points, and the speedup is computed as the ratio of the average speeds (except for resize, where approximate bounds on the performance ratios are used). Clearly, the speedups for addition, blending, and lookup may be substantially increased by the use of tiling (or prefetching in future UltraSPARCs) in order to lower the cache miss rate. As we have seen, the modulo-scheduled VIS code is capable of achieving near-optimal performance for images that are contained entirely in the external cache.

Operation	XIL Memory Speed	σ	VIS Speed	σ	Speedup
Addition	9.6	3.5	88.3	30.5	9.2×
Blending	4.4	1.2	51.4	14.2	11.7×
Lookup (1K)	23.9	2.4	64.9	12.5	2.7×
Lookup (64K)	13.5	0.8	48.8	7.6	3.6×
Convolution	2.5	0.1	21.5	4.2	8.6×
Nearest Resize	3.7	1.7	23.4	8.7	2.3 – 10.5×
Bilinear Resize	0.66	0.6	11.5	4.1	5.2 – 30.9×
Bicubic Resize	0.15	$5e^{-4}$	9.7	3.3	16.4 – 85.8×
2x Resize	27.3	n/a	72.2	n/a	2.6×

Table 3: Summary of VIS versus memory timings

9 Enhancements and Future Directions

Sun has announced a long-term commitment to VIS. Currently it has been implemented without changes in the immediate successor to UltraSPARC-I and is being implemented with enhancements in that chip's successor. Backwards compatibility has been an absolute requirement at all stages, although the details of timing will change. In this section we relate some experiences and observations with respect to the process of reimplementation and enhancement as seen from a software designer's perspective. The details of future enhancements are still in flux at the time of this writing.

9.1 Enhancing VIS

During the course of VIS development, the author solicited ideas from other graphics software developers for new instructions and enhancements to existing instructions. Most suggestions related to small perceived defects in VIS or operations that would have been useful at one time or another during development. Some ideas from the list were:

- Floating point to integer register move instruction
- Support for two-dimensional or non-square arrays using the `array` instruction
- Partitioned shifting capability
- Detection or clamping of overflow and underflow
- Instructions to improve data formatting and rearrangement
- Various small enhancements to existing instructions

After discussion with the VIS architects and various proposals and evaluations of hardware complexity, the list was revised substantially. Some ideas, such as an instruction to move data between the register files, was rejected as being too difficult given the existing SPARC architecture. Instead, the hardware designers indicated that they were planning to implemented forwarding between the load and store buffers, which would have much the same effect. Other ideas were rejected as being amenable to software implementation. In particular, the `array` instruction will require software blocking for good performance; the software can present an array of arbitrary dimensions to the core rendering engine as a set of square blocks with minimal padding. Two dimensional texture mapping, as we have seen, can be performed efficiently with existing instructions and would not benefit greatly from the blocked style of the `array` instruction. Other proposed instructions dealt with issues of precision and overflow; however, for speed it is generally necessary to perform arithmetic in a way that guarantees that overflow cannot occur. Implementing a partitioned shifter was seen as simply too expensive to justify without a concrete application.

Some other proposed instructions were able to be implemented in terms of others. In particular, a general capability to shuffle bytes can stand in for a number of other suggested reformatting instructions and is amenable to hardware implementation.

Although few new instructions will ultimately have resulted from this process, it did provide an opportunity for software and hardware developers to exchange ideas about the proper division of responsibility for performance between the two and to forge an ongoing relationship. Hardware design cycles are such that designers are well along in their future plans by the time first silicon is available to software developers. It is up to software developers to gain experience with new hardware as quickly as possible and to provide feedback during the small remaining window of opportunity to influence future designs. In the author's case, simply being present at meetings where enhancements were discussed resulted in several opportunities to critique and alter proposals made by the hardware architects that would have been detrimental to VIS performance. Had VIS hardware and software designers simply interacted from a distance, the news of these proposals might have come too late to affect their fate.

When proposing enhancements, it is critical to understand their impact on performance. In general it was difficult to interest the hardware designers in any feature without a concrete demonstration of the speedup it would provide for some typical loop. In some cases, the cost of implementing a hardware feature would be increased latency for some class of operations with which it shares hardware resources. While software pipelining can limit the effects of latency on execution time, ultimately there will be some loops that are barely schedulable using the existing latencies due to high register pressure. These loops will slow down considerable if the relevant latencies are perturbed. Since it is difficult to know which loops fall into this category, the effects of increased latency are very hard to predict.

9.1.1 Characterizing VIS

During this process, there was a need to understand "what is VIS"? What principles did the current instruction set adhere to? Only by answering these questions is it possible to add new features that will coexist with the existing ones.

The properties characterizing the successful VIS instructions appear to be similar to those that characterize RISC in general. Instructions are pipelined and a new instruction may be issued and an old one completed every cycle under good conditions. Instructions operate between registers, and use extra data such as condition codes and the `%gsr` only when absolutely necessary. Memory is accessed in the normal way, so interaction with other routines and the operating system is simple. Regular SPARC instructions are responsible for address generation, alignment maintenance, conditional execution, and loop control, using simple addressing modes and regular arithmetic instructions. The key to VIS is that it identifies primitives that are useful in many settings and lets software put them together by means of ordinary instructions, unlike monolithic hardware that presents an all-or-nothing proposition to the programmer.

9.2 Towards Automatic Generation of VIS Code

Currently the compiler does not generate any VIS instructions except those explicitly invoked by means of an inline template. The programmer must structure loops especially for VIS, dealing with alignment issues, unrolling to generate 4 or 8 output values, readahead, and so forth. Clearly it would be desirable to have a tool to convert a simple specification of a desired function into "optimal" VIS code. In this section we discuss some techniques that could ultimately make such a tool possible.

9.2.1 Partial Evaluation

Partial evaluation refers to the specialization of programs with respect to some of their arguments. A program `p` with arguments `in1` and `in2` computes some output:

$$\texttt{out} = [\![\texttt{p}]\!]\ [\texttt{in1}, \texttt{in2}]$$

A *specializer* is run on `p` and `in1` to yield a new program $\texttt{p}_{\texttt{in1}}$ with a single argument `in2`. This program obeys the equation:

$$[\![\texttt{p}_{\texttt{in1}}]\!]\ \texttt{in2} = [\![\texttt{p}]\!]\ [\texttt{in1}, \texttt{in2}]$$

The resulting specialized program may be substantially faster than the original (although only by a linear factor). Jones et al. [Jones93] survey many aspects of this topic.

Specialization is typically defined by syntactic induction on the grammar of a language. For example, an expression with constant components may itself be reduced to a constant. A construction `if (a) b else c` can be specialized to simply `c` if the specializer knows that the expression `a` evaluates to `0`. A `for` loop with a known trip count may be unrolled completely. A `switch` statement with a known value may be replaced by the relevant case or cases being selected. Code that can be shown never to be called can be eliminated. So far these optimizations are within the reach of traditional optimizing compilers. However, partial evaluation is capable of combining and integrating these simple transformations to a greater degree than traditional compilation, sometimes resulting in radically altered code. The classic example of the power of partial evaluation is the observation that specializing an interpreter for a language $\mathcal{L}$ written in a language $\mathcal{I}$ for a particular program is equivalent to compiling that program into language $\mathcal{I}$; specializing the partial evaluator itself with respect to the interpreter results in a compiler from $\mathcal{L}$ to $\mathcal{I}$ for arbitrary programs!

The potential for application of partial evaluation to imaging is great. Many hand optimizations that have proven useful can easily be viewed as instances of partial evaluation. For example, conditionals in a loop whose arguments will be constant over the lifetime of the loop may be eliminated by generating multiple versions of the loop. The ability to write such conditionals would simplify the maintenance of code dependent on factors such as filter width and number of channels. An unrolled loop with conditionals may lose its conditionals when unrolled if those conditionals vary in a repeating pattern (perhaps based on alignment). These transformations offer library programmers an opportunity to avoid massive special casing along multiple axes and thus save coding time, avoid bugs, and simplify maintenance without run-time penalty. In effect the existing benefits of high-level programming are reified by the introduction of a further level of abstraction over and above that of compilation. The programmer need no longer avoid particular structures due to a perceived impact on performance. Although the object code size of an imaging library using these techniques will be increased, the actual portion of the object used by a particular application will probably not grow by much, since each application tends to use a small subset of the library's functionality.

A concrete example of the special casing discussed above is the choice of filter width during image resampling. In practice, using a `for` loop and array indexing to implement the process of convolving a series of pixels with a filter was rejected as unacceptably slow since the ability to schedule the outer loop would be impaired. Instead, a common piece of code was macro processed to create separate functions for each possible width from 2 to 8. Had it been possible to direct the compiler to translate the `for` loop into a set of special cases plus a single general case, code maintenance would have been simplified significantly. Partial evaluation could also potentially automate some of the code transformations described in section 8.4, where we observed a $10\times$ performance increase largely due to specialization on image format and hoisting of redundant computation out of the inner loops.

A more advanced use of partial evaluation is automatic combination of imaging functions. XIL supports the notion of a *molecule*, which is a single procedure implementing a sequence of atomic operations. Molecules are implemented by deferring execution of each called primitive and inserting it into a DAG (directed acyclic graph) structure describing the work to be done. Pattern-matching is performed on this DAG to locate sequences corresponding to predefined molecules. Certain events, such as device input and output or the reading of a pixel value, force evaluation to occur.

An example of a useful molecule is conversion of a sequence of horizontal and vertical convolutions into a single separable convolution. In this case, the code implementing the separable convolution may differ substantially from the code implementing the one-dimensional convolutions. For many other possible molecules, such as calculation of a simple arithmetic expression involving several images, it would be desirable to combine the existing routines for each operation into a custom molecule. Such molecules would make fewer memory references, since there would be no need to write and read from an intermediate image. This would not only reduce instruction count but would lessen the routines' cache footprint. Also, combining the code would allow the optimizer to locate a greater degree of parallelism, e.g., between the graphics adder and multiplier, as well as to perform other optimizations such as elimination of the common subexpressions and dead code. Common subexpressions will most likely exist since each loop performs similar computations such as pointer updating; dead code arises from the fact that some routines may check for cases their predecessors never generate. Techniques from vector processing, such as loop fusion, could be used to force all the code into a single nested loop. Since there are too many potential combinations of primitive routines to provide code for all of them, some mechanism for users to request specific combinations would be useful. Source-to-source transformation techniques inspired by those of partial evaluation will most likely be needed to perform ambitious automatic combination of code.

The success of partial evaluation depends somewhat on language features. Functional languages are the simplest to specialize; imperative languages, particularly those with heavy aliasing such as C are much harder. An imperative but restrictive language such as Java [JavaSoft95] or a subset of C combined with partial evaluation could offer substantial benefits to the imaging coder. In general, a more flexible approach to source-level transformation could ease many programming tasks.

9.2.2 Automatic Loop Parallelization

The common form of many VIS loops may potentially be derived from any suitable array-processing loop by a proper vectorizing code generator. Consider a code generator that combines work from adjacent loop iterations, emitting vector loads, stores, and arithmetic operators. The vectors, initially the length of the destination row, may be divided into a series of 8-vectors and an optional final vector of length 7 or less (*strip-mining*). This may be performed automatically without difficulty. The alignment of the destination vectors can be ensured by suitable prologue code. The source vector loads may be synthesized as:

```
ptr_a = vis_alignaddr(ptr, 0); ptr += 8;
tmp_hi = *(ptr_a);
tmp_lo = *(ptr_a + 1);
val = vis_faligndata(tmp_hi, tmp_lo);
```

This construct, in the context of a loop, may be optimized by noting that `ptr_a` increases by 8 when `ptr` increases by 8; by hoisting the initial alignment of `ptr_a` out of the loop and treating `ptr_a[]` as a first-class array we can use standard array analysis techniques to recognize that `tmp_hi` is equivalent to `ptr_a[i]` and `tmp_lo` is equivalent to `ptr_a[i + 1]`. Furthermore, we see that `tmp_lo` is equal to `tmp_hi` from the previous iteration; this allows us to reduce the load requirement to one per iteration. If the alignment of `ptr` is known at compile time to equal 0, the vector load can be rewritten as an ordinary load.

The generation of VIS arithmetic instructions involves thornier issues. In particular, the semantics of the VIS multiplication instructions do not match those of any standard programming language. The programmer will probably still have to declare the desired precision and range of any intermediate values in order to generate reasonable code.

It should be possible to reimplement much of the existing body of VIS code using a simple, array-oriented notation with properly specified intermediate precisions. After all, the problem of clamped addition described in section 7.1 may be specified as simply:

```
D = clamp(S1 + S2)
```

in a language possessing an image datatype and providing semantics for addition and clamping identical to those of VIS. Many common functions can be similarly expressed using only a handful of arithmetic primitives, and it would be highly desirable to be able to produce optimal code for them on future architectures without the need for rewrites or special hardware expertise.

VIS code generation frequently involves choices between various functionally equivalent ways of computing the same result that have different hardware requirements and/or dependency structures. Choosing between `fmul8x16al` and `fexpand` is one example; summing a set of values using either a tall or wide addition tree is another. In both cases, it is not possible to make an informed choice between them without looking forward somewhat in order to take the schedule constraints into account. This is an instance of the more general problem of phase ordering during code generation; any allocation of the code generator's job into phases will produce a suboptimal result in general since the consequences of decisions made by one phase cannot be fully known until a future phase. A classical example is the ordering of scheduling and register allocation; if register allocation takes place first, antidependencies will be created that are detrimental to scheduling. If scheduling takes place first, it will be unable to take spilling into account.

A recent technique known as mutation scheduling, proposed by Novack and Nicolau [Novack94] offers a way to make code generation decisions within the scheduler. Alternative translations for intermediate code structures are supplied by the compiler-writer. At compile time, each expression is represented by a set of possible "mutations," i.e., synonymous translations with possibly different resource requirements. Instruction selection and placement as well as register allocation, spilling, and rematerialization are performed concurrently. Architectural features such as the combined multiply-add of the Intel i860 (which requires both instructions to be launched in the same cycle) are represented by offering two translations of a multiply-add expression, one using separate instructions with no issue constraint and one in which the instructions are combined. The combined form will be used only if it leads to a better overall schedule during the optimization phase. Such integrated, heuristic-driven techniques appear to have great promise to bridge the gap between parallel algorithms expressed in high-level notation and the fine-grain parallelism of VIS-like instructions.

The challenge of all automated code-generation techniques is to find a balance between specificity and generality. A highly specialized, VIS-only code generator would greatly simplify VIS coding but would not port easily to other acceleration architectures. A general vector-based code generator could be specialized to many platforms but would have difficulty with the peculiarities of each particular architecture. This is an issue that must be tackled if VIS or its successors are to be used across multiple platforms.

9.3 Implementing VIS in Future Generations of SPARC

As was mentioned earlier, VIS will be a feature of multiple SPARC designs spanning several processor generations. In this section, we discuss some of the practical ramifications of this fact for the VIS software developer.

9.3.1 Prefetching

The SPARC v9 instruction set provides a `prefetch` instruction, allowing data to be read from memory into cache or some other buffer memory for future use. Such an instruction appears ideal for use in imaging code, with its highly predictable memory access patterns. Since the exact order in which source pixels will be read is known advance for most important algorithms (although perhaps not in, say, an image warping algorithm), there is no difficulty in issuing a prefetch for pixels to be used many cycles later – in particular, longer than the delay imposed on a cache miss. This would allow code to bypass the cache entirely, providing near-constant performance on image data anywhere in main memory. In fact, the prefetch instruction allows the data to be fetched and deliberately *not* placed in cache; this allows the cache to be used preferentially for data such as lookup tables which are accessed non-sequentially without the problems of overwriting discussed in section 2.3.

The issues surrounding prefetching, such as the ideal number of pixels one should read ahead of the current one and the secondary effects of not placing images in cache (slowing down code which does not use prefetching) have yet to be examined in the context of VIS imaging. In particular, there are trade-offs of performance and implementation complexity between the use of hardware prefetch, in which the processor attempts to detect the stride between adjacent loads and automatically anticipates the next load address; compiler-generated prefetches, which should be correct for straightforward loops such as those that are currently modulo scheduled, but which may have difficulty extending beyond inner loops; and manually inserted prefetch instructions, which can make use of the programmer's understanding of an algorithm's memory access patterns. Experimentation is required to determine the relative merits of these options.

A processor which implements prefetching may also have some disadvantages, in that it may be less aggressive than UltraSPARC-I about continuing to do work during cache miss cycles. Thus prefetching may be a practical necessity for good performance and not merely an additional option.

9.3.2 Hardware Implementation Complexity

The VIS instructions account for only a small portion of the gate count of the UltraSPARC-I processor. However, one must take into account the fact that both the instruction definitions and their implementation were determined in tandem.

As an example of this principle, consider suppressing some of the carries in an adder in order to obtain a partitioned version. This may require few additional gates for the particular adder design used in one generation of processors. If future hardware designers wish to use a more sophisticated design with a highly optimized carry chain for the main floating point adder, they may be forced to include a separate partitioned adder in order to avoid extra gate delays in the main adder's carry chain. Thus the small initial investment in gates may become substantial over time.

Fortunately the capacity of processors will continue to increase substantially from generation to generation, so the need for some extra gates is not fatal in and of itself. However, some architectural changes may incur performance penalties that cannot easily be alleviated by increased hardware. For example, the `edge` instructions are defined to produce a condition code when their arguments point to the same word of memory. This behavior was essentially free since the `edge` instructions make use of the UltraSPARC-I's existing integer comparator. In a future UltraSPARC design currently being architected, the form of the `edge` instructions requires them to be processed in a special functional unit, rather than the integer unit, if the overall chip architecture is not to be violated. This unit would not otherwise contain comparison hardware. Worse, a condition code generated in this unit would not be available for several cycles, requiring the processor to stall on each `edge` instruction, whether or not it is used. As noted above, the condition code is difficult to use from C and the trip count for most VIS loops is easily computed. In order to avoid paying a penalty for the unused functionality, additional `edge` variants that do not generate condition codes may be required. Even with this addition to VIS, old code will have to be modified to use the new, non-backwards compatible variants in order to realize a speed increase. Thus we see that it is possible to introduce architectural features whose cost is only realized several generations in the future.

Certainly any problem of this sort can be solved with enough additional hardware – e.g., `edge` instructions could have their own custom unit. The price for this sort of solution goes beyond mere increased gate count, however. Increased design and simulation resources, area, and power consumption all add to the cost. The mere possibility that a new critical path might be introduced will cause substantial design time to be spent on optimization. Extra complexity must be added to the instruction decode and grouping logic to deal with the new units. All this adds greatly to the exposure of the entire design in terms of time-to-market and bug count. It is difficult in the extreme to justify this exposure for any but the most critical aspects of the processor architecture.

9.3.3 Binary Compatibility

In a multigeneration family of processors, binary compatibility is typically a top priority, with the proviso that buggy programs need not fail in precisely the same ways. In the case of superscalar processors, this means that program order semantics must be rigorously maintained since code cannot be depended on to obey instruction latencies and grouping rules. Beyond the question of compatibility, however, lies the murky area of legacy code performance.

Part of the RISC philosophy is the notion that hardware and software (e.g., compilers) must work together to optimize resource usage. In the commercial world, however, recompilation is not the trivial process that it may appear to be. It takes time for a software vendor to qualify a new compiler for use in their production environment, and the logistics of maintaining and shipping several compiled versions of their applications may be undesirable. Thus it would be a mistake for hardware designers to benchmark their designs using only custom-compiled code. Conversely, techniques such as out-of-order execution and register renaming that attempt to dynamically reschedule code may provide a solution to this problem, but at unacceptable design cost.

In the case of VIS, these problems are exacerbated by several factors. First, the lack of good benchmarks makes it difficult to begin to quantify the effects of a design decision on either legacy or recompiled code. Second, the VIS instructions are completely opaque to the compiler. An operation such as summing a set of floating point numbers may be rearranged by a compiler that understands that addition is transitive and commutative; such rearrangement can uncover parallelism by allowing additions to be scheduled as their arguments become available, rather than relying on an arbitrary order imposed by the parser-generated syntax tree. In the VIS case the compiler has no information about the meaning of `fpadd16`, say, and so cannot perform any restructuring on the dependency graph. VIS code that has been hand-optimized to provide a good match to existing hardware will not be able to take the best advantage of future hardware without some rewriting. Even advanced techniques like those described in section 9.2.2 will not help unless some higher-level abstractions are used to insulate code from the specifics of one implementation of VIS.

10 Conclusions

A number of very high-performance functions have been written using VIS. By and large these functions substantially exceed initial performance estimates, mainly due to the improvements in compiler optimization between the time that VIS was designed and the present. Complex routines such as interpolated scaling outperform the previous generation of imaging hardware (e.g., the SX) by $20 - 40\times$, and generic CPU implementations by up to $85\times$. Early results of video compression experiments appear extremely promising. Not only arithmetic but other areas of data-parallel processing are fair game for instruction set enhancements. Even within imaging, many tasks are limited by the need to shuffle bytes rather than by arithmetic. For example, extraction of data with an odd number of bands into separate images cannot be performed conveniently using the `fpmerge` operation, but could benefit from a more general permutation capability.

The implementation cost of VIS is low but not non-negligible. Functionality that fits into the leftover die space of one processor generation (because it requires only small additions to existing hardware) will not necessarily continue to do so as the fundamental hardware is redesigned, even as dies become ever more capacious. This raises the specter that the need to provide backward-compatible support for VIS might someday force other important functions out of the hardware or materially delay an implementation.

As more aggressive designs attempt to shorten the chip's critical paths, VIS paths may eventually have an effect on processor speed. It will be extremely difficult to weigh the importance of VIS against a decrease in SPEC performance should this occur. In any case, designers who wish to implement special instructions must understand that this entails a long-term commitment of design resources, and is not the relatively trivial undertaking it may appear to be.

The VIS instructions work well with the general RISC principles of the rest of the instruction set – they are for the most part fully pipelined, use a fixed amount of hardware resources, leverage the existing memory interface, and do not cause any exceptions. Thus they are not difficult to integrate with the compiler's machine model. Instructions lacking these properties would be extremely difficult for the optimizer to handle. Although it is possible to imagine larger-scale instructions to perform multi-cycle tasks such as an entire MPEG decoding stage, the ability to integrate VIS with regular processing appears to outweigh the programming convenience of such an approach.

It is doubtful whether instruction set extensions are worthwhile unless they are supported by the compiler. Given a choice of speeding up generic code by a factor of 2 or 3 from general tuning and turning on optimization flags, and writing VIS-like code from scratch and achieving a speedup of 4 or 5, the former approach becomes tempting. The speedup for using VIS must be in addition to, not instead of, that which is achieved by the optimizer in order to justify the man-years required for algorithm development, coding, tuning, and debugging any non-trivial amount of VIS code.

The experience at SMCC has shown that assembly coding is not realistic, even with a dedicated team of skilled programmers. Only a handful of assembly routines were successfully written and debugged, as the compiler performance began to exceed that of the initial hand-written loops. Several assembly routines were abandoned and rewritten in C because they showed a net performance disadvantage. This came as a surprise to those who associate hand-coding with speed; indeed, this belief has been well justified until recently. Modern processors, and in particular superscalar processors, demand a greater amount of parallelism to be made explicit for optimal performance than a hand-coder is likely to be able to provide. The amount of bookkeeping required to emulate the modulo scheduling algorithm, for example, by hand is massive and error-prone, and all intermediate results must be discarded if there is even a tiny change to the code. Since bugs will always be present, it is infeasible to leave performance tuning for the last minute. Instead, the programmer must make performance a priority from the time algorithms are chosen until the code is shipped. Even switching compiler flags is fraught with hazards since the optimizer's support for any new piece of hardware will have bugs that must be uncovered as early as possible in the design cycle. Only by consistently exercising the same tool set (simulator, compiler, optimizer, debugger, assembler, profiler) throughout the process will the required degree of integration with the new hardware be realized.

A crucial benefit of leveraging the compiler is the ability to reuse code between processor generations. Future processors from SME, although they will offer complete binary compatibility with v9 and VIS, will have somewhat different grouping rules and latencies between instructions. Certain specific features of UltraSPARC-I such as non-blocking loads may be replaced by other mechanisms such as prefetching. It is impossible to write code that is completely insensitive to such changes, but C code with inlined VIS instructions does provide a fairly high level of abstraction that will allow future compilers to reschedule the code as well as insert some prefetching instructions automatically. Some coding decisions, such as the replacement of `fexpand`s by multiplications, are specific to the functional units of the current processor and may backfire in future designs. If these decisions are documented appropriately, however, they will not be difficult to reverse as the need arises. Still, the desire to run existing code as-is in the future should not be underestimated.

The true philosopher's stone of CPU-based imaging is automatic generation of VIS-like instructions from regular, high-level source code in a serial language. This is an intrinsically difficult task; much of the existing VIS code required clever algorithmic choices, an understanding of the memory model with respect to faults, and an analysis of precision requirements. Still, there is some hope that the relatively simple ad-hoc routines used by many imaging applications could be generated automatically from a limited number of algorithmic templates. The literature does contain some automatic loop parallelization techniques that could in principle be adapted to such a task. C, with its unrestricted pointer arithmetic, is a particularly difficult source language for such algorithms, but a more restrictive language (e.g., Java, which disallows all pointer arithmetic and most types of aliasing) might offer adequate facilities for developing imaging code while allowing sufficient automated analysis to recognize opportunities for complex instructions to be used. A number of interesting threads within the compiler literature that have been largely abandoned since the advent of RISC, might fruitfully be picked up again by a new generation of researchers.

11 Acknowledgements

This work represents a combination of skills learned at Berkeley and real-life experience porting XIL and Adobe Photoshop at SMCC. The opportunity to be among the first users of VIS was the result of being in the right place at the right time. It has been a very rewarding and educational experience, proving that the real world can sometimes be the best classroom. I left Berkeley to pursue this opportunity and I now thank my advisor Carlo Séquin for allowing me to come full circle and to write this report under his auspices. I especially thank him for all of his patience. I also thank Professor David Culler for volunteering to act as second reader and spurring me to provide a more quantitative look at VIS.

My coworkers on the VIS/XIL team, Peter Farkas, Steve Howell, Aman Jabbi, and Jaijiv Prabhakaran as well as former members Walt Donovan, John Recker, Ray Roth and Chang Zhou deserve special recognition. Two members of the XGL porting team, Grace Wang and Vikas Deolaliker, were also part of the VIS effort. We all tackled VIS together in a spirit of cooperation and overcame many obstacles to turn its promise into reality.

My current and former managers, Ihtisham Kabir, Roger Day, and Rob Mullis, offered me this opportunity to work and learn at Sun's expense, for which I am grateful.

Les Kohn, Marc Tremblay, and many others at STB brought VIS and UltraSPARC to life and answered more than a few questions along the way. I hope we have realized their ambitions and paved the way for a great success. Partha Tirumalai, Nand Mulchandani, and Chris Aoki were especially helpful in resolving all of our compiler issues.

Thanks to Gary Lauterbach at SME and Mike Lavelle in SMCC graphics for taking my VIS wish lists seriously and allowing me to be part of the design process for future generations of UltraSPARC.

I thank Jon Ferraiolo, Allen Chan, and Scott Byer at Adobe Systems for all their cooperation as I tried to create the "World's Fastest Photoshop." Photoshop is a wonderful learning tool and I wish everyone interested in high-performance imaging an equally interesting and enlightening experience.

My former officemates in Evans Hall, Glenn Adams, Chedsada Chinrungrueng, Mark Brunkhart and Roger Bush provided many interesting distractions and a chance to waste the last of my youth in pleasant company. Thanks also to the rest of the Carloids: Ajay Sreekanth, Maryann Simmons, Laura Downs, and Rick Bukowski, for making Berkeley Graphics such a friendly place to be. I would also like to thank Raph Levien for many interesting discussions.

The love and support of my parents, Alison and David, is with me always. Without them none of this would have been possible. I also dedicate this report to my grandmothers, Anne Berlin and Anne Rice Brown, and to the memory of my grandfathers, Samuel Berlin and Abraham Rice.

References

[Briggs92] Briggs, P., K. Cooper and L. Torczon, "Coloring Register Pairs," *ACM Letters on Programming Languages and Systems* (1:1), March 1992, pp. 3-13.

[Chaitin81] Chaitin, G., M. Auslander et al., "Register Allocation Via Coloring," *Computer Languages* 6(1981), pp. 47-57.

[Chaitin82] Chaitin, G., "Register Allocation & Spilling Via Graph Coloring," *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction* (June 1982), pp. 98-105.

[Deering94] Deering, M., S. Schlapp. and M. Lavelle, "FBRAM: A New Form of Memory Optimized for 3D Graphics," *Computer Graphics*, July 1994, pp. 167-174.

[Donovan95] Donovan, W., P. Sabella, I. Kabir, and M. Hsieh, "Pixel Processing in a Memory Controller," *IEEE Computer Graphics and Applications*, January 1995, pp. 51-61.

[Ebcioglu94] Ebcioglu, K., R. Groves, K. Kim, G. Silberman, and I. Ziv, "VLIW Compilation Techniques in a Superscalar Environment," *ACM SIGPLAN* June 1994, pp. 36-48.

[Fisher79] Fisher, J. A., *The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources*, Ph.D. dissertation, Technical Report COO-3077-161. Courant Mathematics and Computing Laboratory, New York University, New York (October 1979).

[Foley90] Foley, J., A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice* (second edition), Addison-Wesley, New York, 1990.

[Granlund92] Granlund, T. and R. Kenner, "Eliminating Branches using a Superoptimizer and the GNU C Compiler," *Proceedings of ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 341-352.

[Greenley95] Greenley, D., J. Bauman et. al., "UltraSPARC (TM) : The Next Generation Superscalar 64-bit SPARC," *IEEE CompCon Spring '95 Digest of Papers*, pp. 442-451

[Intel89] i860™ *64-Bit Microprocessor Programmer's Reference Manual*, Intel, Santa Clara, 1989.

[JavaSoft95] Java documentation is currently only available online on the World-Wide Web at `http://www.javasoft.com/doc.html`.

[Johnson91] Johnson, M., *Superscalar Microprocessor Design*, Prentice Hall, New York, 1991.

[Jones93] Jones, N., C. Gomard, and P. Sestoff, *Partial Evaluation and Automatic Program Generation*, Prentice Hall, New York, 1993.

[Kohn95] Kohn, L., G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner, "The Visual Instruction Set (VIS) in UltraSPARC (TM)," *IEEE CompCon Spring '95 Digest of Papers*, pp. 462-469

[Lam88] Lam, M., "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 318-328.

[Lam90] Lam, M., "Instruction Scheduling For Superscalar Architectures," *Annual Review of Computer Science*, 1990, volume 4, pp. 173-201.

[Lee95] Lee, R., "Realtime MPEG Video via Software Decompression on a PA-RISC Processor," *IEEE CompCon Spring '95 Digest of Papers*, pp. 186-192.

[MPEG93] ISO/IEC JTC1/SC29/WG11, *Information Technology — Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to about 1.5 Mbit/s — Part 2: Video*, ISO/IEC 11172-2, 1993.

[Novack94] Novack, S. and A. Nicolau, "Mutation Scheduling: A Unified Approach to Compiling for Fine-Grain Parallelism," *Languages and Compilers for Parallel Computing*, Springer-Verlag LNCS No. 892, 1994.

[Rau81] Rau, B. and C. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing," *Proceedings of the 14th Annual Workshop on Microprogramming* (October 1981), pp. 183-198.

[Rau92] Rau, B., M. Lee, P. Tirumalai, and M. Schlansker, "Register Allocation for Software Pipelined Loops," In *Proc. ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 283-299.

[SGI93] Silicon Graphics, Inc. *ImageVision Library Programming Guide* Document Number 007-1387-030, 1993.

[SME95a] *UltraSPARC-I User's Manual* Sun Microsystems, part #STP1030-UG, 1995.

[SME95b] *Visual Instruction Set User's Guide* Available from Sun Microelectronics as part of the VIS developer's kit, which includes the SPARCCompiler 4.0, Incas simulator, and VIS header and inline files.

[SPARC94] *The SPARC Architecture Manual*, Prentice-Hall, New York, 1994.

[SunSoft94] *XIL 1.2 Programmer's Guide* SunSoft Document Number 801-6936-10, 1994.

[Tirumalai96] Tirumalai, P., D. Greenley, B. Beylin, and K. Subramanian, "UltraSPARC: Compiling for Maximum Floating Point Performance," *IEEE CompCon Spring '96 Digest of Papers*, pp. 408-416.

[Ward89] Ward, J. and D. Cok, "Resampling Algorithms for Image Resizing and Rotation," *SPIE Digital Image Processing Applications*, 1989, pp. 260-269.

[Wilf86] Wilf, H., *Algorithms and Complexity*, Prentice-Hall, New York, 1986.

[Wolberg90] Wolberg, G., *Digital Image Warping*, IEEE Computer Society Press, Washington, 1990.

[Zhou95] Zhou, C., L. Kohn, D. Rice, I. Kabir, A. Jabbi, and X. Hu, "MPEG Video Decoding with the UltraSPARC Visual Instruction Set," *IEEE CompCon Spring '95 Digest of Papers*, pp. 470-475.