An Evaluation of Native Signal Processing on the UltraSPARC-II with Sonar Beamforming Algorithms

Gregory E. Allen
EE 382M – Dr. Lizy John – Fall 1998
Computer Performance Evaluation and Benchmarking
The University of Texas at Austin
gallen@arlut.utexas.edu

Abstract

The high performance of today's general purpose processors makes it feasible to perform digital signal processing on the main CPU of a workstation. We develop and evaluate the performance of two sonar beamforming kernel routines for execution on Sun UltraSPARC-II processors. The selected kernels evaluate both single-precision floating point and Visual Instruction Set performance. Because these algorithms operate on large, continuous streams of data, memory latency hiding techniques, including software prefetching, are necessary for maximum performance. We present performance results for these kernels, and evaluate the effects of software prefetching and VIS implementation. We find that loop unrolling gives excellent results in nearly all cases. We also find that native signal processing performance is very good, but that extensive hand-coding and optimization is necessary in order to achieve the best results.

1. Introduction

Today's high-performance general-purpose CPUs make it feasible to perform substantial signal processing on the main CPU of a workstation. The term "native signal processing" (NSP) has been used to describe this approach [1]. Single-cycle multiply-accumulates (MACs), traditionally available only in application-specific digital signal processors, are now commonly available in general-purpose RISC processors. Several manufacturers have also added single-instruction multiple-data architecture extensions to their general-purpose processors, which are intended to enhance performance in multimedia applications. One such example is the Visual Instruction Set (VIS) in the Sun Microsystems [8] UltraSPARC processor.

Real-time sonar beamforming algorithms can require on the order of billions of multiply-accumulates per second, and therefore have traditionally been implemented in custom hardware. Native signal processing on modern multiprocessor workstations make a real-time implementation with commodity hardware possible, at a fraction of the development and manufacturing costs of a custom hardware solution. Modern workstation operating systems can provide fixed-priority scheduling for critical real-time systems without the traditional need for a proprietary real-time operating system.

The objective of this research is to develop, optimize, and evaluate the performance of sonar beamforming kernel routines for execution on Sun UltraSPARC-II processors, and to attempt to obtain maximum performance. The kernels evaluated implement a high-resolution multi-fan digital interpolation beamfomer which combines the outputs of vertical and horizontal sensor elements to image an underwater environment in three dimensions. The multi-fan vertical beamforming kernel requires integer-to-float conversion and multiple dot products, and is implemented with VIS. The horizontal beamforming kernel performs index lookup and digital interpolation, and is implemented in single-precision floating-point format.

Even with processor NSP performance advances, access to high-latency main memory causes the processor to stall. The stream-oriented nature of real-time signal processing algorithms amplifies this problem because memory I/O is high. Memory latency hiding techniques [2], including loop unrolling, software pipelining, and software prefetching [3] can be used to help alleviate this bottleneck. Because these beamforming kernels involve large, continuous streams of data, memory latency hiding techniques will be necessary for maximum performance. Therefore, an additional goal of this research is to examine these techniques as they apply to signal processing kernels.

Section 2 introduces the time-domain sonar beamforming algorithms implemented for this research. Section 3 briefly describes the UltraSPARC Visual Instruction Set (VIS). Section 4 addresses memory latency hiding techniques, including software prefetching. Section 5 discusses methodology and tool usage. Section 6 presents our results and analysis. Section 7 concludes this document.

2. Sonar beamforming

High-resolution sonars generally consist of an array of underwater sensors along with a *beam-former* to determine from which direction a sound is coming. The sensor element outputs must be combined to form multiple narrow beams, each of which "looks" in a single direction and is insensitive to sound in neighboring directions.

Time-domain beamforming is realized by weighting, delaying, and summing the outputs of an array of transducers. The beamforming time delays are determined by geometrically projecting the elements of the sensor array onto a line that is perpendicular to the "pointing angle" for the desired beam. This is demonstrated in Fig. 1 with a semi-circular array of 80 elements, for a beam pointing 20˚ off axis.

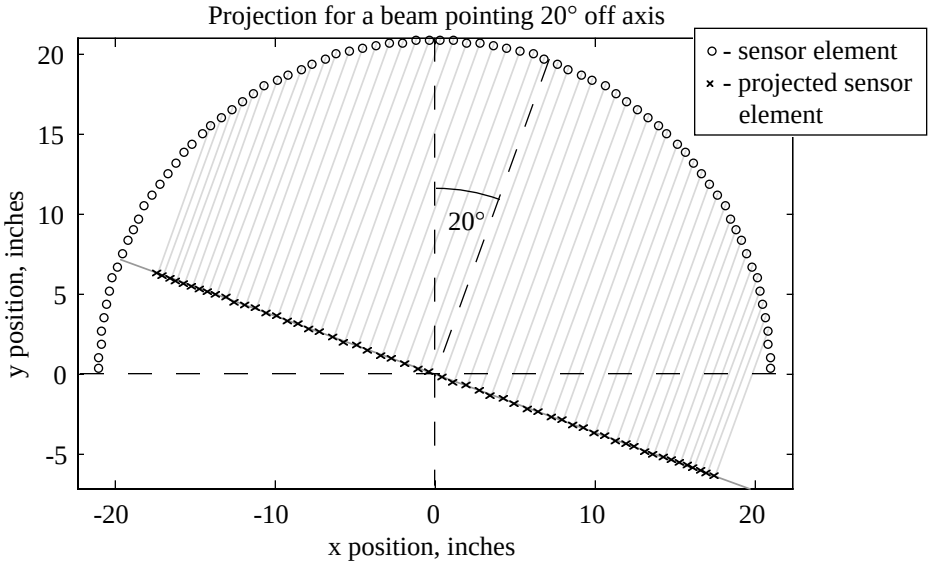

Figure 2.1: Projection of sensor elements from a semi-circular array.

The distance from each physical element location to the perpendicular line (divided by the speed of sound) is the necessary time delay for the corresponding element. Note that just over 50 of the elements have been projected, and the remaining elements have been left out. Although the remaining elements could be used in the calculation, their response in the direction of interest is relatively small for this geometry, and they would merely add noise. Leaving these elements out would also substantially reduce computation.

2.1. Digital interpolation beamforming

For digital implementations, the beamforming time delays must be quantized, which perturbs the beam response. The time delay resolution required for beam steering is much greater than the sampling interval required to preserve the frequency content of the signal [4]. Digital interpolation beamforming is a classic technique which avoids sampling at this higher rate by using interpolation of the receiver signals to achieve more precise time delay resolution, thus reducing the quantization error at a cost of more computation.

For this paper, three-dimensional beamforming is decomposed into horizontal and vertical components, and the horizontal beamformer uses digital interpolation beamforming. The digital interpolation beamforming algorithm can be modeled as a sparse gather and a dot product for each sample of each beam output. We wish to form 61 beams from 80 elements, using on average 50 staves (logical horizontal elements) per beam with two-point time-delay interpolation. For each of the 50 staves in a beam, we use an index to gather two consecutive time samples, and perform weighting and interpolation with two multiply-accumulate operations. For each beam sample, this requires on average 50 index lookups and 100 MACs. Figure 2.2 shows the coefficients for a single beam in a manner that conveys both the indices (as position) and weighting value (where white is zero and black is unity). Note that the shape of the array is clearly visible in the coefficients.

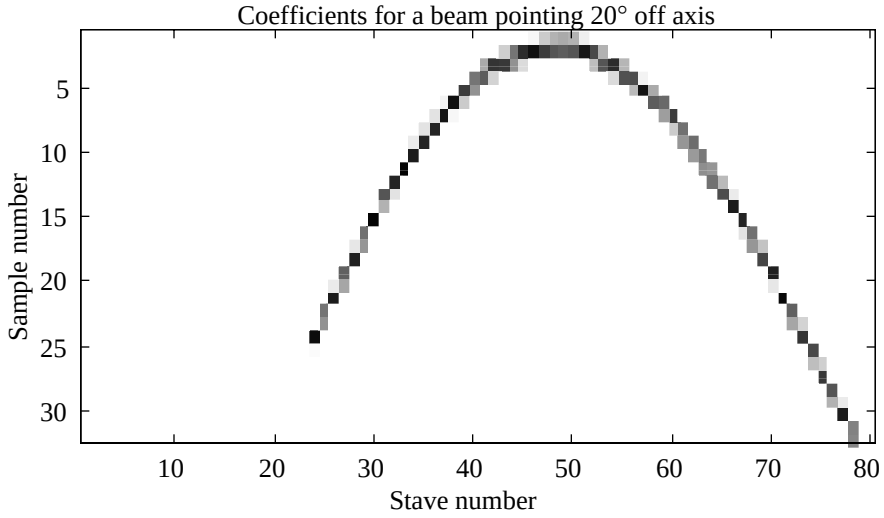

Figure 2.2: Coefficients for a single sample of a single beam.

To compute one sample of all 61 beams, we must perform approximately 3000 index lookups and 6100 MACs. The coefficient size (using 32-bit integer indices and 32-bit floating-point weights) is approximately 36 Kbytes, and the incoming sample data area (in 32-bit floating-point) is approximately 12 Kbytes.

2.2. Vertical Beamforming

In order to image in three dimensions, there are multiple vertical transducers for every horizontal element. Before horizontal (interpolation) beamforming can be performed, vertical beamforming is used to group each vertical sensor column into a stave, or logical horizontal element. Figure 2.3 illustrates this operation.

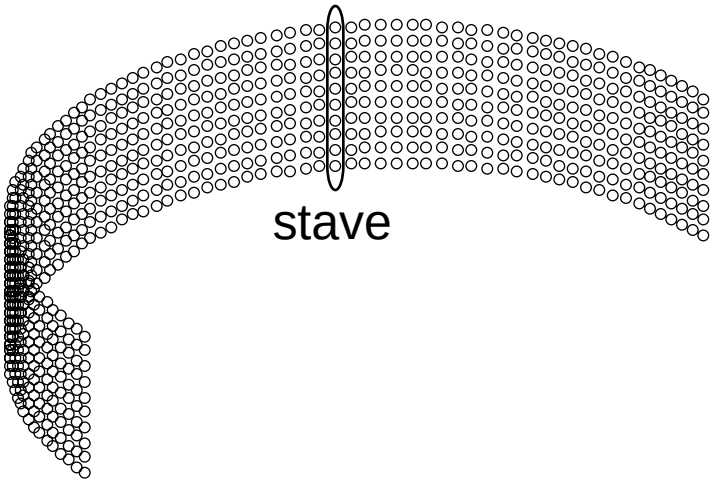

Figure 2.3: Vertical sensor columns are grouped into logical staves.

Vertical beamforming is relatively simple – no time delay or interpolation is required, and staves are formed as a simple dot product of the associated elements. To form 3 sets of staves with 10 vertical elements, 30 MACs are required. To form one sample for the entire array (80 staves), 2400 MACs are required.

Samples at the input of the vertical beamformer are in integer format, as they come directly from a digitizing circuit in the sensor array. Samples at the output of the vertical beamformer must be in floating-point format for the subsequent horizontal beamformer. This means that at some point during computation, integer-to-float conversion must be performed, and that the MACs may be performed in either integer or floating-point format. The vertical beamformer is an ideal candidate for the Visual Instruction Set, which performs integer arithmetic.

3. The Visual Instruction Set

Sun's UltraSPARC processor includes a set of instructions called the Visual Instruction Set (VIS), which is specifically optimized for video and image processing [5]. VIS treats a 64-bit register as 2, 4, or 8 partitioned data words, and performs operations on multiple words with a single instruction. Although not specifically designed to support 1-D signal processing, the UltraSPARC with VIS is an attractive target that should allow substantial performance on fixed-point algorithms [6].

The goal of VIS is to speed up signal processing kernels. It includes several partitioned fixed-point data types: 4x8-bit or 2x16-bit words in a 32-bit register, and 8x8-bit, 4x16-bit, or 2x32-bit words in a 64-bit register. VIS adds over 50 new CPU instructions, which include basic arithmetic and logic, packing and unpacking partitioned data types, alignment, data conversion, and others. Sun has developed libraries that allow VIS instructions to be called as inline functions from C.

For this paper, we require the highest precision (and slowest) of the VIS modes for the vertical beamformer – signed 16-bit by 16-bit multiplies with a signed 32-bit accumulator. VIS does not directly implement a 16-bit by 16-bit multiply, but provides two different 8-bit by 16-bit multiplies (fmuld8sux16 and fmuld8ulx16) which gives the desired operation when their results are added. In this mode, only two partitioned operations are possible, as two 32-bit results consume an entire 64-bit register. Two partitioned 16x16->32 MACs can be performed with two 8x16 multiplies and two 32-bit adds. Because the multiplies and the adds can be executed in parallel, this causes the peak performance for this mode to be 1 operation per clock (where a MAC is two operations). In order to approach the peak attainable performance for the processor, we must employ memory latency hiding techniques.

4. Memory latency hiding techniques

As processors become faster and memories grow larger, the latency for accesses to global memory will increase. To reduce this latency, systems offer cache memories which can be accessed very rapidly, but have limited storage capacity. Enormous benefits can be achieved by arranging computation to reuse cached data, but this approach still must pay the initial cache loading penalty. Furthermore, some algorithms cannot easily be arranged to take advantage of the sequential access model of caches, and must pay the penalty for many cache misses. Software

techniques which can help to hide these memory access delays include loop unrolling, software pipelining, and software prefetching.

4.1. Loop unrolling

Loop unrolling is a technique to enlarge a program's basic blocks – multiple copies of a loop are combined to form a new, larger loop. The instructions in this new loop can be carefully rescheduled to improve performance (in addition to the reduced looping overhead). By increasing the time between the data request and data consumption, the memory latency can be overlapped with useful computations.

Loop unrolling is also important for providing additional instruction-level parallelism. On many architectures (including the UltraSPARC) which provide single-cycle MACs, the multiply and add operations must not have any data dependency. Loop unrolling can relieve data dependencies, allowing multiple independent operations in a single cycle. Loop unrolling has few risks and no overhead associated with it that may nullify its benefits, as long as sufficient registers to unroll the loop are available. Loop unrolling is a commonly used optimization technique in modern compilers.

4.2. Software pipelining

Software pipelining is a technique in which memory accesses and computations are overlapped from different iterations in a program loop. This technique has been shown to provide performance gains, especially in low-latency environments [2]. Unfortunately, this technique increases the number of registers required and the register lifetimes because a register has to be associated with preloaded data. This technique is more complex than loop unrolling for a compiler to implement. Loop unrolling with instruction rescheduling can be considered a form of software pipelining.

4.3. Software prefetching

In the software prefetching [3] technique, the processor instruction set is includes a nonblocking *prefetch* instruction that causes data at a specified memory address to be brought into the cache. These prefetch instructions can be issued at some time prior to when the data is needed, so that the memory latency can be overlapped with other useful computation. When the data is actually needed and a load instruction is issued, the data is already cached and can be quickly accessed.

Many programs have memory access patterns which are highly predictable, but not sequential. Most of these patterns can be determined at compile time, allowing the compiler to manage prefetching effectively. Software prefetching consumes some overhead – it requires additional instructions to calculate the effective address and to issue the cache load, and consumes additional cache space. Unlike loop unrolling, prefetching must be used correctly so that the benefits are not nullified. On high-performance machines that can issue multiple instructions per cycle, this technique should be particularly useful because the prefetch instructions could be executed for free.

One of the enhancements to the UltraSPARC-II architecture is the implementation of prefetch instructions [7], which were implemented in the spirit of the literature [3]. In this paper we wish to evaluate the performance gains achievable with the inclusion of software prefetching.

5. Methodology

Our ultimate goal is to achieve maximum performance for the horizontal and vertical beamforming kernels, measured as execution time, using every means at our disposal. Other statistics gathered (such as load stalls or cache performance) provide useful insights into kernel performance, but are only of secondary importance. In pursuit of this goal, several different tools were utilized.

5.1. The SPARCompiler5.0 Developer Release

The SPARCompiler is Sun's compiler for their own SPARC processor. The SPARCompiler is generally regarded as the best optimizing compiler for the SPARC, and is used by Sun to compile the SPEC benchmarks. A new major release (5.0) of the SPARCompiler is now available to developers, but has not been fully released at the time of this writing. This new version of the compiler includes ANSI C++ support, 64-bit support on the UltraSPARC-II for the new Solaris 7 (or 2.7), and several optimization and performance enhancements. One eagerly awaited performance enhancement is the ability to issue prefetch instructions with the `-xprefetch=yes` compiler option.

For programming with VIS, the SPARCompiler includes inline assembly macros which are called like a function in C, and are optimized to a single opcode. For this project, similar macros were written for prefetch and fitos (which converts a 32-bit integer to a 32-bit floating-point number), so that these instructions can be directly called from a C program.

The SPARCompiler5.0 Developer Release is used to compile all beamforming kernels. Level 5 compiler optimization is turned on at all times, but the profiling feedback optimization was not used. This means that for all results presented in this paper, the compiler is attempting to perform its own optimizations in addition to the loop unrolling (and other such optimizations) in the source code.

5.2. The Shade analyzer pficount

Shade [9] is a performance analysis tool from Sun which can perform instruction set simulation, trace generation, and custom trace analysis. It is useful for obtaining detailed, dynamic, instruction-level information about a program. For this project, a Shade analyzer called `pficount` was written which dynamically counts the number of prefetch instructions in a program (absolute count and percentage of all instructions). This tool was verified by writing a program with a known number of prefetch instructions (using the inline assembly macro), and with the Shade tool `icount`. This tool can be used to quickly determine if, when, and how many prefetch instructions in a program, and whether the SPARCompiler is generating prefetch instructions as advertised.

5.3. INCAS

INCAS (It's a Near Cycle Accurate Simulator) [10] is Sun's near cycle accurate model of the UltraSPARC-I processor. It offers a convenient way to count program execution cycles and remove pipeline stalls for code optimization at the assembly language level. Although INCAS models the UltraSPARC-I instead of the UltraSPARC-II (which has larger caches and implements the prefetch instruction), it is very useful for code optimization. Drawbacks of INCAS are that it is very computationally intensive (and therefore slow for large benchmarks), and that it requires examining code at the assembly language level. By pre-warming the cache, using small benchmarks, and assuming perfect prefetching, very insightful kernel optimization clues can be obtained from this tool.

5.4. Performance instrumentation counters

Many processors (including the UltraSPARC) have hardware performance counters which can count events such as instructions, load stalls, or cache hits. The `perf-monitor` [11] tool allows user access to these features on the UltraSPARC-II. This tool consists of a loadable kernel module that accesses the performance instrumentation counters (PIC) and a configurable application tool which can measure the countable performance parameters of an executable (similarly to the Unix

`time` tool). It is also possible to access the PIC module directly so that only the NSP kernel performance is measured. This tool is almost completely non-invasive and non-sanitized, and gives actual performance results at real time.

5.5. Measuring execution time

Because execution time is our primary indication of performance, care must be taken in its measure. For this research, benchmarks are performed on a Sun Ultra Enterprise 4000 series workstation with eight 336 MHz UltraSPARC-II processors, 2 GB of RAM, and Solaris 2.6 as the operating system. Although eight processors are present, kernel performance benchmarks are executed on only one processor. This server class of workstation, with a large number of processors and high memory latency, is a realistic target for computationally intensive beamforming algorithms [12]. Fortunately, software prefetching has been shown to perform well in high memory latency systems [2].

Execution time is calculated as the average of ten trials for a reasonably sized benchmark. Execution is performed with all memory allocated and locked, with real-time scheduling priority. This prevents slowdown due to virtual memory page faults and preemption by other processes. Benchmarks are performed when the machine is not heavily loaded, to prevent excessive unrelated memory traffic. Scaling performance is purposely ignored here, but addressed in [12]. These precautions prevent large time variations across multiple trials; typical standard deviations are below 0.5% of the mean.

5.6. MFLOPS and MIOPS

Digital signal processing algorithms are commonly specified and measured in millions of floating-point operations per second (MFLOPS), and sonar beamforming is no exception. For the algorithms provided in Section 2, each multiply and each add counts as an operation. If we know the number of samples we are calculating, the number of operations per sample, and the overall execution time, we can calculate the useful MFLOPS as a reasonable number for comparison. An additional advantage to using MFLOPS is that we can compare our performance to the known peak MFLOPS for the processor. To reiterate, although our results are presented as MFLOPS, these numbers are computed from the measured execution time (and the known number of operations for the algorithm).

In order to make a fair comparison, we use MIOPS (millions of integer operations per second) when the algorithm is implemented in integer math (as in the VIS vertical beamformer). This is exactly the same measure as MFLOPS, but the math was performed using integers instead of floating-point. It is again computed from execution time and the known number of operations for the algorithm. This is *not* MIPS – looping and addressing additions are not counted – only the same operations that would be counted for MFLOPS. For comparison, IPC (instructions per cycle) is also presented in select locations.

6. Results and analysis

Level 5 optimization for the SPARCompiler5.0 Developer Release was used, with prefetching turned on, and profiling feedback optimization turned off. Despite extensive attempts with compiler flags, we could never get the SPARCompiler to issue automatically generated prefetch instructions (as measured with the `pficount` tool). For all results presented, each evaluated optimization case (such as loop unrolling) is implemented in the source code, with the compiler attempting to perform its own subsequent optimizations. Results are presented as useful MFLOPS (or MIOPS), which are calculated from the measured execution time and the number of operations in the algorithm.

6.1. Horizontal beamforming kernel performance

The horizontal beamforming kernel performs approximately 3000 index lookups and 6100 floating-point MACs per sample. This benchmark calculates 65536 samples, requiring about 804 million floating-point operations (index lookups are not counted). Fig. 6.1 shows performance results vs. unrolling the outer loop (by sample) in the source code. The maximum performance obtained was 444.3 MFLOPS, executing in 1.81 seconds per trial. At this maximum, the processor is computing 1.32 FLOPs per cycle (where the peak is 2), and issuing 2.19 instructions per cycle (measured directly with the perf-monitor PIC module).

This experiment clearly demonstrates the positive effect of loop unrolling. Unrolling the outer loop has the additional advantage of reducing the number of loads. Computing a single point in the inner loop requires 5 loads and 4 FLOPs. When unrolling by sample, each subsequent iteration requires only 1 load for 4 more FLOPs – the necessary values are already in registers. Continuing to unroll until all registers are exhausted gives the best performance.

This experiment also indicates a benefit for reducing the problem size to fit in internal level 1 cache (the UltraSPARC D-Cache). Calculation of each sample requires approximately 36 Kbytes for coefficients and 12 Kbytes of data, but the cache is only 16 Kbytes. By making multiple passes through the data and calculating only a subset of the beams at each pass (6 were used), we reduce the problem size. If we keep the data used for all passes small enough to fit in external level 2 cache (the UltraSPARC E-Cache), we pay the penalty for access to main memory on the first pass, but subsequent passes can fetch from the faster level 2 cache. For the 4 MB external cache used in this experiment, this is several thousand samples.

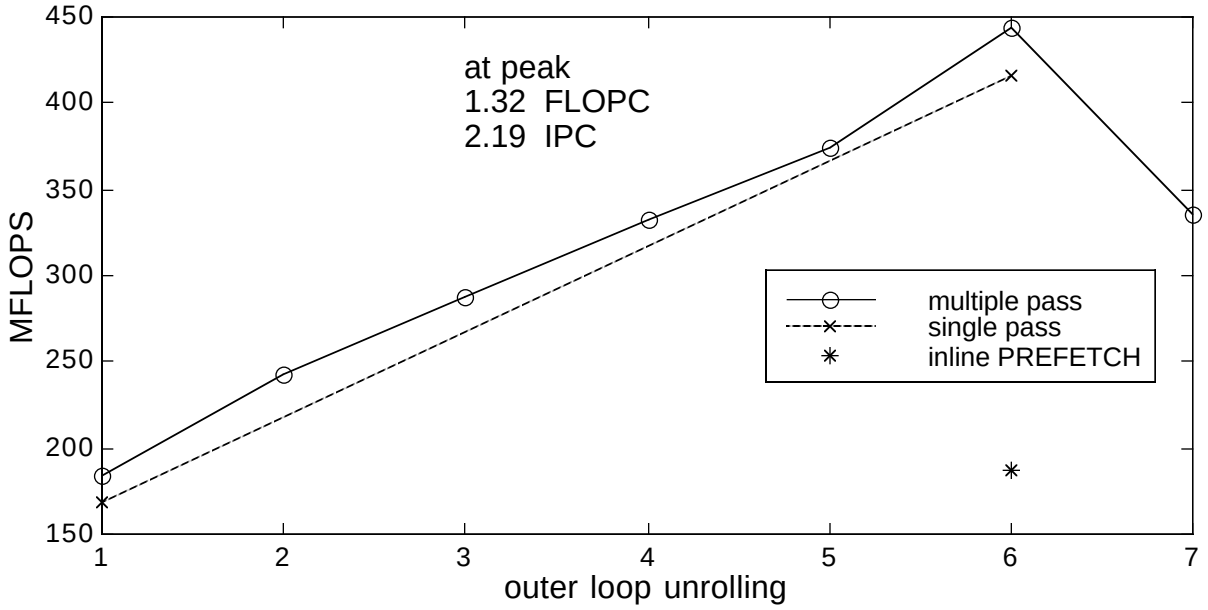

Figure 6.1: Horizontal beamformer kernel performance.

The improvement for making multiple passes is helpful but not outstanding. At the peak, a 6.7% improvement is measured. Other statistics measured with perf-monitor include a 4.1% improvement in D-Cache read hit-ratio (from 88.7% to 92.8%) and a 4.9% reduction in the percentage of cycles with a pipeline stall due to outstanding memory loads (from 32.2% to 27.3%).

Note that neither of these optimizations are something that a compiler will ever be able to perform because they require too much high-level knowledge of the algorithm. However, the compiler optimizes very well when exposed to this much parallelism in a basic block, and performs its own inner loop unrolling. This loop is in fairly high-level C, which was written especially to suit the compiler. By generating assembly output, one can determine how well the compiler "understands" the loop to optimize it. Experience shows that the compiler performs best on very simple

loops with large expressions, using arrays and indices rather than pointers (which would be incremented at the end of each loop). The form of these loops is surprisingly fragile – attempts at low-level optimizations that the compiler can perform (like common sub-expression elimination) are best left to the compiler for best results.

Software prefetching is also better left to the compiler, but could not be made to work. By inserting inline function calls from C, we can issue prefetch instructions. In an attempt to evaluate prefetching with the horizontal beamformer, a single prefetch instruction was inserted into the inner loop of the best-performing code. As the asterisk in Fig. 6.1 shows, this gave very poor results – a slowdown of 2.37. This clearly demonstrates the fragility of the compiler loops. Despite the fact that only one assembly opcode has been added, the compiler no longer "recognizes" the loop, and no longer performs its aggressive optimization on it (as verified by examination of the assembly output).

6.2. Vertical beamforming kernel performance

The horizontal beamforming kernel performs 2400 MAC operations (integer or floating-point) per sample. For this benchmark 128K samples are computed, requiring about 629 million integer or floating-point operations (see Section 5.6). Fig. 6.2 shows performance results in MFLOPS or MIOPS for several different cases. The maximum performance obtained was 313.3 MIOPS, executing in 2.008 seconds per trial. At this maximum, the processor is computing signed integer 16-bit multiplies (with 32-bit results) and signed 32-bit accumulates at a rate of 0.93 IOPs per cycle (where the peak is 1), and issuing 1.41 instructions per cycle (measured directly with the perf-monitor PIC module). Note that a 16x16->32 MAC counts as only two operations, despite the fact that it requires four VIS instructions (see Section 3) – comparing it otherwise would give VIS and unrealistic (and unreasonable) speedup advantage.

Because incoming data is in integer format and outgoing data is in floating-point, the vertical beamformer calculations can be performed in either integer or floating-point, with format conversion after or before the calculation. Although the UltraSPARC can perform two FLOPs per cycle and only one 16x16->32 operation per cycle (when averaged) with VIS, data conversion makes the VIS version considerably faster. In Fig. 6.2, floating-point implementations are grey and integer implementations are white.

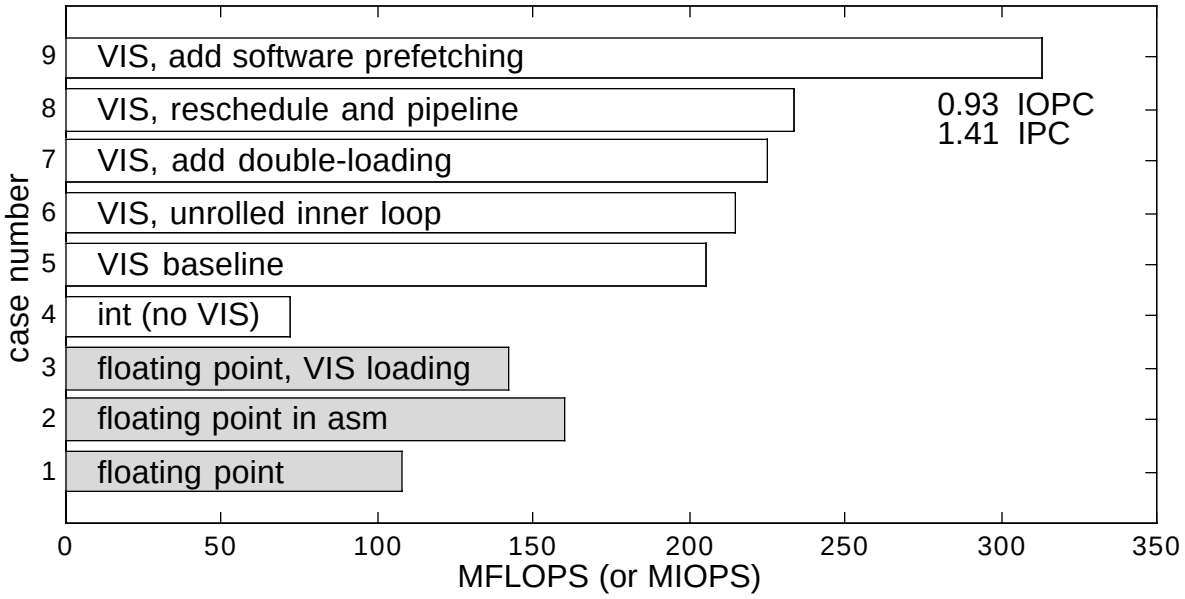

Figure 6.2: Vertical beamformer kernel performance.

Without VIS, converting a 16-bit integer to a single-precision floating-point number requires about 6 instructions (load 16-bit, sign-extend, store 32-bit, load 32-bit into float register, convert to float) because sign-extension is not possible in a floating-point register. The first two cases use this conversion – 10 conversions are required to perform 30 MACs! Hand-coding in assembly gives a 49% speedup, but the conversion overhead is horrible. With VIS, we can convert a 16-bit integer to a single-precision floating-point, but it is not signed. By converting to binary-offset (unsigned) arithmetic and subtracting the offset after calculation, we can perform the conversion in 3 instructions (load 16-bit into float register, xor the sign bit, convert to float). Case 3 uses this method, which performs better than the non-VIS conversion, case 1. No assembly optimization was performed for this version, but expected performance would be considerably better than case 2. Still, conversion overhead is unacceptable.

For comparison, case 4 implements the vertical beamformer in integer math, without VIS. Performance is poor, as expected. The remaining cases use VIS. Here, conversion overhead is small – the calculation result exists as a 32-bit signed integer in a floating-point register, which can be converted with a single instruction (fitos) before storing. Even the baseline VIS case outperforms the best floating-point case by 28%.

Programming in VIS is very close to programming in assembly language – each line of C translates into only a few (or even one) instruction. In contrast, each line of code in the horizontal beamformer translates into dozens of instructions. For this low level of coding the compiler is poor at optimization (especially instruction scheduling), so lines of code must be very carefully placed. INCAS is necessary here for eliminating pipeline stalls, to achieve the best performance.

By unrolling, rescheduling, and pipelining the source code, progressively better results can be obtained. Because the ultimate goal is maximum performance, all of the various combinations were not tested. In the UltraSPARC, two adjacent aligned 32-bit numbers can be loaded as a one 64-bit number and operated upon independently. This reduces the number of loads, and is tested in case 7. Overall results show that for this algorithm, VIS offers a 46% performance boost. Software prefetching to hide memory latency (case 9) is addressed in the following section.

6.3. Prefetch performance

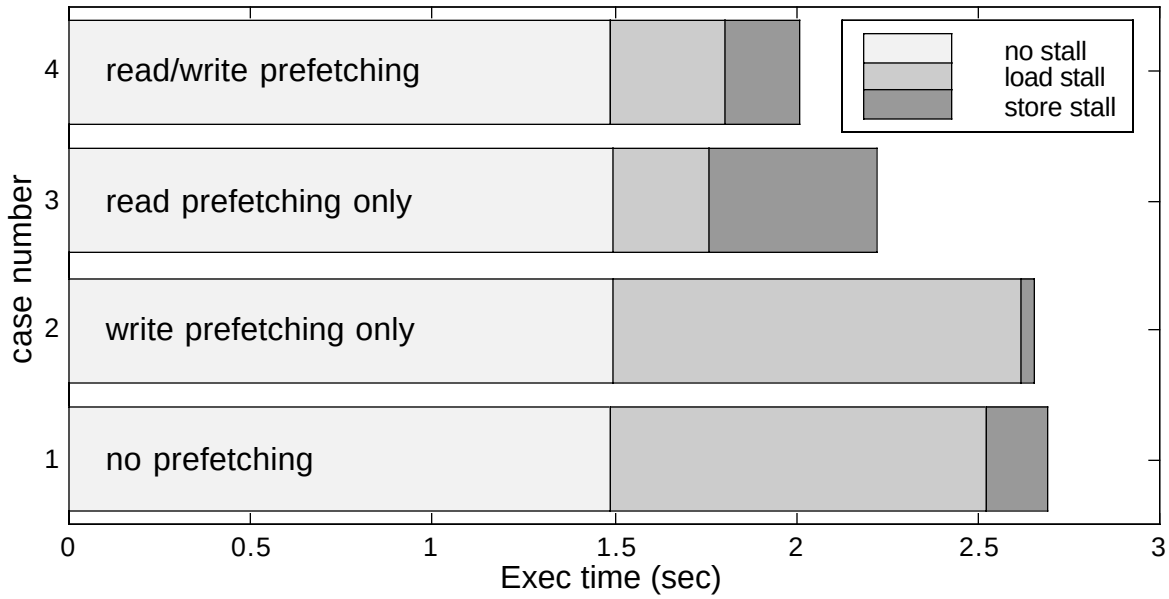

Figure 6.3: Prefetch performance in the vertical beamformer.

For the best performing VIS vertical beamformer routine (unrolled with rescheduling and pipelining), we examine the effects of software prefetching. It is interesting to note that level 1 cache (D-Cache) statistics do not change with the addition of prefetch instructions – the performance counters do not distinguish between a cache miss due to a load or store and a cache miss due to a prefetch. However, the number of load (Load_use) or store (Dispatch0_storeBuf) stalls

change dramatically. As shown in Fig. 6.3, software prefetching gives the vertical beamformer an additional 34% performance gain.

This figure shows execution time, categorized as load stall time, store stall time, or no stall (execution) time. (All other stalls combined are less than 1%). Clearly, the useful execution time remains consistent throughout the trials, prefetching write addresses reduces store stalls, and prefetching read addresses reduces load stalls. The figure show that further reduction of stalls may still be possible.

7. Conclusion

Native signal processing on the UltraSPARC-II can give very good results. We achieved 444.3 MFLOPS with the horizontal beamformer, and 313.3 MIOPS with the vertical beamformer, on a 336 MHz processor. Loop unrolling as a means for memory latency hiding and increased instruction-level parallelism provides excellent results in nearly all cases. Clearly, the use of VIS and software prefetching can result in excellent performance gains (95% in the vertical beamformer).

The new SPARCompiler5.0 gets a mixed review. It gives excellent optimization performance for large, simple loops with large expressions (as in the horizontal beamformer). However, the lack of automatically generated prefetch instructions is a major setback. These instructions can be included with inline assembly macros, but this reduces the effectiveness of the optimizer.

Low-level programming with VIS or prefetch instructions (as in the vertical beamformer) is difficult and time consuming, and similar to assembly or DSP code. For this style of code, the compiler instruction scheduling is poor, and a tool like INCAS is required to get good results. While we are still programming in a high level language, we must be concerned with instruction-level detail for good optimization. However, excellent performance gains can be achieved if the expense of hand optimization is justifiable.

Acknowledgments

G. Allen was supported by the Independent Research and Development Program at Applied Research Laboratories: The University of Texas at Austin.

References

[1] P. Lapsley, "NSP Shows Promise on Pentium, PowerPC," *MicroProcessor Report*, 1995.

[2] L. John, V. Reddy, P. Hulina, and L. Coraor, "A Comparative Evaluation of Software Techniques to Hide Memory Latencies." *Proceedings of the 28th Hawaii International Conference on System Sciences*, Vol. I, pp. 229-238, Jan. 1995.

[3] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching." *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 40-52, Apr. 1991.

[4] R. G. Pridham and R. A. Mucci, "A Novel Approach to Digital Beamforming." *Journal Acoustical Society of America*, vol. 63, no. 2, pp. 425-434, Feb. 1978.

[5] *Visual Instruction Set User's Guide*, Sun Microsystems, 1995.

[6] W. Chen, H. Reekie, S. Bhave, and E. Lee, "Native Signal Processing on the UltraSPARC in the Ptolemy Environment." *Proceedings of the 1996 30th Asilomar Conference on Signals, Systems & Computers*, pp. 1368-1372, Nov. 1996.

[7] D. L. Weaver and T. Germond, eds. *The SPARC Architecture Manual, Version 9*, Prentice-Hall, Inc., 1994.

[8] The Sun Microsystems Web Page: `http://www.sun.com/`

[9] *The Shade User's Manual*, Sun Microsystems, 1998.

[10] *Incas User's Guide 2.0*, Sun Microsystems, 1993.

[11] The Perf-monitor Web Page: `http://www.sics.se/~mch/perf-monitor/`

[12] G. Allen, *Real-Time Sonar Beamforming on a Symmetric Multiprocessing UNIX Workstation Using Process Networks and POSIX Pthreads*. Master's Report, Dept. of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712-1084, http://www.ece.utexas.edu/~allen/MSReport/, Aug. 1998.