Vectorizing SPECint95

Krste Asanović
Computer Science Division
University of California at Berkeley
Berkeley, CA94720
krste@cs.berkeley.edu

Abstract

The SPECint95 benchmark suite contains compute-intensive integer codes that are generally regarded as non-vectorizable. This study reveals, however, that half of the benchmarks can be accelerated significantly with vector execution, though this requires minor modifications to the source code in some cases. For the T0 vector microprocessor, a geometric mean improvement of 1.32 is obtained across all eight benchmarks, with individual speedups in the range 1.16–4.5. Profiling information reveals that the vector unit provides large speedups but only on limited portions of the runtime, whereas superscalar processors provide more modest speedups across the entire runtime. This result implies that vectors combined with superscalar instruction issue will yield much larger speedups than if either technique is used in isolation. For example, the vector unit from T0 would only add about 7% to the area of the R10000 superscalar microprocessor but would increase its SPECint95 performance by around 28%. These results demonstrate that a vector unit can improve cost/performance even for codes with low levels of vectorization.

1 Introduction

Increases in the level of scalar instruction-level parallelism (ILP) supported by microprocessor microarchitectures have been accompanied by dramatic increases in the die area of these scalar units, primarily because of the complexity of managing multiple scalar instructions at various stages of execution. Vector instruction set extensions provide a much simpler hardware mechanism for supporting high degrees of parallelism, provided computations can be cast into a data parallel form. Vectors have proven successful at improving the performance of supercomputers running scientific and engineering applications, but there has been little research applying vectors to other application areas. The need for greater performance on multimedia applications has renewed interest in such data parallel instruction sets, with several manufacturers introducing data parallel extensions to their instruction sets [PW96, TONH96, Lee96] and several research groups investigating vector microprocessor designs [WAK$^+$96, KPP$^+$97, LD97, Esp97]. It is therefore plausible that future microprocessors will include small but powerful vector units to accelerate both new multimedia and human-machine interface applications as well as traditional scientific and engineering tasks. Currently, only supercomputers have vector units and so few non-supercomputer applications are written with vectorization in mind. If vector units become commonplace on low-cost systems and provide the fastest mode of execution, compiler writers and programmers will have an incentive to vectorize a wider range of tasks.

The SPEC95 benchmarks [Corb] have become popular both to measure performance of commercial processors and to provide the workload for compiler and architecture research studies. The SPEC95 benchmarks are divided into SPECfp95, which contains floating-point codes, and SPECint95, which contains compute-intensive integer codes. While SPECfp95 contains many programs originally developed for vector supercomputers and which are known to be highly vectorizable, the SPECint95 benchmarks are generally regarded as non-vectorizable. This paper reports on a study which reveals that half of the SPECint95 codes, including some described as "non-vectorizable" by the SPEC documentation, can be accelerated significantly using vector execution, though this occasionally requires some minor modifications to the source code.

2 Methodology

From Amdahl's law [Amd67], we know that vector speedup is given by

$$\frac{T_s}{T_v} = \frac{1}{(1-f) + f/V}$$

where T_s is the execution time of the program running in scalar mode, T_v is the execution time for the program running in vector mode, f is the fraction enhanced by vector execution (the *vector coverage*), and V is the vector speedup for that fraction. More generally, programs have several different portions of their runtime that can be accelerated to differing degrees, which we can express by rewriting Amdahl's law as

$$\frac{T_s}{T_v} = \frac{1}{(1-\sum_i f_i) + \sum_i (f_i/V_i)}$$

where V_i is the vector speedup for fraction f_i of the runtime.

The approach taken in this study has two main steps. The first step is to profile the SPECint95 codes running on conventional superscalar microprocessor-based systems to identify routines that are both time-consuming and vectorizable. Two workstations were used for the profiling measurements, a Sun Ultra-1/170 workstation running Solaris 2.5.1 and an Intel Pentium-II workstation running NetBSD1.3. The Sun machine has a 167 MHz UltraSPARC-I processor [CDd+95], with in-order issue of up to 4 instructions per cycle. The Sun C compiler version 4.0 was used to compile the code using the flags: `-fast -xO4 -xarch=v8plus -dn`. Execution time profiles were obtained using either the `-p` compiler option with `prof`, or with high-resolution timer calls (Solaris `gethrtimer`) embedded in the application. The Pentium-II processor [Gwe97] runs at 300 MHz with out-of-order execution of up to 3 CISC instructions per cycle. The `gcc` version 2.7.2.2 compiler was used to compile the code with optimization flags: `-O3 -m486`. Execution time profiles were obtained using either the `-pg` compiler option with `gprof`, or with high-resolution timer calls (NetBSD `gettimeofday`).

The second step is to port the codes to run on the T0 vector microprocessor [WAK+96] described below in the next section. The SPECint95 codes were not written with vectorization in mind, and in some cases, have been highly tuned for scalar execution. This artificially limits the potential for vectorization, and so minor changes were allowed within source code modules provided there were no changes to global data structures or program behavior. The intent is to model how these types of code would be written and tuned for future microprocessors with vector units, rather than to determine the effectiveness of automatic vectorization of the unmodified SPECint95 sources. The scalar code was also modified and optimized to allow a fairer comparison.

Execution is profiled both before and after vectorization on T0. These timings give values for the vector speedup, V_i, as well as the fraction of runtime, f_i, for each piece of vectorizable code. The workstation systems are also profiled to give values for f_i.

3 The T0 Vector Microprocessor

T0 is a research prototype containing a MIPS-compatible RISC scalar unit extended with a vector coprocessor. The scalar unit is very similar to the MIPS R3000 [Kan89], except that it includes the MIPS-II instruction set extensions and has a fully-pipelined 3 cycle load latency with 2 interlocked load-delay slots. The processor has a small 1 KB instruction cache but no data cache. Scalar load and store instructions access off-chip SRAM main memory directly.

The vector coprocessor is tightly coupled to the scalar unit and includes 15 general-purpose vector registers (plus a zero register) each holding 32 elements of 32 bits, two vector arithmetic functional units, and one vector memory unit. Both vector arithmetic functional units support a full complement of 32-bit integer arithmetic, logical, and shift operations, together with support for scaled, rounded, and saturated fixed-point arithmetic. One of the vector arithmetic functional units also contains a 16-bit by 16-bit multiplier producing 32-bit results. Vectorized conditional execution is supported via vector conditional move instructions. Both vector arithmetic functional units contain 8 parallel pipelines (*lanes*) and can produce up to 8 results per clock cycle. The vector memory functional unit loads and stores vector registers to and from memory, supporting unit-stride, constant-stride, and indexed (scatter/gather) accesses. The path to off-chip memory has a 128-bit wide data bus and a single address port. Unit-stride vector loads and stores transfer up to four 32-bit operands, eight 16-bit operands, or eight 8-bit operands per cycle. Non-unit strided and indexed vector loads and stores are limited by the single address port and transfer up to one element per cycle regardless of operand size. The vector coprocessor has fully pipelined instruction startup to eliminate vector startup penalties, and implements all forms of chaining to reduce inter-instruction latencies.

The Spert-II workstation accelerator [WAK$^+$96] integrates a 40 MHz T0 together with 8 MB of SRAM memory on a double-slot SBus card. The board is mounted in a Sun workstation which provides I/O facilities. There is no automatically vectorizing compiler for T0, but the `gcc` cross-compiler version 2.7.0 is available for scalar code and the `gas` cross-assembler has been extended to handle the vector coprocessor instructions. As described below, the vectorizable code was manually translated into assembly code. Timings were obtained using the on-chip cycle counter.

4 Benchmark Modifications

Three versions of each code were obtained by modifying the SPECint95 sources.

The first version contains only changes required to port the code to the Spert-II board. In some cases, the benchmark was reduced in size to fit in the small 8 MB memory. Also, a few modifications were made to port to the limited operating system environment on the board. No changes were made to the main computational routines in this version. This version is called the *original* code in the results.

The second version contains source code optimizations to improve performance when running in scalar mode to allow a fairer comparison against the hand-vectorized code. Some changes were made to work around the limitations in the `gcc` compiler used for T0 and the Pentium-II. In particular, `gcc` can only inline a function if its definition textually precedes the call site, so a few important function definitions were reordered to mimic more sophisticated inlining. As described below, some other modifications were made to each benchmark to improve scalar performance, e.g., explicit loop unrolling in `m88ksim`. Also, any vectorization changes that improved scalar performance were also included. This version is called the *scalar-optimized* code in the results.

The final version is locally restructured to allow vectorization, and vectorizable routines are manually translated into assembly code. This version also incorporates any beneficial scalar optimizations from the scalar-optimized code and is called the *vectorized* code in the results.

The vectorized codes are linked with the T0 standard C library, which includes vectorized versions of the `memcpy`, `memset`, `strlen`, `strcpy`, and `strcmp` routines. The original and scalar-optimized codes are linked with an alternative C library containing hand-tuned scalar assembler versions of these routines.

The following subsections describe the modifications made to each benchmark in more detail.

4.1 `m88ksim`

The `m88ksim` benchmark times a simulator for the Motorola 88100 microprocessor. The standard reference input first loads a memory test program which is too large to fit into Spert-II memory. The simulator execution profile is not highly dependent on the simulated program, so the input was changed to run only the last portion of the benchmark which simulates a Dhrystone executable. This reduced total benchmark running time from 370 to 226 seconds on the Ultra-1/170. The standard `ctl.in` input was replaced with the following:

```
lo dhry.big
cacheoff
g
sd
```

q

Considerable time is spent in two vectorizable functions: `killtime` and `ckbrkpts`. The `killtime` routine contains two loops to update busy time status for the 32 registers and the 5 functional units in the system. The register update loop was explicitly unrolled for the scalar-optimized version. Both loops were vectorized for the vectorized version, with vector lengths 32 and 5.

The `cbrkpts` routine contains a loop to check on the 16 possible breakpoints, but this loop exits early whenever a breakpoint condition is met. The loop is vectorized by speculatively executing all 16 loop iterations, with the exit iteration determined subsequently. This speculative execution is straightforward to implement in software because the loop does not write memory and the loop bounds are compile-time constants.

4.2 `compress`

The `compress` benchmark is a modified version of the Unix `compress` command based on the LZW compression algorithm [Wel84]. To fit into the board's memory, the benchmark was changed to compress and decompress 2,000,000 characters of text. The text was generated by the SPEC-supplied harness code with arguments "`2000000 e 2231`". The benchmark was split to give separate timings for compression and decompression, with the two components referred to as `comp` and `decomp` in the results below.

The scalar-optimized version of the code was almost completely rewritten from the original to provide a clearer structure and faster code. Some of the major changes included removing references to global variables inside loops, handling exceptional conditions, such as code bit size changes and string table clear, outside of the main control flow, and using an array of structures rather than two arrays for the hashed string table to reduce address calculations and cache misses.

Profiling revealed that significant time was spent packing and unpacking codes from the input and output bitstreams. The original compress routine packs each 9- to 16-bit code into the output bitstream one code at a time. A more efficient approach is to keep a buffer of unpacked codes, and then to pack the entire buffer into the output bitstream in one step. In particular, a single check for 16-bit codes can be used to choose a specialized loop which avoids the bitfield manipulations required for shorter code lengths. This is an important optimization because 16-bit codes are the most common for long files. Similarly, the decompress routine can be accelerated by unpacking multiple codes at a time. The scalar-optimized code includes these pack/unpack optimizations.

The original decompression routine reads out strings from the string table one character at a time in reverse order into a temporary string buffer. The characters in the buffer are then copied in correct order into the output stream. The scalar-optimized version of the code puts the characters into the buffer in the correct order and so can use a `memcpy` to move characters into the output stream.

The vectorized code attains speedups by vectorizing the pack/unpack operation and also the standard `memcpy` routine. When 8 N-bit codes are packed, they occupy N bytes of storage. Vectorization is across these independent N-byte sections of packed input or output codes. A buffer of 512 codes is used, giving a vector length of 64.

The execution time for `compress` is very sensitive to code quality in the inner scalar loops. The compiler's assembler output for this inner loop was hand-optimized to overcome problems in `gcc`'s instruction-scheduling and delay-slot filling in both the scalar-optimized and vector codes for T0. In addition, the pack/unpack routines of the scalar-optimized code were hand-scheduled, and the `memcpy` call for the vector code was hand-inlined.

4.3 `ijpeg`

The `ijpeg` benchmark repeatedly performs JPEG image compression and decompression while varying various compression parameters. It is the most highly vectorizable of the SPECint95 programs.

A number of changes were made to the original scalar code to port to the Spert-II board. The `MULTIPLIER` type was changed from `int` to `short` in `jmorecfg.h` to match the interface with the T0 vector inverse DCT (Discrete Cosine Transform) routine. The `ijpeg` DCT routines have been designed to work with 16-bit integers and so this change doesn't affect the accuracy or runtime of the scalar routine. The T0 forward DCT vector routine first performs 1D DCTs down columns followed by 1D DCTs across rows, whereas the original SPEC code does rows first. Although mathematically equivalent, rounding errors cause a slight change in the coefficients and hence in test output. The original and scalar-optimized versions were modified to work the same way as the vectorized version so that all codes would produce identical output. This resulted in no change to run-time and less than 0.2% change in compressed file sizes for the test image.

The constant `JPEG_BUFFER_SIZE` was reduced to 1 MB in `spec_main.c` to reduce memory consumption by statically allocated arrays. The timings used the `penguin.ppm` reference input image reduced in resolution by a factor of two in both directions to fit into the reduced memory. The reduced size image does change the scalar execution profile slightly, and halves the vector lengths of some routines, but the runs should still be representative of typical JPEG code.

The benchmark command line arguments were:

```
-image_file penguin.ppm -compression.quality 90
-compression.optimize_coding 0 -compression.smoothing_factor 90
-difference.image 1 -difference.x_stride 10 -difference.y_stride 10
-verbose 1 -GO.findoptcomp
```

The `ijpeg` code has been well tuned for scalar execution but a few minor optimizations were added. The original code performs coefficient quantization using an integer divide, with a quick magnitude check to skip cases where the result will be zero. The quantization tables are changed rarely, and so the division can be replaced by a reciprocal multiply and arithmetic shift right. When packing bits into the output stream, the original code calls the `emit_bits` subroutine twice for non-zero AC coefficients, once for the preceding zeros symbol and again for the non-zero coefficient magnitude. These two bit strings can be combined to reduce the number of calls to `emit_bits`, though two calls must still be made if the total number of bits is greater than 24. A further optimization is possible, similar to that in `compress`, where instead of calling `emit_bits` as each symbol is generated, all symbols for a block are first buffered before making a single call to pack all the symbol bitstrings to the output stream.

The vectorized version replaces many routines with vector code. The forward and inverse 8×8 2D DCTs are vectorized across the constituent 1D 8-point DCTs. The forward DCTs are performed with vector length 512, corresponding to the test image width. The inverse DCTs are performed in small groups of 1 or 2 blocks, limiting vector lengths to 8 or 16. The vector length for inverse DCT could also have been increased up to the image width by buffering a scan line of coefficients, but this would have required more than local restructuring of the code. The coefficient quantization and dequantization is also vectorized with vector lengths of 64. The image manipulation routines `rgb_ycc_convert`, `upsample`, and `downsample`, have vector lengths proportional to image width (512), or half of the image width (256) for subsampled components.

The `encode_MCU` routine has two incarnations, one which generates Huffman coded bitstrings (`encode_one_block`) and one which just calculates the symbol frequencies for optimizing Huffman coding (`htest_one_block`). These routines need to scan the 63 AC forward DCT coefficients to find runs of zeros. A vector routine was written to scan the coefficients and return a packed bit vector. This packed bit vector can then be scanned quickly with scalar code to find the symbols for Huffman coding. The Huffman code lookup can also be vectorized, but it was found that the vector version ran at about the same speed as the scalar version on T0. The scalar code can use an early exit loop to find the coefficient magnitude, but the vector version must use multiple conditional instructions to determine the magnitudes and these consume more than half the vector lookup runtime.

4.4 `li`

The `li` benchmark is a simple Lisp interpreter. The original SPECint95 benchmark can be run unchanged on the Spert-II board. Profiling revealed that a considerable fraction of time is spent in the mark and sweep garbage collector. The only change made for the scalar-optimized code was to reorder routines in the garbage collector code to allow `gcc` to inline them.

The mark phase of the `li` garbage collector was vectorized by converting the depth-first traversal of the live storage nodes into a breadth-first traversal [AB89]. The mark routine dynamically manages a queue of pointers on the run-time stack. The queue is initialized with the root live pointers at the start of garbage collection. A stripmined loop reads pointers from the queue and overwrites it with pointers for the next level of the search tree. Any pointers that should not be followed are set to NULL. Most vector machines provide some form of vector compress instruction to pack together selected elements of a vector [HP96]. T0 lacks a vector compress instruction and so a scalar loop is used to compress out NULL pointers from the queue. The mark phase is finished when the queue is empty after the scalar compress operation. The average vector length is 80 during this phase.

The `li` interpreter allocates node storage in segments, each of which contains a contiguous array of nodes. The sweep phase was vectorized by partitioning the nodes within a segment into sections, then vectorizing over these sections. The vector code chains together the unmarked nodes within each section into a local free list. A scalar loop then stitches the multiple free lists together to give a single free list, which has the same node ordering as the original scalar code. The vector length is determined by the number of nodes allocated at one time within each segment — 1000 for the `li` benchmark.

4.5 Other SPECint95 benchmarks

The remaining SPECint95 benchmarks could not be appreciably vectorized with small local changes. Although the `perl` benchmark is dominated by complex control flow, it spends a few percent of its runtime in vectorizable standard C library functions [Corb] and so might experience a small speedup from vectorization. `perl` could not be ported to the Spert-II board due to its extensive use of operating system features.

The `go` benchmark spends significant time in loops but makes extensive use of linked list data structures, which hamper vectorization. It is possible that a substantial rewrite could make use of vectors.

The `gcc` and `vortex` benchmarks are dominated by complicated control structures with few loops and it is unlikely that even extensive restructuring could uncover significant levels of vector parallelism.

5 Results

Table 1 shows the extent of the code modifications in terms of lines of code in the original sources that were affected. Except for `compress`, only a small fraction of the source was modified for vectorization. The `compress` code is a small kernel, and was extensively rewritten for the scalar-optimized version. Although it required the largest percentage of modifications for vectorization, the absolute number of lines changed is small.

The four codes present a range of possibilities for how vector execution could be incorporated into these types of applications. The `m88ksim` and `ijpeg` codes as written would require little programmer effort to obtain the benefits of vector execution. Garbage collectors, such as that in `li`, are small but important kernels. Garbage collectors are often highly tuned by language implementers, because any speedups benefit all programs run under the language system. The `compress` code requires some local restructuring to obtain a data parallel version, but this

Benchmark	Original (LOC)	Scalar-Optimized Changes (LOC)		Vector Changes (LOC)	
m88ksim	19,915	4	(0.0%)	65	(0.3%)
compress	1,169	1,071	(100.0%)	*80	(7.5%)
ijpeg	31,211	84	(0.3%)	778	(2.5%)
li	7,597	0	(0.0%)	198	(2.6%)

Table 1: This table shows the number of lines of code in the original benchmark, together with the number of lines of original code changed for the scalar-optimized and vectorized versions. *compress vectorization changes are counted relative to the scalar-optimized version.

restructuring also improves the performance of scalar execution.

Table 2 presents the overall timings for the codes on each platform. The scalar optimizations were tuned for Spert-II, and although generally these optimizations improve performance on the other platforms, in a few cases performance is reduced. In particular, the explicit loop unrolling in m88ksim reduces performance on both the UltraSPARC and the Pentium-II. The li scalar-optimized code on the UltraSPARC was also slightly (2%) slower than the original code, even though the only modification was to reorder some routine definitions. The scalar-optimized code for compress is twice as fast as the original SPEC source code on T0, but only 15–18% faster on the workstation systems. This difference in speedup might be due to the better compiler and superscalar issue on the Sun system and the out-of-order superscalar scheduling on the Pentium-II giving better performance on the original code.

Table 3 gives a timing breakdown for the benchmarks. The scalar-optimized version was used for the scalar profiles, except for m88ksim on the workstation systems where the faster original version was used. These profiles were obtained either with statistical profiling or with embedded calls to high-resolution timers, both of which are intrusive methods that slightly affect the timing results compared to Table 2.

The vector speedup obtained on whole benchmarks varies widely, with ijpeg having the greatest speedup (4.5) confirming the suitability of vectors for these types of multimedia applications. The lowest speedup is for comp (1.07) whose runtime is dominated by scalar hash table operations. Using a combined figure of 1.16 for compress, and assuming no speedup for the non-vectorized codes, the geometric mean vector speedup for T0 across all eight SPECint95

Benchmark	Original (s)	Scalar-Optimized (s)	Vector (s)
Ultra-1/170, 167 MHz			
`m88ksim`	226.5	241.0	N/A
`compress`	1.49	1.30	N/A
`ijpeg`	37.6	36.1	N/A
`li`	403.8	412.1	N/A
Pentium-II, 300 MHz			
`m88ksim`	133.7	140.2	N/A
`compress`	1.25	1.06	N/A
`ijpeg`	24.8	24.3	N/A
`li`	209.4	191.5	N/A
Spert-II, 40 MHz			
`m88ksim`	1853.1	1831.5	1300.2
`compress`	7.43	3.75	3.23
`ijpeg`	271.5	269.7	63.8
`li`	2493.0	2279.1	1871.8

Table 2: Execution time in seconds for the three versions of each benchmark on each platform. The `compress` times are the sum of `comp` and `decomp`.

Application Routine	Ultra-1/170 167 MHz (s)	(%)	Pentium-II 300 MHz (s)	(%)	T0 Scalar 40 MHz (s)	(%)	T0 Vector 40 MHz (s)	(%)	T0 Vector Speedup
`m88ksim`									
`killtime`	66	(26)	34.8	(26.2)	457.3	(25.0)	57.6	(4.4)	7.93
`ckbrkpts`	41	(16)	15.6	(11.7)	230.1	(12.6)	101.1	(7.8)	2.28
Other	143	(57)	82.5	(62.0)	1144.1	(62.5)	1142.3	(87.8)	1.00
Total	250	(100)	132.8	(100.0)	1831.5	(100.0)	1301.0	(100.0)	1.41
`comp`									
`packcodes`	0.11	(11.0)	0.04	(4.6)	0.22	(9.6)	0.03	(1.2)	8.58
Other	0.85	(89.0)	0.81	(95.4)	2.05	(90.4)	2.12	(98.8)	0.97
Total	0.96	(100.0)	0.85	(100.0)	2.27	(100.0)	2.15	(100.0)	1.07
`decomp`									
`unpackcodes`	0.04	(12.3)	0.04	(10.5)	0.24	(16.3)	0.03	(2.4)	9.46
Other	0.30	(88.6)	0.18	(89.5)	1.23	(83.7)	1.06	(97.6)	1.16
Total	0.34	(100.0)	0.22	(100.0)	1.48	(100.0)	1.08	(100.0)	1.37
`ijpeg`									
`rgbyccconvert`	4.1	(10.4)	3.17	(13.1)	23.1	(8.7)	2.2	(3.6)	10.7
`downsample`	7.1	(18.0)	4.47	(18.5)	62.1	(23.4)	6.5	(11.0)	9.5
`forward_DCT`	12.7	(32.2)	5.95	(24.6)	73.6	(27.8)	5.1	(8.6)	14.3
`encode_MCU`	4.4	(11.2)	3.26	(13.5)	26.3	(9.9)	18.8	(31.6)	1.4
`inverse_DCT`	5.3	(13.5)	3.71	(15.3)	42.4	(16.0)	4.6	(7.7)	9.2
`upsample`	2.2	(5.6)	1.44	(6.0)	14.6	(5.5)	1.1	(1.8)	13.3
Other	3.6	(9.1)	2.20	(9.1)	22.7	(8.6)	21.2	(35.6)	1.1
Total	39.4	(100.0)	24.20	(100.0)	264.8	(100.0)	59.5	(100.0)	4.5
`li`									
`mark`	96.2	(23.3)	57.0	(28.8)	426.6	(18.8)	102.3	(5.5)	4.17
`sweep`	29.2	(7.1)	20.4	(10.3)	129.9	(5.7)	45.6	(2.5)	2.85
Other	286.7	(69.6)	120.3	(60.8)	1712.0	(75.5)	1708.0	(92.0)	1.00
Total	412.1	(100.0)	197.7	(100.0)	2268.4	(100.0)	1855.9	(100.0)	1.22

Table 3: Breakdown of runtime for scalar and vector SPECint95 applications. The `compress` benchmark is split into `comp` and `decomp` components. The scalar-optimized profile is given for the scalar systems, except for `m88ksim` on Ultra and Pentium-II, where the faster original version is profiled.

benchmarks is 1.32.

Individual functions exhibit much higher speedups, with several of the `ijpeg` routines running over 10 times faster in vector mode. The vector unit therefore improves performance by accelerating limited portions of the execution time by a large amount. Note that these vector speedups are measured relative to optimized scalar code, and are much larger than would be possible with current superscalar microarchitectures.

For `decomp` and `ijpeg`, there is also a more moderate speedup over the whole runtime, but this is due to the vectorized standard C library functions which are not timed individually.

Comparing the profiles of scalar code, we see that all the platforms are broadly similar in their distribution of runtime amongst the vectorizable and non-vectorizable portions of code. The biggest differences occur for the Pentium-II on `comp`, where it spends comparatively less time in vectorizable code, and on `li`, where it spends comparatively more. The workstations are superscalar designs whereas T0 is single-issue, suggesting that speedups from instruction-level parallelism are distributed throughout the benchmark rather than being concentrated in the regions that were vectorized.

6 Discussion

The above results present clear evidence that significant portions of the SPECint95 benchmark suite can be executed with vector instructions after some minor modifications to the source code, but the magnitude of the vector speedup is affected by several factors. The discussion in this section estimates the impact of these factors, which include fabrication technology, memory hierarchy, vector architecture, and code quality, and also shows how vector speedup can be profitably combined with superscalar speedup.

T0 is fabricated in an older 1.0 μm two-metal CMOS technology which results in a larger die area and lower clock rate compared with the newer technology used to fabricate the superscalar processors. Because it does not interfere with core processor functions, the addition of a vector unit should have no impact on processor cycle time, and so we can assume that vector microprocessors will have the same clock cycle as vectorless microprocessors when implemented in the same technology. Table 4 shows the performance of the benchmarks with all machines scaled to the same clock rate and with speedups measured relative to T0 running scalar-optimized code.

Benchmark/Routine	T0 Scalar	Ultra-1/170	Pentium-II	T0 Vector
`m88ksim`				
`killtime`	1.00	1.66	1.75	7.93
`ckbrkpts`	1.00	1.35	1.97	2.28
Other	1.00	1.92	1.85	1.00
Total	1.00	1.76	1.84	1.41
`comp`				
`packcodes`	1.00	0.48	0.75	8.58
Other	1.00	0.58	0.34	0.97
Total	1.00	0.57	0.36	1.07
`decomp`				
`unpackcodes`	1.00	1.45	0.84	9.46
Other	1.00	0.99	0.91	1.16
Total	1.00	1.04	0.90	1.37
`ijpeg`				
`rgb_ycc_convert`	1.00	1.34	0.97	10.7
`downsample`	1.00	2.10	1.85	9.5
`forward_DCT`	1.00	1.40	1.65	14.3
`encode_MCU`	1.00	1.43	1.08	1.4
`inverse_DCT`	1.00	1.92	1.52	9.2
`upsample`	1.00	1.61	1.35	13.3
Other	1.00	1.53	1.37	1.1
Total	1.00	1.61	1.46	4.5
`li`				
`mark`	1.00	1.06	1.00	4.17
`sweep`	1.00	1.07	0.85	2.85
Other	1.00	1.43	1.90	1.00
Total	1.00	1.32	1.53	1.22

Table 4: Relative speed of each component normalized to same clock rate on each platform. The speedups are measured relative to the scalar-optimized code running on T0. The workstation timings for `m88ksim` are for the original code which is faster for those systems.

T0 has an unconventional memory hierarchy, with a small 1 KB direct-mapped primary instruction cache (I-cache) but no data cache (D-cache), and a flat three cycle latency main memory system. This memory hierarchy should produce *lower* vector speedups compared with a more conventional memory system. The small I-cache produces more misses, concentrated outside of vectorizable loops which generate few I-cache misses. The overall effect is to reduce the vectorizable fraction of runtime, f_i, by increasing the time spent executing non-vectorizable scalar code. The lack of a primary D-cache has two main effects. First, all scalar accesses have three cycle latencies rather than the two cycles typical for primary D-caches. The tuned scalar code for the vectorizable loops is mostly able to hide this extra cycle of latency, whereas it will likely slow performance on other code. Second, there are no D-cache misses. While both vector and scalar unit will experience approximately the same number of D-cache misses in a conventional memory hierarchy, most of the vectorizable code has relatively long vectors which should help hide miss latencies to the next level of cache. The SPECint95 codes have relatively small working sets and little execution time is spent in misses from typical sizes of external cache, even on fast processors [CD96].

The vector speedup is also affected by the design of the vector unit. T0 has a simple vector unit that lacks some common vector instructions. For example, the addition of a vector compress instruction would reduce the execution time for the `li` `mark` routine from 102.3 seconds to an estimated 47 seconds, increasing speedup to a factor of 9. T0 also lacks masked vector memory operations which would further reduce `mark` runtime to around 33 seconds, improving vector speedup to 13. Another example is the `sweep` routine, where masked stores would decrease runtime to around 27 seconds, improving vector speedup to 4.9. The `encode_MCU` routine performance could be increased with the addition of a “count leading zeros” instruction to determine AC coefficient magnitudes. These vector unit optimizations require little additional die area, and no increases in main memory bandwidth. Several of the routines are limited by vector strided and indexed memory instructions that transfer only one element per cycle on T0. Additional address ports would improve throughput considerably, but at some additional cost in the memory system. An alternative approach to improve cost/performance would be to reduce the number of parallel lanes in the vector unit; this would reduce area but have limited performance impact for those routines limited by address bandwidth.

The relative timings are also affected by code quality. The vector routines were manually translated into assembly code, likely resulting in higher code quality than automatic compilation and hence greater vector speedup. The `ijpeg` DCT routines were the most difficult to schedule and would probably show the largest improvement over compiled code. But for the `comp`, `decomp`, and `li` benchmarks, and for the `rgb_ycc_convert`, `upsample`, and `downsample` portions of `ijpeg`, performance is primarily limited by the single address port and so there is little opportunity for aggressive assembler hand-tuning to improve performance. The vectorization of the `m88ksim` benchmark is trivial with no stripmine code and little choice of alternative instruction schedules, and so performance should be very close to that with a vectorizing compiler. The speedup provided by the vectorized standard C library routines requires no compiler support beyond function inlining.

Because of the effort involved, the manual vectorization strategy is limited to a few key loops in each benchmark. Compared with an automatically vectorizing compiler, this limits vector coverage and hence reduces vector speedup. Another distortion from hand-tuning is that only the scalar routines compared against vectorized routines were tuned, which reduces vector speedup compared with more careful tuning or higher quality compilation of the non-vectorizable scalar code.

Almost all of these factors act to reduce the vector speedup for T0 compared to a future vector microprocessor with a conventional memory hierarchy and an automatically vectorizing compiler. The exception is the quality of the vector code for each routine, but this only changes the vector speedup, V_i, not the vectorizable fraction, f_i. Because the vector speedups are high and the fraction vectorized is low, the resulting overall speedup is not very sensitive to the values for V_i. As a highly pessimistic example, consider if the vector speedups of the `ijpeg` DCT routines were reduced by a factor of 2, and other speedups were reduced by a factor of 1.5, except for the standard library routines and `encode_MCU` which remain unchanged. In this case, the resulting geometric mean speedup on SPECint95 would only drop to 1.26.

7 Combined Superscalar and Vector Execution

Because of their simple control logic and regular structure, vector units are inexpensive additions to microprocessor designs. For example, the QED R5000 microprocessor [Gwe96] has a single-issue integer unit, very similar to that of T0, together with a floating-point unit, memory management unit, and split primary caches each of 32 KB. Although the vector unit from T0 can complete up to 24 integer operations per cycle, it would add only 24% to the R5000 die area when scaled to the same technology. The results above suggest that a smaller vector unit, with half the number of lanes would achieve most of the benefit. This reduced vector unit would require only 12% additional die area and could use the existing 64-bit cache interfaces.

The results in Table 4 show that superscalar processors speed up both vectorizable and non-vectorizable code by approximately the same amount, whereas vector units only speed up the smaller vectorizable fraction but to a much greater degree. A particularly attractive design alternative is to combine vector and superscalar techniques, giving a combined speedup of

$$\frac{T_s}{T_{ss+v}} = \frac{1}{(1-f)/S + f/V}$$

where S is the superscalar speedup, and T_{ss+v} is the execution time of the combined superscalar and vector processor.

As an example, consider the MIPS R10000 [Yea96] which is a quad-issue out-of-order superscalar microprocessor. The R10000 achieves approximately 1.7 times speedup over the R5000 on SPECint95 when running at the same clock rate with the same external cache [Cora]. Based on the previous results, we can assume a vector unit would achieve a speedup of 1.32 over the R5000 by speeding up 28% of the execution time by a factor of 8. From the above equation, we can predict that adding a vector unit to the R10000 would increase its speedup to 2.18, or an additional 1.28 times greater than the superscalar speedup alone. Although the R10000 has the same primary cache configuration as the R5000, the multiple functional units and complex instruction issue management logic inflate the die to 3.4 times the area of the R5000. The full T0 vector unit would only add 7% area to the R10000 die, and could use the existing 128-bit primary and secondary cache interfaces.

8 Related Work

There has been little work in vectorizing codes outside of the traditional supercomputing application areas. Lee [Lee92] reported that the Cray compiler could automatically vectorize the main loop from SPECint92 `eqntott` but did not present performance numbers. Appel and Bendicksen vectorized a stop and copy garbage collector for the Cyber 205 vector memory-memory architecture but used synthetic garbage structures to measure performance [AB89]. The SCANDAL group at CMU has developed techniques for vectorizing irregular computations including sorting and various graph algorithms [RM94, Zag98]. Lee provides some performance data for a vectorized stream cypher running on T0 [LD97] and a comparison of performance and cost with superscalar microprocessors.

9 Summary and Conclusions

This study has shown that half of the SPECint95 benchmarks can be significantly accelerated with vector execution, though in some cases this required minor source code changes. For the T0 vector microprocessor, vector speedups are in the range 1.16–4.5 with a geometric mean performance increase of 1.32 across all eight benchmarks. Profiling results reveals that the vector unit accelerates code by providing large speedups but only on limited portions of the runtime, whereas superscalar processors accelerate code by providing more modest speedups over most of the runtime. Because they achieve speedup in different ways, the two architectural techniques can be combined to yield larger speedups than when each is used individually. For example, adding a vector unit to the MIPS R10000 superscalar microprocessor was estimated to improve SPECint95 by 28% while only requiring 7% additional die area.

Vector units have proven successful in accelerating scientific and engineering codes, and there has also been much recent interest in applying vectors to new multimedia applications. This paper demonstrates that these vector units can also accelerate a much wider range of tasks. Adding a vector unit to a superscalar processor can significantly improve cost/performance even for codes which have low levels of vectorization, because vector units are compact yet can provide much larger speedups than are possible with current superscalar microarchitectures on data parallel portions of the workload. The resulting superscalar vector architectures are a promising approach for future microprocessor designs.

10 Acknowledgments

Thanks to many anonymous.

References

[AB89] Andrew W. Appel and Aage Bendiksen. Vectorized garbage collection. *Journal of Supercomputing*, 3:151–160, 1989.

[Amd67] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, number 30, pages 483–485, 1967.

[CD96] Zarka Cvetanovic and Darrel D. Donaldson. AlphaServer 4100 performance characterization. *Digital Technical Journal*, 8(4):3–20, 1996.

[CDd$^+$95] A. Charnas, A. Dalal, P. deDood, P. Ferolito, B. Frederick, O. Geva, D. Greenhill, H. Hingarh, J. Kaku, L. Kohn, L. Lev, M. Levitt, R. Melanson, S. Mitra, R. Sundar, M. Tamjidi, P. Wang, D. Wendell, R. Yu, and G. Zyner. A 64b Microprocessor with Multimedia Support. In *Proceedings IEEE International Solid-State Circuits Conference*, volume 38, pages 178–179, February 1995.

[Cora] Standard Performance Evaluation Corporation. SPEC CPU95 Results. SPEC95 results are available from the SPEC web site: `http://www.specbench.org`

[Corb] Standard Performance Evaluation Corporation. *SPEC95*. 10754 Ambassador Drive, Suite 201, Manassas, VA 20109.

[Esp97] Roger Espasa. *Advanced Vector Architectures*. PhD thesis, Universitat Politècnica de Catalunya, February 1997.

[Gwe96] Linley Gwennap. R5000 improves FP for MIPS midrange. *Microprocessor Report*, 10(1):10–12, January 1996.

[Gwe97] Linley Gwennap. Klamath extends P6 family. *Microprocessor Report*, 11(2):1,6–8, February 1997.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture — A Quantitative Approach, Second Edition*. Morgan Kaufmann, 1996.

[Kan89] G. Kane. *MIPS RISC Architecture (R2000/R3000)*. Prentice Hall, 1989.

[KPP+97] Christoforos Kozyrakis, Stylianos Perissakis, David Patterson, Thomas Anderson, Krste Asanović, Neal Cardwell, Richard Fromm, Jason Golbus, Benjamin Gribstad, Kimberly Keeton, Randi Thomas, Noah Treuhaft, and Kathy Yelick. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, 30(9):75–78, September 1997.

[LD97] C. G. Lee and D. J. DeVries. Initial results on the performance and cost of vector microprocessors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 171–182, December 1997.

[Lee92] Corinna Lee. *Code optimizers and register organizations for vector architectures*. PhD thesis, University of California at Berkeley, May 1992.

[Lee96] Ruby B. Lee. Subword parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, October 1996.

[PW96] Alex Peleg and Uri Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–59, October 1996.

[RM94] Margaret Reid-Miller. List ranking and list scan on the Cray C-90. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 104–113, Cape May, NJ, June 1994.

[TONH96] Marc Tremblay, J. Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS speeds new media processing. *IEEE Micro*, 16(4):10–20, October 1996.

[WAK+96] John Wawrzynek, Krste Asanović, Brian Kingsbury, James Beck, David Johnson, and Nelson Morgan. Spert-II: A Vector Microprocessor System. *IEEE Computer*, 29(3):79–86, March 1996.

[Wel84] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[Yea96] Kenneth C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[Zag98] M. Zagha. Efiicient irregular computation on pipelined-memory multiprocessors. Ph.D. Thesis (In Preparation), 1998.