Published in *Proceedings of the First SUIF Compiler Workshop,* pp. 59–67, January 1995.

A Vectorizing SUIF Compiler

Derek DeVries and Corinna G. Lee
Department of Electrical and Computer Engineering
University of Toronto

Abstract

With advances in VLSI technology, it is now possible to implement vector processors on a single chip. Vector architectures are ideal for providing the cost-effective, real-time performance demanded by graphic-intensive applications such as MPEG players and video conferencing software. However, these vector microprocessors lack compilers and instead rely on libraries of hand-written assembly functions that use their vector hardware. For our research, we are developing a retargetable vectorizing compiler for such vector microprocessors to improve their ease of use.

The starting point for our work is the SUIF compiler. In this paper, we describe how we augmented SUIF and discuss which aspects of SUIF were or were not needed for the retargeting effort. For example, we added code that uses SUIF's library of dependence tests to identify vectorizable loops. On the other hand, because of the mapping of high-level loop structures to vector instructions, we do not rely heavily on SUIF's scalar optimizations to generate high quality vector code. To demonstrate the effectiveness of our SUIF vectorizing compiler, we show how it transforms an MPEG decoder program into vector assembly code.

Over the last decade, computer technology has developed to the point where applications such as M-PEG players, video conferencing programs like the H.261 standard, 3D visualization software, and CAD tools are enjoying widespread use. The proliferation of such graphic-intensive applications has relied traditionally on application-specific hardware to obtain the performance needed to make such applications viable for real-time use.Although software-only solutions are desirable for reasons of economy and ease of use, they suffer from poor performance and hence are rarely implemented. Rather than relying on either application-specific hardware or software programs to implement these graphic-intensive applications, we advocate using a hardware/software combination that provides the performance of the hardware-only solution with the ease-of-use of the software approach.The approach we propose is to use a vector processor as the underlying hardware support for effective software solutions that are written in a high-level language and that are then compiled into the assembly language of the target vector architecture.

Although vector architectures can provide the real-time performance demanded by graphic-intensive applications, their implementation costs have been the major prohibiting factor to the use of vector processors in this burgeoning applications market. With the steady advances in fabrication technology, however, this major obstacle has been removed recently and it has become possible to build a vector processor on a single chip. These new processors provide anywhere from a few simple vector-like instructions [1,2] to full-blown vector architectural support [3].

With today's technology, our proposed vector-based approach is close to being a viable solution. One key component, however, is missing: a compiler that can exploit the unique features of this new breed of vector processors and thus provide the requisite performance at a reasonable cost. Currently, the designers of these vector processors are using the library approach. In other words, handwritten assembly-language functions that use the vector features are provided in a library and are called by application programs that are written in a high-level language. One of the goals of our research is to develop a retargetable vectorizing compiler that will provide the missing link to our proposed hardware/software solution.

This paper describes our efforts to develop a vectorizing compiler. Our initial target architecture is the Torrent architecture [3], which is being developed by researchers at the University of California at Berkeley. We will then target the new UltraSPARC architecture with its VIS instructions [1]. The starting point for our work is the SUIF compiler. In this paper, we begin with a brief overview of vector architectures and the conditions a program fragment must meet to use vector hardware. We then describe the three phases of our vectorizing compiler and how they are integrated into SUIF. We conclude with a report on the current status of our compiler implementation.

1. Overview of Vector Architectures and Conditions for Vectorizability

In its simplest form, a vector architecture is one in which data is worked on in sets, rather than as discrete scalar values. A special vector register set is provided for this purpose. Each vector register in the set is composed of a number of scalar registers (elements). A special control register, the vector length register (VLR), is also provided. Whenever an action is performed, the contents of the VLR are examined and the first VLR elements of each specified vector register are used to perform the specified operation. A VLR value of zero implies that no operation will be performed, while a VLR value of one is equivalent to a scalar operation. Because many program constructs, such as branches and function calls, are inherently scalar, a scalar unit is also provided to handle such constructs in addition to handling any other program feature that cannot be operated on efficiently by the vector unit.

The Torrent architecture fits into this general vector model. However other processors, such as the UltraSPARC [1] or the HP-7100-LC [2], do not provide a separate vector unit. Instead they provide instructions that act on narrow, short vectors that are multiple 8- or 16-bit quantities stored in a single 64-bit scalar register.

Vector processors exploit data parallelism within inner loops. This is most easily visualized as providing a set of instructions that act on entire vectors at once, rather than on single elements of a vector. In reality the maximum vector size is fixed by the hardware. This ranges from 4 elements on the UltraSPARC to 32 elements in the Torrent to 128 or more vector elements in other vector architectures. Vectors longer than the maximum hardware vector size (MVL) must be worked on in sub-vectors of at most MVL.

For a statement in an inner loop to use vector hardware, it must satisfy certain conditions to guarantee that the use of vector instructions will not change the semantics of the original program. One condition is that the statement must use variables that can access different locations on different iterations of the loop. Such variables include array references and pointer references. We call such variables *vector operands* since they can be mapped to vector registers. The second condition is that the statement cannot produce a result that it or some other statement that it depends on uses in a later iteration. The result can be in the form of data or a condition that determines whether the statement executes. This second condition is the more difficult to determine and has been the focus of much research activity over the past 15 years. Note that statements involving both vector operands and scalar operands can also be vectorized if the vector operand satisfies the dependence constraint and the scalar operand is loop invariant, a function of the loop index, or a simple scalar reduction.

To determine whether a statement is vectorizable or not, a dependence graph is constructed [4]. A node in a dependence graph is a statement in a loop, and there is a directed edge from one statement A to another statement B if statement B uses a result produced by statement A, where the result can be in the form of data or control. Once the dependence graph is constructed, it is traversed to determine if it contains any cycles. A statement in a dependence cycle corresponds to a statement that is directly or indirectly self-dependent and thus does not satisfy the second condition for vectorizability.

The no-cycles constraint can be relaxed slightly if the dependence distance of the cycle can be determined. The dependence distance between two references to an array element is the number of loop iterations between those two references. By guaranteeing that the second reference will occur after the first, it is possible to use vector instructions even in the presence of a dependence cycle. Because the vector processor operates on a section of data instead of the entire vector at once, preserving the dependence cycle can be done by setting the VLR to be shorter than the smallest dependence distance in the cycle. Note that if the smallest dependence distance is one iteration, there is almost no performance benefit gained from using vector instructions. Hence this technique is only beneficial if the dependence distance is reasonably long.

2. Implementation Details of Vectorizing SUIF Compiler

We will now describe how we modified and augmented the SUIF compiler so that it will identify vectorizable program fragments and generate appropriate vector code for them. In addition to the usual compiler phases needed for translation, there are three distinct phases to vectorizing a source program: pre-processing, vectorization, and code generation. As Figure 1 shows, we modified various passes of SUIF and added our own to implement these three phases. The bold passes are those that we modified while the vectorize pass is the one we added.

The primary goal of the preprocessing phase is to minimize the number of scalar expressions present in the inner loop body, thereby expanding the number of expressions available to be vectorized. The source program is first translated to SUIF by *scc*. Then *moo* is run to locate all privatizable variables. A modified version of *porky* is then run to extract scalar reductions, eliminate induction variables and replace privatizable variables.

A new pass, *vectorize*, is then run. It performs a few simple data acquisition passes through the source code and performs the vectorization phase, which consists of dependence analysis and code transformation. For dependence analysis, *vectorize* builds a program dependence graph, containing both control and data dependence arcs, and uses this graph to perform vectorizing transforms on inner for-loop bodies. Inner loops are then strip-mined to execute in strips of at most the maximum vector length supported by the hardware, and the loop body is split into vectorizable and scalar execution components.

A loop, when stripmined, is transformed into two nested loops: an outer strip-control loop with a step size of some multiple of the original loop's step size, and an inner loop with the original step size and containing the loop body. When vector stripmining is performed, all scalar components of the original loop will appear in the inner loop (the scalar loop) and all vectorizable statements will appear in the body of the outer strip-control loop.

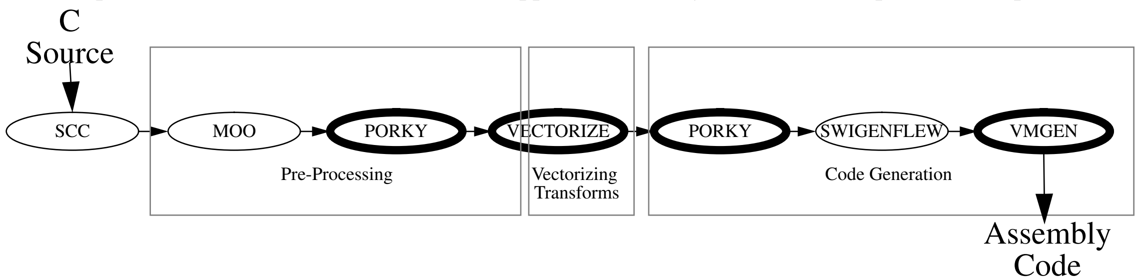


Figure 1: Structure of the vectorizing SUIF compiler

Code generation will proceed from this point, with applications of *porky* (again) and *swigenflew*, followed by a modified version of *mgen*. The target architecture for this work is Berkeley's Torrent microprocessor [3]. Because Torrent is a MIPS-II compliant ISA with vector enhancements implemented as co-processor two, the existing *mgen* need only be extended rather than completely re-written. In the future we plan to also target the UltraSPARC. A separate research effort at the University of Toronto is creating a SPARC code generator for SUIF [5]. Our vectorizing transformations will provide the front-end support needed to utilize the VIS instructions (Visual Instruction Set) supported by the UltraSPARC.

Throughout the following sections, a piece of the MPEG decoder loop[6] will be used to demonstrate the transformations used in the three phases. This loop maps MPEG frame data into a RGB colour image. In the full loop, the values for four adjacent pixels are calculated in each iteration. In Figure 2, we show only the calculation for one pixel. The first three lines of the loop body determine the average colour values for the four pixels. The last four lines merge these colour values with the luminosity value for a pixel to produce the colour values for one pixel. In the full loop, these last four lines are replicated (with small changes) to produce the colour values for the other three pixels. To produce a 640x400 image, this loop iterates 200 times, while the outer control loop (not shown) performs 320 iterations, for a total of 64 000 iterations of this loop body per displayed frame. For a real-time display, thirty frames must be processed per second, thus requiring 1.92 million iterations of the full MPEG decoder loop to be executed every second.

2.1 Pre-processing Phase

Moo and *porky* are used in conjunction to locate privatizable variables and perform operand substitution on these variables. A privatizable variable is a scalar variable which is dead at the start and end of the loop body. The SUIF distribution version of *porky* augments the names of privatizable variables with a prefix that is used in subsequent passes to identify these variables as privatizable. We modified *porky* so that an array of the maximum vector size is declared for each privatizable scalar variable. In addition, references to the scalar are replaced by references to the new array. This serves two purposes: it allows the dependence analyzer to include privatized variables in the dependence graph, and it allocates adequate space to spill these vectors to memory if needed without the need to know the size of the array being worked on. This technique does degrade performance on any statements that reference a privatized variable and that must be executed in the scalar loop.

Porky is also used to perform induction variable elimination and to extract scalar reductions from the loop body. These two transformations act to reduce the number of scalar variables involved in the loop body, thereby increasing the number of statements that are candidates for vectorization. The results of this transformation on the MPEG loop can be seen in Figure 3.

Vectorize is then used. It begins by searching inner loops for expressions using the loop index variable and attaching a SUIF annotation to the loop indicating whether such expressions are present. Array subscript calculations are not included in this search. If such expressions are present, an *induction vector*, a vector containing all values the loop index variable will assume, will be created in the next phase. At the same time, *vectorize* searches expression

Published in *Proceedings of the First SUIF Compiler Workshop,* pp. 59–67, January 1995.

```
#define MAX 0xff
#define BITS 8
#define BITS2 2*BITS
#define CLAMP(ll,x,ul) (((x)>(ul))?(ul):(x))

for (x=0;x<cols/2;x++) {
      CR=cr[off_set+x]; CB=cb[off_set+x];
      c_r=1.40200*CB; c_g = -0.71414*CB;
      c_g+=0.34414*CR; c_b=1.77200*CR;

      L=lum[2*x]<< BITS;
      r=(L+c_r)>>BITS; g=(L+c_g)>>BITS; b=(L+c_b)>>BITS;
      r=CLAMP(0,r,MAX); g=CLAMP(0,g,MAX); b=CLAMP(0,b,MAX);
      row1[2*x]=r|(g<<BITS)|(b<<BITS2);
}
```

a) The original MPEG kernel written in C

```
for (x = 0; x < cols / 2; x++)
      {
      CR=cr[off_set + x];
      CB=cb[off_set + x];
      c_r = 1.402 * (double)CB;
      c_g = (-0.714) * (double)CB;
      c_g = (double)c_g + 0.344 * (double)CR;
      c_b = 1.772 * (double)CR;
      L = lum[2 * x] << 8u;
      r = L + c_r >> 8u;
      g = L + c_g >> 8u;
      b = L + c_b >> 8u;
      if (255 < r)
                              { suif_tmp0 = 255; }
      else
                              { suif_tmp0 = r; }
      r = suif_tmp0;
      row1[2 * x] =  (unsigned int)(int)((unsigned int)r |
                     (unsigned int)(g << 8u)) | (unsigned int)(b << 16u);
      }
```

b) The SUIF representation of the kernel. The clamping of the g and b values have been deleted for brevity.

Figure 2: The MPEG example in both its original form and after being processed by *scc.* The macros have been expanded.

trees for scalar references that are not loop invariant or functions of the induction variable and marks these trees as non-vectorizable.

Further bookkeeping is done at this time to handle functions and any breaks from the loop. For clarity these issues are dealt with in a later section.

2.2 Dependence Analysis and Code Transformation

Vectorize now enters the next phase, which performs dependence analysis and code transformation.

The first step is to develop a dependence graph to represent both structured control flow and data dependence. *Vectorize* traverses the program, attaching control dependence annotations to the top-level instructions of each expression tree in the loop body. These annotations record both the condition test on which the expression depends and whether the expression resides in the *then* or *else* clause of the *if* construct. Control dependencies introduced by breaks statements are handled separately and hence are not included in the dependence graph. Next, a second pass through the source code is made, gathering all array references and using the SUIF dependence library to test for

```
for (x = 0; x < cols / 2; x++)
      {
      privatized_CR4[x] = cr[off_set + x];
      privatized_CB5[x] = cb[off_set + x];
      privatized_c_r6[x] = 1.402 * (double)privatized_CB5[x];
      privatized_c_g7[x] = (-0.714) * (double)privatized_CB5[x];
      privatized_c_g7[x] =   (double)privatized_c_g7[x] +
                             0.344 * (double)privatized_CR4[x];
      privatized_c_b8[x] = 1.772 * (double)privatized_CR4[x];
      privatized_L9[x] = lum[2 * x] << 8u;
      privatized_r10[x] = privatized_L9[x] + privatized_c_r6[x] >> 8u;
      privatized_g11[x] = privatized_L9[x] + privatized_c_g7[x] >> 8u;
      privatized_b12[x] = privatized_L9[x] + privatized_c_b8[x] >> 8u;
      if (255 < privatized_r10[x])
                             { privatized_suif_tmp0113[x] = 255; }
      else
                             { privatized_suif_tmp0113[x] = privatized_r10[x]; }
      privatized_r10[x] = privatized_suif_tmp0113[x];
      row1[2 * x] =  (unsigned int)(int)((unsigned int)privatized_r10[x] |
                     (unsigned int)(privatized_g11[x] << 8u)) |
                     (unsigned int)(privatized_b12[x] << 16u);
      }
```

Figure 3: The kernel after pre-processing. In this case all scalar references have been replaced by vector temporaries.

dependences between these references. The dependence vectors reported by the library are attached to the expression tree.This completes the generation of the dependence graph. Data dependencies are recorded at the reference point in the expression, while control dependencies are recorded at the top of each expression tree.

The upper portion of Figure 4 shows the dependence graph for the example MPEG loop. From Figure 3, we can see that the loop body contains 15 statements that reference array variables. Hence the associated dependence graph contains 15 nodes labeled 1 through 15. Data dependencies are shown as solid directed arcs while control dependencies are grey-shaded. Note that there are control dependencies from the *if* statement (#11) to the *then* and *else* statements (#12 and #13).

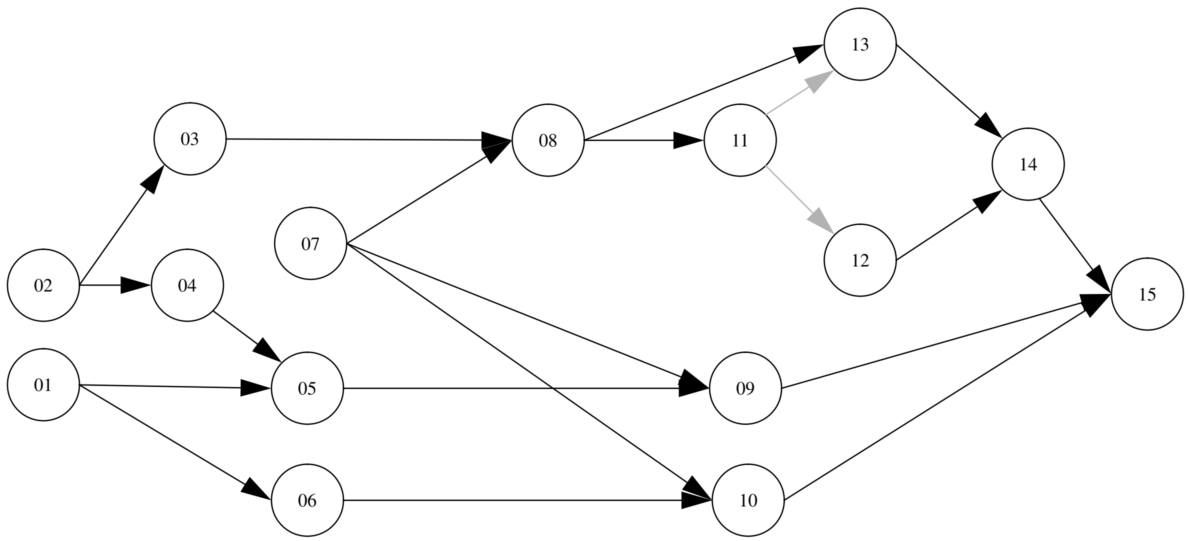
The dependence graph is passed to Tarjan's algorithm, as presented by Wolfe [4]. Tarjan's algorithm traverses the dependence graph and classifies each statement as either vectorizable, self-dependent, or as participating in a cycle. It returns a list of each type. Since Tarjan's algorithm is concerned only with the contents of the dependence graph it may report some statements that contain both array and scalar references as being vectorizable when the scalar component of the expression actually makes vectorization impossible. For this reason, the list of vectorizable statements are then checked for annotations which were generated in the preprocessing phase and which indicate that the associated statements are not vectorizable. Any which are not vectorizable are removed from the vectorizable list, while all those remaining in the list receive an annotation indicating their vectorizable status.

The list of vectorizable statements is passed to a simple sorting algorithm which uses the dependence graph to determine the correct order for execution of these statements. Since there are no cycles, the sort is guaranteed to find a viable execution order.

Now code transformations can occur. The *for* loop must be strip-mined, so that it executes in strips of at most the maximum vector length (MVL). Since the desired number of iterations is not likely to be an integer multiple of the maximum vector length, the first strip will execute fewer iterations than this, while all subsequent strips will be the maximum length permitted by the hardware. MVL is typically a power of 2, thus the calculation of the length of the first strip requires only a simple bit-wise *and* operation. The inner stripmined loop contains all the scalar components of the original loop body. The sorted vectorizable statements appear outside this inner scalar loop, but within the strip-control loop. The result of this transformation and the example MPEG loop can be seen in Figure 4. Note that only the strip-control loop is shown for our example. This is because there are no scalar components in the example and hence no need for an inner scalar loop.

When conditionals are involved in the loop body, the *if* gets disassembled into its three components: the condition test, the *then* clause and the *else* clause. The condition test is executed to produce a *condition vector*, a boolean

Published in *Proceedings of the First SUIF Compiler Workshop,* pp. 59–67, January 1995.




```
VL17 = cols / 2 & (MVL-1);
VLhold18 = VL17;
for (Vindx16 = 0; Vindx16 < cols / 2; Vindx16 += VL17)
      {
      privatized_CB5[Vindx16] = cb[off_set + Vindx16];
      privatized_c_g7[Vindx16] = (-0.714) * (double)privatized_CB5[Vindx16];
      privatized_CR4[Vindx16] = cr[off_set + Vindx16];
      privatized_c_g7[Vindx16] = (double)privatized_c_g7[Vindx16] + 0.344 *
                                  (double)privatized_CR4[Vindx16];
      privatized_L9[Vindx16] = lum[2 * Vindx16] << 8u;
      privatized_g11[Vindx16] = privatized_L9[Vindx16] + privatized_c_g7[Vindx16] >> 8u;
      privatized_c_b8[Vindx16] = 1.772 * (double)privatized_CR4[Vindx16];
      privatized_b12[Vindx16] = privatized_L9[Vindx16] + privatized_c_b8[Vindx16] >> 8u;
      privatized_c_r6[Vindx16] = 1.402 * (double)privatized_CB5[Vindx16];
      privatized_r10[Vindx16] = privatized_L9[Vindx16] + privatized_c_r6[Vindx16] >> 8u;
      if (!(255 < privatized_r10[Vindx16]));
      privatized_suif_tmp0113[Vindx16] = 255;
      privatized_suif_tmp0113[Vindx16] = privatized_r10[Vindx16];
      privatized_r10[Vindx16] = privatized_suif_tmp0113[Vindx16];
      row1[2 * Vindx16] =    (unsigned int)(int)((unsigned int)privatized_r10[Vindx16] |
                             (unsigned int)(privatized_g11[Vindx16] << 8u)) |
                             (unsigned int)(privatized_b12[Vindx16] << 16u);
 VL17 = VLhold18;
 VLhold18 = MVL;
 }
```

Figure 4: The dependence graph and transformed code for MPEG.

vector containing all condition test results for the current iteration of the strip-control loop. The condition vector is used to modify the execution of all statements from within the *then* and *else* clauses. The conditional dependence is managed through annotations, and so the control relationships are not apparent in the transformed code when it is translated back into C. Hence, in the case of the MPEG example in Figure 4, it appears that the *then* and *else* clauses are both being executed unconditionally. The condition vector is generated and managed by the code generator.

Because dependence arcs may exist between the scalar loop and the vector statements outside this loop, where to issue the scalar loop is an important consideration. To determine where in the list of vector instructions the scalar body should appear, the scalar loop can be viewed as a single, extremely powerful, vector instruction. If a position in the vector instruction list exists where all dependence conditions are satisfied, then the scalar loop can be placed there. Otherwise, if this pseudo-instruction participates in any dependence cycle, all vector instructions participating in that cycle must be moved into the scalar loop, or aggressive analysis of the scalar loop needs to take place in the hope of being able to partition it into multiple scalar loops that can be scheduled separately. If there are no depen-

Published in *Proceedings of the First SUIF Compiler Workshop,* pp. 59–67, January 1995.

dence arcs between the vector list and scalar loop body, then the order is immaterial.

As a final step, a few operations are added to the strip-control loop. These instructions manage the setting of the vector length register for the next iteration. In addition, if an induction vector is needed, its generation is placed in the landing pad, and its update is placed at the end of the outer loop.

Vectorizing code transformations are now complete. The resulting SUIF file contains enough information for the code generation pass to produce vector code.

2.3 Functions and Breaks

Functions cause particular problems for a compiler attempting to vectorize a loop. In the absence of robust interprocedural analysis, we must assume that the function will change all of the variables constituting the loop state. Under this assumption, we must forgo vectorizing any loop containing function calls. To expand the number of loops available for vectorization, the compiler must be supplied with more information about the behaviour of the functions. We introduce two classes of functions: pure and safe-global. A pure function is one which accepts zero or more arguments and returns a value based only on those arguments. It does not modify its arguments nor any global variables. Safe-global functions are ones which, while they do not change either their parameters or global variables affecting (or being affected by) the loop, they do modify some global data. For example, finding the cosine of an angle is a pure function, while pushing a number onto a stack is a global- safe one. Function names in each class may be specified at the *vectorize* command line.

Any function call with arguments permitting vectorization and with its name appearing in the pure or safe-global function lists may be bracketed by a vector-emulation loop and placed in the strip-control loop. This extra scalar loop will degrade performance, but the mix of vector and scalar bodies should perform better than a scalar-only execution. As an added benefit, keeping the function call out of the scalar loop body decreases the likelihood of a cycle occurring between the vector list and the scalar loop. If the loop body contains a conditional break, the reason for the two classes of function becomes apparent.

Loop bodies containing conditional breaks, a common occurrence in general-purpose code, pose special problems for a vector processor. In the absence of dependence cycles, it is possible to vectorize a loop containing break statements although care must be taken to ensure correct functionality. Some number of vector statements will be executed with vector length *N*, followed by the break determination for that strip, and then by the execution of the remainder of the statements for the strip. For a strip without a true break condition, this second set of statements is also executed with a vector length of *N*. If the break condition became true on element *M* of the break test, then we must kill the unneeded results from the first half of the strip and prevent the calculation of unneeded results in the second half of the loop.

Recall that during the dependence analysis pass, break-induced control dependencies were ignored. Consequently, the subsequent sort can (and likely will) mix statements from before and after the break. This statement reordering complicates the vectorizing of break statements slightly. Any statement that appeared before the break in the original code must be executed *M* times, while any statement appearing after the break must be performed *M–1* times.

To correctly vectorize a loop with break statements, each statement in a loop containing a break carries an annotation indicating its position relative to the break. This information is used in the code generation phase to set the VLR appropriately to commit the correct number of results. Committing the results cannot occur until all break conditions for the loop have be evaluated.

A loop containing both break statements and function calls presents a somewhat more complicated situation. Since we do not know on what iteration the break will occur, we do not know how many times we will need to call a function that appears in such a loop. A pure function can be called as often as desired, the worst result of unnecessary function invocation being wasted time. A safe-global function can only be called the same number of times it would have been called in the original scalar code. Thus if such a function appears after its associated break in the transformed code, the emulation loop mechanism will still work. If the function call must appear before its break determination, it must be moved to the scalar loop body.

The test for functions and the creation of book-keeping annotations for breaks are performed in *vectorize* during the preprocessing phase.

2.4 Code Generation

Since almost all of the required transformations have been applied at the source-code level, code generation will

Published in *Proceedings of the First SUIF Compiler Workshop,* pp. 59–67, January 1995.

be a relatively straightforward process. Ignoring the added complication of control flow, each expression tree annotated as vectorizable can be viewed exactly as a scalar expression tree of the same form. Each program statement gets divided into its scalar and vector components. The scalar operations are usually devoted to preparing any scalar register values, such as base addresses or stride lengths for loads. These scalar operations are then followed by the of the vector operation.

Figure 5 shows the MIPS assembly code for the MPEG decode loop. For easier readability, we use a variant of the vector instructions implemented in the Torrent architecture. The length of a vector strip is stored in the register VLR and the maximum vector length is 32. The privatized vector operands are assigned to vector registers and hence there is no need to perform address calculations for these operands. In addition, the address calculation for a non-privatized array reference is performed only once for several accesses because several elements of an array are accessed by one vector instruction.

```
#VL17 = cols / 2 & 31;
          load     r1,&cols;
          lsri     r2,r1,1;
          andi     r2,r2,31;
#VLhold18 = VL17;
          cpy      r3,r2;
#for (Vindx16 = 0;
#      Vindx16 < cols / 2;
#      Vindx16 += VL17)
          ldc      r4,0;
          label:   TOP;
          cfvu     VLR,r2; #set vector length
#privatized_CB5[Vindx16] = cb[off_set + Vindx16];
          load     r5,&off_set;
          add      r5,r5,r4;
          addi     r5,r5,&cb;
          lbiai.v  v1,r5,r0;
#privatized_c_g7[Vindx16] = (-0.714) *
#                  (double)privatized_CB5[Vindx16];
          ldc      r5,-0.714;
          fxmul    v2,v1,r5;
#privatized_CR4[Vindx16] = cr[off_set + Vindx16];
          load     r5,&off_set;
          add      r5,r5,r4;
          addi     r5,r5,&cr;
          load.v   v3,r5;
#privatized_c_g7[Vindx16] =
#      (double)privatized_c_g7[Vindx16] +
#      0.344 * (double)privatized_CR4[Vindx16];
          ldc      r5,0.344;
          fxmul    v4,v3,r5;
          fxadd    v2,v4,v2;
#privatized_L9[Vindx16] = lum[2 * Vindx16] << 8u;
          ldc      r6,2;
          sll      r5,r4,r6;
          addi     r5,r5,&lum;
          load.v   v4,r5,r6;
          ldc      r5,8;
          sll      v4,r5,v4;
#privatized_g11[Vindx16] = privatized_L9[Vindx16] +
#       privatized_c_g7[Vindx16] >>8u;
          slr      v2,r5,v2;
          fxadd    v2,v2,v4;
#privatized_c_b8[Vindx16] = 1.772 *
#      (double)privatized_CR4[Vindx16];
          ldc      r5,1.772;
          fxmul    v3,v3,r5;
#privatized_b12[Vindx16] = privatized_L9[Vindx16] +
#      privatized_c_b8[Vindx16] >>8u;
          ldc      r5,8;
          slr      v3,r5,v3;
          fxadd    v3,v3,v4;
#privatized_c_r6[Vindx16] = 1.402 *
#      (double)privatized_CB5[Vindx16];
          ldc      r5,1.402;
          fxmul    v1,v1,r5;
#privatized_r10[Vindx16] = privatized_L9[Vindx16] +
#       privatized_c_r6[Vindx16] >>8u;
          ldc      r5,8;
          slr      v1,r5,v1;
          fxadd    v1,v1,v4;
#if (!(255 < privatized_r10[Vindx16]));
          ldc      r5,256;
          slt      v5,v1,r5;
#privatized_suif_tmp0113[Vindx16] = 255;
          ldc      r5,255;
          cmvequ   v6,v5,r5;
#privatized_suif_tmp0113[Vindx16] =
#      privatized_r10[Vindx16];
          cmvgtz   v6,v5,v1;
#privatized_r10[Vindx16] =
#      privatized_suif_tmp0113[Vindx16];
          cmvequ   v1,v0,v6;
#row1[2 * Vindx16] =
#     (unsigned int)privatized_r10[Vindx16] |
#     (unsigned int)(privatized_g11[Vindx16] << 8u))|
#     (privatized_b12[Vindx16] << 16u);
          ldc      r5,16;
          sll      v3,r5,v3;
          ldc      r5,8;
          sll      v2,r5,v2;
          or       v2,v2,v3;
          or       v1,v1,v2;
          slri     r5,r4,1;
          addi     r5,r5,&row1;
          ldc      r6,2;
          store.vs v1,r5,r6 #vector store, stride 2
#VL17 = VLhold18;
          cpy      r2,r3;
#VLhold18 = 32;
          ldc      r3,32;
          add      r4,r4,r2; #increment loop index
          slt      r5,r4,r1; # test and branch
btrue     TOP,r5;
```

Figure 5: Assembly code for the MPEG loop. Code for fixed-point to integer conversion has been omitted.

It is instructive to contrast the vector code in Figure 5 to an equivalent scalar version. Obtaining an equivalent scalar version of this loop is a simple matter of replacing the vector instructions with their scalar equivalents and updating the loop index variable by one rather than the vector strip length. An immediate savings is obtained by using the vector version: address calculations and loop-overhead instructions are executed once per VL iterations. Using vector memory instructions essentially converts array references into pointer references for accessing individual elements of an array. To achieve comparable performance when using scalar-only code, extensive optimizations involving data flow analysis is required to eliminate the address calculations and convert the array references to pointer references.

Control flow complicates code generation to a great extent, since control across many iterations of the scalar loop body must be considered. *If* constructs require that a condition vector be generated. Calculations dependent on the *if* condition are performed either with the condition vector as an operand or by using it to mask out unneeded values. In the case of the Torrent, the condition vector can be used to selectively copy vector elements from one vector register to another using instructions such as "cmvequ" and "cmvgtz". For *if* constructs that do not assign to an array element in both the *then* and *else* clauses, it may be required to load the target array, perform the selective move, and then write the array back to memory.

Non-structured control flow caused by breaks present even more complications. Since we cannot know where the break will occur, we cannot commit any results to memory before the break has been evaluated. This simplifies killing of unneeded results since we need only set the vector length on the store correctly. It does mean that we have greater need for vector registers or memory spill space to keep these temporary results around until the break has been determined. When multiple break points exist from the loop, we must wait until the last one has been evaluated before we can safely commit operations.

3. Status & Improvements

At this point we can successfully perform vectorizing transforms on moderately complex code. Preprocessing passes to find and replace privatizable variables and gather information about program statements are complete. The dependence graph is developed with both data and control dependencies and is used to partition loops into vectorizable and non-vectorizable statements and to re-order the vector instruction list while preserving program semantics. All described code transformations take place except for the generation of the induction vector and correct placement of the scalar loop body. At this time the position of the scalar loop is fixed.

Our current effort is directed at incorporating the break handling and function classification mechanisms described in this paper into the *vectorize* pass. Following this, the placement algorithm for the scalar loop body will be implemented.

After work in these areas is complete, work on the code generator for the Torrent architecture can proceed. This will involve extending the SUIF distribution version of *mgen*. Once the Torrent implementation has been validated with Berkeley's Torrent simulator, work on retargeting to the UltraSPARC can begin. The largest part of the work for this retargeting will take place in the code generator. We hope to be able to reuse most of the code in the pre-processing and vectorization passes.

References

[1] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, G. Zyner. The Visual Instruction Set (VIS) in UltraSPARC. In *The Proceedings of Compcon'95*, pages 462-469, San Fransisco, March 1995.

[2] Ruby B. Lee. "Accelerating Multimedia with Enhanced Microprocessors." *IEEE Micro*, Special Issue on Hot Chips VI, Vol 15, No 2, pages 22-32, April 1995.

[3] Krste Astanovic. The Torrent Architecture Manual. University of California, Berkeley, 1994.

[4] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, 1989.

[5] Todd Mowry and Antonia Zhai. Scalar Optimization & Code Generation Development at the University of Toronto. To appear in *Proceedings of the First SUIF Compiler Workshop*, January, 1996.

[6] Arno Fornella. Some MPEG Decoding Functions on Spert: An Example for Assembly Programmers. TR-94-027, International Computer Science Institute, 1994.