Compiling for Vector-Thread Architectures

Mark Hampton
MIT Computer Science and
Artificial Intelligence Laboratory
Cambridge, MA 02139
mhampton@csail.mit.edu

Krste Asanović
Computer Science Division, EECS Department
University of California at Berkeley
Berkeley, CA 94720-1776
krste@eecs.berkeley.edu

ABSTRACT

Vector-thread (VT) architectures exploit multiple forms of parallelism simultaneously. This paper describes a compiler for the Scale VT architecture, which takes advantage of the VT features. We focus on compiling loops, and show how the compiler can transform code that poses difficulties for traditional vector or VLIW processors, such as loops with internal control flow or cross-iteration dependences, while still taking advantage of features not supported by multithreaded designs, such as vector memory instructions. We evaluate the compiler using several embedded benchmarks and show that we can obtain substantial speedups over a single-issue, in-order scalar machine.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); C.1.4 [**Processor Architectures**]: Parallel Architectures; D.3.4 [**Programming Languages**]: Processors

General Terms

Design, Performance

Keywords

Vector processors, Compilers, Code generation

1. INTRODUCTION

In recent years, computer architects have focused on improving performance using explicitly parallel computing instead of relying on more complex superscalar designs and higher clock rates. As a result, compiler technology has become increasingly important for new processor designs. Established explicitly parallel architectures, such as vector or VLIW, have a rich history of compilation techniques, but they also suffer some limitations on the types of codes that can be parallelized—e.g. vectorization is usually restricted to innermost loops. More recently, with the advent of the multicore era, architects have been exploring multithreaded approaches. However, multithreaded designs usually incur high synchronization costs, and are considerably less efficient than vector architectures for data-parallel tasks.

The vector-thread (VT) architectural paradigm [19, 20] unifies the vector and multithreaded execution models. A control processor interacts with a vector of *virtual processors*, and can broadcast instructions to them in a similar manner to a vector machine. Virtual processors also have the ability to direct their own control flow, as in a multithreaded design. The Scale architecture is an instantiation of the VT paradigm, which simultaneously exploits data-level, thread-level, and instruction-level parallelism. Designed as an "all-purpose" architecture, Krashinsky et al. [19, 20] have shown that Scale provides competitive performance across several embedded application domains, and that it particularly excels at exploiting loop-level parallelism. However, their work used handwritten assembly code, which is impractical for large-scale application development.

In this paper, we describe our compiler for the Scale architecture. The primary contribution of this paper is the development of a working back-end code generator that is able to take advantage of the architectural features in Scale. Our compiler infrastructure is still early in its development—for example, the compiler does not currently handle while loops, and it does not incorporate any novel analyses. However, even at this early stage, the compiler is able to parallelize simple DOALL loops as well as loops that typically present problems for vector or VLIW compilers, such as loops with internal control flow, loops with cross-iteration dependences, and outer loops. We evaluate the compiler across several embedded benchmarks and show significant speedups over a single-issue scalar processor.

2. VECTOR-THREAD ARCHITECTURE BACKGROUND

This section provides an overview of vector-thread architectures and the Scale VT processor prototype. For further details, refer to Krashinsky et al. [19, 20].

2.1 Vector-Thread Architectural Paradigm

In the VT architectural model, a control processor interacts with a vector of virtual processors. Each virtual processor (VP) is a thread that contains a set of registers and execution resources. VPs execute RISC-like instructions that are grouped into atomic instruction blocks (AIBs). The control processor can *vector-fetch* an AIB that will be exe-

CGO'08, April 5–10, 2008, Boston, Massachusetts, USA.

cuted by all of the VPs in parallel, thus exploiting data-level parallelism (DLP). The control processor can also execute *vector-load* and *vector-store* commands to transfer blocks of data between VPs and memory. Each VP can also direct its own control flow with a *thread-fetch* of the next AIB, thus enabling thread-level parallelism (TLP). A VP thread halts and waits for the next vector-fetch from the control processor after executing an AIB that does not issue a thread-fetch instruction. The execution of vector-fetched AIBs and vector memory commands can be freely intermingled with the execution of thread-fetched AIBs, allowing DLP and TLP to be exploited simultaneously within the architecture. The VPs are also connected by a unidirectional ring, the *cross-VP network*, which allows each VP to send values to its neighbor.

2.2 The Scale Vector-Thread Processor

The Scale processor prototype [21] is a low-power, high-performance design for embedded systems, and it demonstrates that the VT paradigm is well-suited for this application domain. Figure 1(a) is a simplified high-level diagram of the Scale microarchitecture. Scale contains four parallel lanes within the vector-thread unit (VTU). Virtual processors are striped across the physical lanes and are time-multiplexed within each lane to share the physical execution resources. The example configuration presented in Figure 1(a) has a vector length of 16, but Scale can support up to 128 VPs. A special vector memory unit (VMU) handles vector-load and vector-store commands.

Figure 1(b) is a more detailed view of one lane, showing how it is actually partitioned into four heterogeneous clusters to support instruction-level parallelism (ILP). All clusters can handle standard integer ALU instructions. Additionally, cluster 0 supports VP memory operations, cluster 1 supports VP fetch instructions, and cluster 3 supports integer multiply and divide. Clustering is exposed to the compiler, which is responsible for partitioning VP instructions between the clusters. Scale makes extensive use of decoupling to tolerate latencies, so different clusters within a lane can be executing code for different VPs simultaneously. There are four separate cross-VP networks in Scale, connecting sibling clusters in different lanes, and these networks contain queues to provide decoupling between neighboring VPs.

A VP's general registers in each cluster are either *private*—if they are only used by that VP—or *shared*—if they are used by other VPs. The shared registers can be used to hold constants set by the control processor, or to hold temporary values within the execution of a single AIB (the execution of each AIB is atomic with respect to other AIBs, but VPs can interleave execution at AIB boundaries), or to implement efficient reduction operations across VPs. Although not shown in Figure 1(b), cluster 0 also contains a separate set of *store-data* registers which are similar to private registers but are only used for holding values to be stored to memory. Additionally, each ALU's input operands are exposed as programmer-visible *chain registers*, which are used to avoid accessing the register files for short-lived values to reduce energy [6]. The use of chain registers and shared registers also reduces the per-VP register requirements. For example, as shown in Figure 1(b), cluster 2 does not use the physical register file; this could be due to the fact that instructions on that cluster only use chain registers for their computations. By contrast, cluster 3 uses all available physical registers. The maximum length of the virtual processor vector is dependent on the cluster with the highest per-VP physical register requirements, as discussed in the next section. In Figure 1(b), cluster 3 determines the maximum vector length. An important Scale code optimization is to try to use the chain registers and shared registers in each cluster to reduce private register usage, hence allowing increased vector length and potentially greater performance.

2.3 Scale Code Example

The typical programming model for a VT architecture is to map loops to the virtual processor vector, with each VP executing an iteration. Figure 2 shows a simple vectorizable loop, and we here assume that the input and output arrays are guaranteed to be disjoint. The Scale code for this function is shown in Figure 3. The code contains both control processor instructions as well as VP instructions, which are delimited by `.aib begin`/`.aib end` directives. The control processor is responsible for issuing a `vcfgvl` configuration command to specify the number of private and shared registers used in each cluster, which determines the maximum vector length, `vlmax`. Note that for the purposes of the configuration, the number of private registers listed for cluster 0—which also supports memory operations—is the larger of the number of private registers used and the number of store-data registers used. In Figure 3, the register requirements of cluster 3 determine `vlmax`: $(\lfloor(32-1)/1\rfloor) \times 4 = 124$. If a configuration command does not use any private registers, the maximum vector length is set to 128.

The `vcfgvl` command in the example causes the active vector length, `vl`, to be written to the `t0` register. The active vector length is the minimum of `vlmax` and the value of `a0`, which holds the number of loop iterations to be executed. Since the multiply coefficient is a constant value, the control processor writes it into a shared register on cluster 3. It then uses *strip mining* [24] to launch multiple loop iterations simultaneously. In each strip-mined loop iteration, the control processor sets the vector length, and then performs two vector loads (with an auto increment addressing mode) to obtain the inputs. The actual computation is performed by vector-fetching the AIB, causing the VPs to execute the multiply-add sequence on the input elements. Each VP places its result in a store-data register, which is used by the auto-incrementing vector store. The strip-mined loop continues until all elements have been processed.

```
void mult_add(int len, int *in1, int *in2, int *out) {
  int i;
  for (i = 0; i < len; i++)
    out[i] = COEFF*in1[i] + in2[i];
}
```

Figure 2: C code for mult_add function.

Note that the given code example is very similar to what might be generated for a traditional vector architecture. A key difference is that VP instructions are grouped within an AIB that the control processor issues in one vector-fetch instruction, whereas a traditional vector machine would issue separate vector-multiply and vector-add instructions. While existing vectorization techniques can be leveraged for Scale,

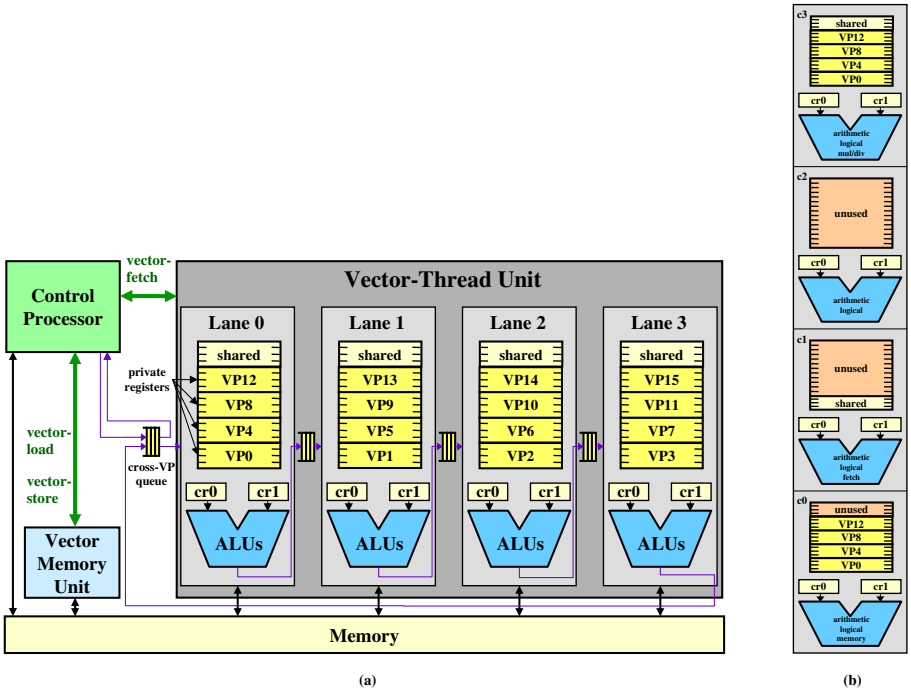

Figure 1: (a) High-level overview of Scale architecture. For simplicity, certain details are omitted, such as the use of clustering. (b) Expanded diagram of the clusters within a single lane.

there are also some unique compilation challenges, which are covered in the next section.

3. COMPILER OVERVIEW

This section presents an overview of the Scale compiler infrastructure[1]. Figure 4 shows the compiler flow for our work. The infrastructure ties together three existing compilers: SUIF [34] is used as the front end; Trimaran [10] is used for back end code generation; and a GCC-based cross-compiler tool chain [2] is used to produce the final executable. The SUIF front end parses a C source code file and converts it into the SUIF intermediate representation (IR). The compiler then performs memory dependence analysis, annotating the IR with dependence information that will be later used by Trimaran. The SUIF IR is then fed into a SUIF-to-Trimaran converter [23], which outputs the program in Trimaran's intermediate representation. Trimaran performs classical optimizations including common subexpression elimination, copy propagation, loop-invariant code motion, and dead code elimination. The output of the optimization phase is then sent to the scalar-to-VP code transformation phase, which attempts to map code to the vector-thread unit. After the transformation phase, cluster assignment is performed, and then the first (prepass) instruction scheduling phase occurs. Prepass instruction scheduling is followed by register allocation and a second (postpass) instruction scheduling phase, after which AIBs are formed for the VP code. Once AIB formation occurs, the compiler then searches for opportunities to replace occurrences of general registers with chain registers. Finally, Trimaran generates assembly code, and this is processed by the GCC cross-compiler to create the final binary executable. The following sections discuss certain key compiler phases in more detail, primarily focusing on how code is actually mapped to the vector-thread unit.

[1]All references to the "Scale compiler" in this paper refer to the compiler for the Scale architecture, not the Scale compiler [3] for the TRIPS architecture.

3.1 Memory Dependence Analysis

A key motivating factor for using SUIF as our front end is its dependence library, as an accurate dependence graph is important in order to determine what parallelism can be exploited in the program. We use the library to annotate memory operations with direction vectors indicating the existence of dependences. A special direction type is used for cross-iteration dependences with a distance of 1, as these can be mapped to transfers on the cross-VP network.

The dependence analysis is potentially complicated by the fact that Scale was designed for embedded systems, which typically run programs written in C. The use of pointers in C programs creates an aliasing problem, in which the compiler cannot determine whether two different pointers will access the same memory location, and hence must make an extremely conservative assumption of dependences, resulting in little or no exploitable parallelism. To help with this problem, we extended the SUIF front end to support the `restrict` keyword, which indicates that the object pointed to by a particular pointer will not be accessed by any other pointer.

3.2 Scalar-to-VP Code Transformation

In this phase, the compiler attempts to map parallel sections of code to the vector-thread unit. Since VT architectures excel at exploiting loop-level parallelism, the compiler currently only focuses on transforming loops, which frequently dominate execution time in the types of applica-

```
mult_add: #a0=len, a1=in1, a2=in2, a3=out
  # vcfgvl command below configures the number of private and
  # shared registers for each cluster to determine vlmax; the
  # command also sets both the active vector length (vl) and
  # the t0 register to the minimum of a0 and vlmax
  # configuration format: c0:p,s c1:p,s c2:p,s c3:p,s
  vcfgvl  t0, a0,            1,0,   0,0,   1,0,   1,1
  sll     t0, t0, 2          # input/output stride
  vwrsh   COEFF, c3/sr0      # write constant to shared reg
stripmineloop:
  setvl   t1, a0             # (vl,t1) = min(a0,vlmax)
  vlwai   a1, t0, c3/pr0     # vector-load in1, inc ptr
  vlwai   a2, t0, c2/pr0     # vector-load in2, inc ptr
  vf      scale_add_aib      # vector-fetch AIB
  vswai   a3, t0, c0/sd0     # vector-store out, inc ptr
  subu    a0, t1             # decrement counter
  bnez    a0, stripmineloop # loop until done
  vsync                      # allow VPs to finish
  jr ra                      # return

scale_add_aib:
  .aib begin
  c3      mult.lo pr0, sr0      -> c2/cr0   # mult in1 by COEFF
  c2      addu    cr0, pr0      -> c0/sd0   # add to in2
  .aib end
```

Figure 3: Scale code for mult_add function.

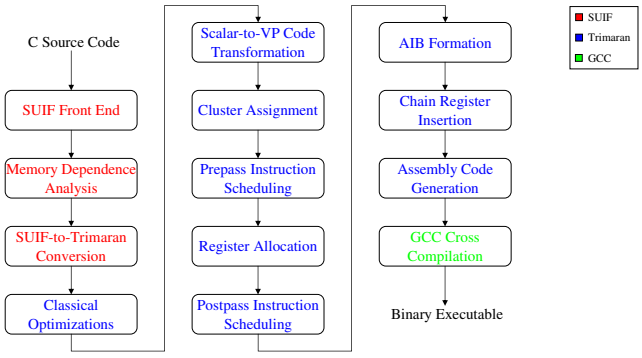

Figure 4: Scale compiler flow.

tions we consider in this work. The compiler uses Trimaran's existing infrastructure to detect all of the loop nests in the function under consideration. It then processes each loop nest in an attempt to find a loop that can be transformed. If a loop is successfully transformed, then its loop nest will not be modified any further.

To illustrate the details of this compiler phase, we will show how the function in Figure 5 is transformed. This example contains a loop nest that also has internal control flow. The inner loop has a loop-carried dependence due to the accumulation. Figure 6 shows the scalar code that is the input to the transformation phase on the left. The parallelized output code of the transformation phase is shown on the right. In the following sections, we describe how the transformation actually occurs.

Overview

As stated in Section 2, the typical approach to parallelize loops for VT architectures is to strip mine the loop so that

```
void example(unsigned int len, unsigned short * restrict in,
             unsigned char * restrict mask,
             unsigned int * restrict out) {
  unsigned int i, j, accum;
  for (i = 0; i < len; i++) {
    accum = out[i];
    for (j = 0; j < len-i; j++)
      accum += (in[j]*in[j+i]) >> SCALE;
    if (mask[i] == 0)
      accum >>= 1;
    out[i] = accum;
  }
}
```

Figure 5: C code example used to illustrate how the compiler works.

the control processor launches a group of VPs simultaneously, with each VP executing a single loop iteration. The basic blocks in the loop will be mapped to a combination of VTU commands (including any vector memory instructions), vector-fetched VP code, thread-fetched VP code, and scalar instructions—e.g. to process induction variables. The mapping used for a particular basic block depends on its type. Any given loop can be decomposed into basic blocks that can be grouped into four different categories, with the groups potentially overlapping so that a single block may be placed in multiple categories. First, there is a header block at the beginning of the loop. For the outer loop in Figure 6, the header block is block 1. There may also be a group of blocks that have a back edge to the header block. In Figure 6, block 6 is the single back edge block for the outer loop. Additionally, there may be a group of blocks that have a control-flow edge exiting the loop—block 6 is also an exit block for the outer loop. The final group consists of any remaining blocks in the loop that do not fall into the first three categories. This group consists of blocks 2, 3, 4, and 5 for the outer loop in Figure 6. An innermost loop with no conditionals has a single basic block, which is the header block, a back edge block, and an exit block—this is block 3 for the inner loop in Figure 6. Although Trimaran can handle more complex loop structures, we impose the restriction that a loop can only have a single back edge block and a single exit block, and that those blocks must be the same. The reason for this requirement is so that when VPs direct their own control flow by executing fetch instructions, they will always transfer control back to the control processor at the same point in the program. Otherwise, it would not be possible for the control processor to manage the VPs as a group, as it would need to handle return points individually. In practice, this restriction did not cause a problem, as the codes that we targeted were automatically mapped by the compiler to have a single back edge/exit block.

Another restriction on the compiler is that loops with data-dependent exit conditions—i.e. "while" loops—are not currently handled. One approach to deal with these loops is to speculatively execute iterations and to later nullify the effects of any VPs that should not have been launched. Future work will explore compilation of this type of code.

Figure 6 contains annotations between the scalar code and parallelized code describing the type of mapping that occurs. In the Scale compilation model, the header block

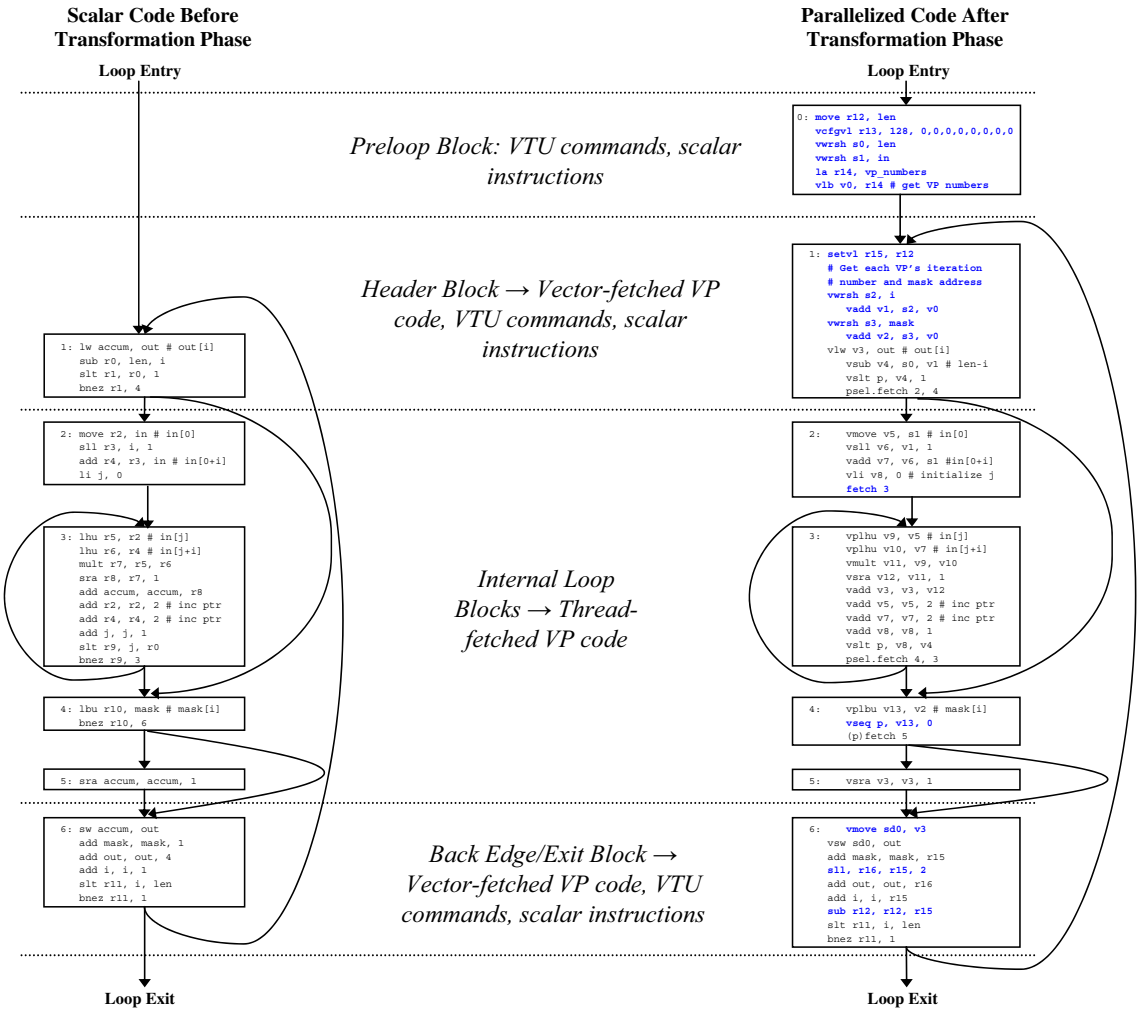

Figure 6: The control-flow graph on the left contains the scalar code that is the input to the transformation compiler phase. On the right is the parallelized output of the phase. All register numbers are virtual. The type of register depends on the prefix in front of the number: "r"=scalar; "v"=private VP; "s"=shared VP; "sd"=store-data VP. Certain scalar registers are referred to by their variable names for clarity. Highlighted instructions in the parallelized code are inserted by the transformation phase, as opposed to simply being converted from a scalar instruction. VP instructions that will be grouped into AIBs are indented to differentiate them from scalar instructions and VTU commands executed on the control processor.

and back edge block are transformed to a combination of vector-fetched VP code for non-memory operations, VTU commands including vector memory instructions, and scalar instructions to handle any induction operations as well as the spawning of new VPs. Since the control flow for the remaining loop blocks may differ from iteration to iteration, it is dependent on data that will be unique to a particular VP. Thus, those blocks are transformed to thread-fetched VP code, and each memory instruction is transformed to a VP memory operation within the appropriate block. A VP will continue to direct its own control flow until it reaches the back edge block, at which point it will halt.

As seen in Figure 4, the actual formation of AIBs takes place in a later phase of the compiler, when the directives to delimit the beginning and end of each AIB are inserted. This means that the compiler actually interacts with VP code at the level of a basic block. Only later, when the AIBs are actually formed, will a basic block's VP code in the intermediate representation be divided into one or more AIBs, as described in Section 3.5. The motivation for this phase ordering is twofold. First, since there is a size limit for AIBs (32 instructions per cluster for Scale), if the AIB boundaries were created before register allocation, there would be a potential for spill code insertion to cause the limit to be violated. Second, delimiting AIBs before instruction scheduling would impose restrictions on the scheduler, as all VP instructions in an AIB would have to be moved as a group relative to any instructions outside the AIB. This could hinder performance—for example, as discussed in Section 3.4, when processing several vector loads, it can be desirable to have each load immediately followed by any dependent VP computation. To avoid complicating the scheduling of VP instructions relative to non-VP instructions such as vector loads, the intermediate representation supports arbitrary

intermingling of both types of instructions within a basic block. The assembly code generator handles the actual separation of VP instructions into a distinct section of the file. For each AIB that is relocated into its own section from the header block or back edge/exit block, the assembly code generator also inserts a vector-fetch instruction in its original location.

Loop Selection

In traditional vectorizing compilers, selecting which loop in a nest should be vectorized typically depends on factors such as the structure of memory accesses or the loop trip count—which requires profiling to estimate dynamically determined trip counts. This can be a complicated process [5], and one that we have deferred for future work. Rather than attempting to account for various loop-specific factors, the compiler currently uses a simple heuristic of first working from the innermost to the outermost loop in the nest, searching for a DOALL loop that can be transformed. We attempt to transform innermost loops first, because the control processor is more efficient at handling branches than virtual processors. If no DOALL loops in a nest can be mapped to the vector-thread unit, then the compiler processes the loop nest again, attempting to map loop-carried dependences to the cross-VP network. Cross-VP communication is primarily designed to be used within a vector-fetched AIB, so the compiler first determines if the innermost loop has no internal control flow and if it can be transformed using cross-VP transfers. If that is not possible, the compiler restricts the vector length to equal the number of *independent VPs*, and again works from the innermost loop in the nest to the outermost loop. The number of independent VPs is the maximum number of VPs that can execute in parallel without having to be time-multiplexed. In the Scale implementation, this value is the number of lanes. Since using independent VPs removes the possibility a VP will be time multiplexed on a lane, it also removes the possibility of deadlock when using arbitrary cross-VP communication.

It should be noted that although Scale does not support explicit reduction operations, reductions can be handled by using the standard parallelization technique of computing partial results on each lane (in a shared register) and then merging the results during a final execution phase. However, we have not yet added compiler support for this approach, so currently the compiler treats all loop-carried dependences in the same manner when selecting which loop to transform. In Figure 6, the inner loop contains a loop-carried dependence on `accum`, so the compiler will first consider the outer loop to see if it can be parallelized.

Loop Transformation

To determine if a loop can be transformed, the compiler employs a modified version of the approach used for vector architectures by Allen and Kennedy [5]. The compiler first conducts a dependence analysis. Since a loop may contain internal control flow, the compiler processes each possible control flow path through the loop and updates the dependence graph accordingly. The SUIF-generated direction vectors for memory dependences are used to provide additional information such as potential cross-VP transfers. Once the dependence graph is constructed, the compiler uses Tarjan's algorithm to identify strongly connected components [32]. If there are any dependence cycles that occur in the loop currently being considered, it will not be transformed unless the cycle involves register dependences contained within an inner loop (as those will be local to each VP), or unless cross-VP communication is possible. Another requirement is that all memory operations that will potentially be transformed to vector memory instructions—i.e. those in the loop's header block or back edge block—must have a statically determinable stride value.

If the compiler determines that a loop meets the requirements for transformation, it sets up the code necessary for the scalar processor to control the execution of the loop. It creates a special preloop basic block that contains the configuration command for the VTU. This is block 0 in Figure 6's parallelized code. The command is updated after the chain register insertion phase with the actual configuration identifying the number of registers required in each cluster. The preloop block also sets up loop-invariant values, including constants, registers that are never written, and AIB addresses needed by VP fetch instructions. All of these values are written into shared registers using the `vwrsh` command. (For clarity, Figure 6 shows fetch instructions using the block labels rather than actual registers.) Besides the performance benefit of avoiding the computation of these values in each iteration, mapping the values to shared registers that can be used by multiple VPs also reduces the per-VP physical register requirements.

Another function of the preloop block is to load each VP's number—found within a special data block containing the numbers 0 through 127—into a private register if necessary. This is done if an induction variable will be used in an internal loop block, and each VP has to compute the value of that variable. For example, the value of `mask[i]` is loaded in block 4 of Figure 6. Since this is an internal loop block, a VP load instruction will be used, and the address register `v2` has to be set up by the control processor. This is performed in block 1 by having each VP take the baseline value of the `mask` address for that strip-mined loop iteration—i.e. the value needed by VP 0—and adding its own VP number.

Block 1 also contains the `setvl` command to set the vector length in each strip-mined iteration. The induction operations for the loop (found in block 6) are updated so that the induction value is multiplied by the vector length. Note that the loop counter increments from 0 to the trip count `len`. However, decrementing from `len` to 0 would be more appropriate when using the `setvl` instruction, which takes the number of iterations remaining as its source operand. In the example, the compiler adds another variable `r12` which decrements to 0, while keeping the separate counter `i` which increments to `len`. While this is simple, optimized code would merge the two variables.

There is generally a one-to-one mapping between scalar instructions and VP instructions, so transforming the code is mostly straightforward. Note that most of the opcode names in the parallelized code are the same as their scalar counterparts with a "`v`" added as a prefix to indicate a vector instruction. VP memory instructions have a "`vp`" prefix to distinguish them from vector memory commands. Branches are somewhat different for VPs than for a typical scalar processor. When a VP finishes executing a block of code, it will terminate unless it fetches another block. There is no notion of "falling through" to the next block of code. Thus, in block 2, an explicit `fetch` instruction is added to fetch block 3. In block 5, since there is no fetch instruction, the thread will

terminate and control returns to the control processor. To conditionally fetch a block, the predicate register has to be used. Each VP has a single private predicate register that can be used for control flow and for if-conversion [4]. Block 1 shows how a scalar conditional branch instruction is converted to a `psel.fetch` instruction which selects between two different blocks to fetch. In block 4, if the predicate is false, no fetch will occur and the VP will terminate.

Register values that are live within a single iteration are unique to each VP, so they are mapped to private registers. Although not shown in Figure 6, cross-VP values are mapped to a "dummy" register file, and are later converted to `prevVP`/`nextVP` names in the assembly code that correspond to the queues between each lane. The reason for using registers for these values is to ensure that the compiler can easily track the dependences between receive/send operation pairs. When generating code with cross-VP transfers, the compiler also sets up a VTU command in the preloop block for the control processor to push initial cross-VP values into the queue, and creates a postloop block containing commands that pop final cross-VP values from the queue after the loop ends.

3.3 Cluster Assignment

The compiler is responsible for assigning each VP instruction to a particular cluster. It uses a modified version of the clustering approach described in [11], which targets a typical VLIW architecture and tries to balance work among the clusters while still avoiding inter-cluster moves along the critical path. For a VLIW compiler, the focus is on improving ILP for a single thread. By contrast, Scale also exploits DLP and TLP, and although reducing per-thread latency is important in the compiler, it is typically more important to focus on the other forms of parallelism to improve processor throughput—thus reducing the overall latency of the program. Since the vector length depends on the cluster with the most severe register requirements, the Scale compiler prioritizes achieving a more balanced partitioning of instructions between clusters over minimizing the per-thread latency.

Additionally, to enhance decoupling—which also serves to hide latency—the compiler attempts to generate acyclic dataflow between the clusters, as a specific cluster's operations within an AIB have to be executed as an atomic group for each VP. The compiler also tries to place long-latency operations on clusters separate from the instructions which depend on the results. This is intended to take advantage of the decoupling between clusters. For example, if a multiply feeds an add within a single AIB, placing the multiply and add on separate clusters will allow the multiply latency to be hidden by the interleaved execution of independent VPs.

3.4 Instruction Scheduling

Typical instruction scheduling approaches try to reduce the critical path length of the section of code being analyzed. This frequently leads to long-latency instructions being scheduled early, while short-latency dependent instructions are scheduled later. As a result, register lifetimes are often lengthened by the scheduler, leading to increased register usage. By contrast, the primary goal of Scale's instruction scheduler is to minimize register usage by scheduling dependence chains together. This is done for two reasons. First, by reducing the number of registers used by each VP, the maximum vector length may increase, enabling greater processor throughput. Second, scheduling dependence chains together makes it more likely that the compiler will be able to explicitly target chain registers—the inputs to the ALU—potentially further reducing the register usage for each VP and also enabling more energy-efficient execution. As mentioned earlier, long-latency operations are usually placed on a different cluster than any dependent instructions, so Scale's use of decoupling helps to hide the latency from targeting dependence chains in this phase. We modified the traditional list scheduler used in Trimaran to attempt to group dependence chains together, and if no chaining opportunities exist, to revert to its standard approach.

3.5 AIB Formation

In this phase, the compiler processes each basic block and inserts directives around VP instructions that delimit the boundaries of AIBs. As mentioned previously, the actual separation of AIBs into a separate section of the file takes place during assembly code generation. There are several situations which will cause an AIB to be terminated. First, for each VP instruction the compiler determines whether the AIB size limit will be exceeded. Another reason to end an AIB is on a control processor instruction that has a dependence on one of the VP instructions in the current AIB. Finally, the end of a basic block will also terminate an AIB.

3.6 Chain Register Insertion

Once AIBs are formed, the compiler can map temporary values to chain registers, which are only valid within an AIB. The use of chain registers can reduce the physical register file resources required by each VP and thus potentially increase the vector length. To determine which values can be mapped to chain registers, the compiler constructs the live ranges for each register value, keeping a list of potential candidates. A value with a live range that crosses an AIB boundary cannot be mapped to a chain register. Also, every operation that executes on a particular cluster overwrites the values in that cluster's chain registers. If a potential chain register value would not be overwritten by another operation during its lifetime, it will be mapped to a chain register.

Note that it is possible for VP register spill code to be generated when first running register allocation. Performing chain register insertion during the initial allocation phase could reduce register pressure and lessen the possibility of registers being spilled. However, chain register insertion has to take place after AIB formation, and forming AIBs before running the register allocator for the first time could create a problem if spill code causes the AIB to overflow its size limit. An alternative strategy employed by the TRIPS compiler is to form blocks before register allocation, but to use iterative block splitting if spill code insertion causes a block size violation [31]. We intend to explore this possibility in future work.

4. EVALUATION

We evaluate our compiler implementation by using benchmarks from the EEMBC benchmark suite [1]. Table 1 contains a description of the benchmarks as well as the types of loops that were parallelized during compilation. The autocorrelation benchmark contains a loop nest in which the inner loop has a cross-iteration dependence. By default, our

compiler parallelizes it using outer-loop vectorization (listed as `autocor_olv` in the table). However, for the sake of comparison, we also configured the compiler to parallelize the inner loop by using Scale's cross-VP network (`autocor_xvp`). Aside from inserting the `restrict` keyword to indicate that pointers do not alias, the only other modifications we made to the source code involved changing the loops for `rgbcmy` and `rgbyiq` to use array accesses instead of pointer accesses. This is due to a limitation we imposed for our initial compiler development, that an induction variable could only be updated once within a loop iteration to simplify the computation of strides for vector memory accesses. Benchmarks `rgbcmy` and `rgbyiq` contain pointer variables that are incremented multiple times within the loop, inhibiting parallelization within our current setup. However, we intend to lift this restriction in future work to enable us to compile code without having to make any pointer-to-array conversions.

Since our compiler infrastructure is still early in its development, we are restricted in the variety of benchmarks that we are able to compile. It should be noted that we have not yet incorporated any established parallel transformations in the compiler front end. Also, we currently have certain artificial restrictions in place within the compiler that were inserted to ease the development and debugging process. As we lift those restrictions, the compiler will be able to handle more programs. Despite the limited number of benchmarks, a key point is that they represent a wide variety of loop types, including loops that would be non-vectorizable without significant transformations. Our initial focus was not on obtaining a vectorizable intermediate form from original source code, but rather on how to map the intermediate form to the vector-thread architectural features. Adding established front-end loop transformations would be straightforward and would support a larger set of benchmarks, but would not change the back-end code generation strategy. It should also be noted that we only include benchmarks that we can parallelize without programmer intervention, with the exceptions of the modifications discussed above. Previous work [17, 15] has shown the difficulties of automatically parallelizing EEMBC benchmarks, using Intel's compiler in an evaluation. The Intel compiler is unable to vectorize many of the innermost loops in EEMBC benchmarks for various reasons (although the use of the `restrict` keyword would likely help in certain cases). It can alternatively target outer loops, but this requires user assistance, such as OpenMP directives. Although this is only a single example, it serves to illustrate the point that the problems encountered in automatically parallelizing certain benchmarks are not unique to the Scale compiler.

Our compiler-generated code is evaluated on the Scale microarchitectural simulator. The simulator includes detailed models of the VTU and the cache, but uses a single-instruction-per-cycle latency for the control processor. The fact that control processor pipeline hazards are not modeled improves the performance of the baseline scalar code, thus providing us with a pessimistic evaluation of our speedups. All simulations in this work use the default Scale configuration of a VTU with 4 lanes, 4 clusters per lane, and 32 physical registers per cluster. Table 1 contains the speedup of the compiler-generated vector-thread code over scalar code running on the control processor. The scalar code is generated using the same compilation infrastructure described in Section 3, but without performing any scalar-to-VP code transformations.

The smallest speedup is about 3× for `fir`. This benchmark contains a loop nest with an inner loop that consists of only two accumulations. Since the compiler parallelizes the inner loop using cross-VP transfers, there is little computation to perform in parallel. Additionally, two of the four VP instructions in the parallelized benchmark are long-latency multiplies, further limiting the speedup. At the other extreme, `hpg` has a large speedup of 33×. However, there is one caveat with this result. In general, we observed that when generating scalar code, the compiler infrastructure used in this work (SUIF-to-Trimaran-to-GCC) is competitive with the GCC-only approach used by Krashinsky et al. [19], and in some instances is superior. However, for `hpg`, our compiler generates code that contains a significant number of register spills, and this code is slower by about a factor of 3 than using GCC alone. For the sake of consistency, we used the same baseline compiler infrastructure for all benchmarks, but even when compared against the faster GCC-generated scalar code, we still obtain about an 11× speedup for `hpg`. `rgbyiq` has a speedup of about 26× regardless of the baseline scalar code used. This is a simple DOALL loop that the compiler can handle in a straightforward manner. For `autocor`, the use of the cross-VP network produces a superior speedup to outer-loop vectorization. This is not inherent to the program—with hand-coded assembly, the outer-loop vectorized version is superior—but due to inefficiencies in our compiler infrastructure when generating code with thread fetches. By contrast, the inner-loop vectorized version of `autocor` that uses the cross-VP network only has a single basic block to parallelize, which is simpler for the compiler to handle. As we further develop the compiler and integrate other optimizations such as if-conversion, the efficiency of outer-loop vectorization should increase.

An interesting comparison can be made between the results for Scale and the limit study on thread-level speculation conducted by Islam et al. [15]. TLS designs are somewhat more flexible than Scale in the types of codes they can handle, as they typically have hardware support to squash speculative threads that have violated dependences. By contrast, we are still exploring the possibility of generating code that can support speculative VPs, so we have to be conservative in our approach to handling dependences between threads. In [15], the speedup over a single-issue scalar processor is computed for a variety of multicore TLS configurations. The benchmarks consist of the EEMBC consumer and telecom suites, which includes `autocor`, `hpg`, `rgbcmy`, and `rgbyiq`. For an ideal TLS machine with an infinite number of cores and zero thread-management overhead, the speedups are approximately 5× for `autocor`, 19× for `hpg`, 14× for `rgbcmy`, and 17× for `rgbyiq`. Note that for `autocor` and `rgbyiq`, the actual speedups achieved by the Scale compiler using a realistic model for the vector-thread unit exceed the upper bound on the TLS speedups. (We do not include `hpg` for the reason mentioned earlier.) For a 16-core machine with no thread-management overhead (which matches the issue capability of Scale's VTU but is idealized with respect to latency), the speedups are approximately 5× for `autocor`, 9× for `hpg`, 8× for `rgbcmy`, and 9× for `rgbyiq`. Obviously our setups for the evaluation are somewhat different; still, this comparison attests to the performance improvements made possible by various Scale features, such as

Benchmark	Description	Loop Type	Avg. Vector Length	Speedup
autocor_olv	Fixed-point autocorrelation	DI	16.6	7.3
autocor_xvp	Fixed-point autocorrelation	XI	60.6	10.1
fir	Finite impulse response filter	XI	35.0	3.3
hpg	High pass grey-scale filter	DP	63.6	33.0
rgbcmy	RGB to CMYK color conversion	DC	20.0	7.2
rgbyiq	RGB to YIQ color conversion	DP	28.0	26.0

Table 1: Benchmark descriptions, types of loops that were parallelized, average vector length, and speedups over scalar code. The different loop types are taken from [19]: [DP] data-parallel loop with no control flow, [DC] data-parallel loop with conditional thread-fetches, [XI] loop with cross-iteration dependences, [DI] data-parallel loop with inner loop. For `autocor`, the `data3` dataset was used.

vector memory instructions and decoupled execution. The Scale chip prototype provides evidence that a 16-issue VT architecture has much lower area and power consumption than a 16-core multiprocessor [21].

Figure 7 shows how the optimizations performed in various compiler phases affect the speedups. "No Opt" refers to simply performing the scalar-to-VP code transformation: the clustering used is the approach described in [11] with none of the Scale-specific modifications described in Section 3.3. The standard Trimaran list scheduler is used with no attempt to schedule dependence chains together, and no chain registers are used. We then enable the various optimizations to determine their impact. In certain cases, turning on optimizations does not affect the average vector length, as shown in Figure 8, and thus performance is relatively unchanged. For some benchmarks, even when the vector length is increased, it does not help performance. For example, although scheduling dependence chains together doubles the vector length for `fir`, it does not increase resource utilization, as the baseline code already spawns a sufficient number of VPs to maximize performance for a particular cluster assignment.

Performing Scale-specific clustering provides significant benefits for some benchmarks, and is inconsequential in others. We noted that even when turning on our cluster optimization, the compiler sometimes packed the majority of operations within a single cluster, thus inhibiting decoupling. This indicates that we need to spend more time improving our algorithm. Dependence chain scheduling provides small benefits for `rgbcmy` and `rgbyiq`, but has a negative impact on `hpg` in spite of the fact that it increases the average vector length. (We have not yet pinpointed the reason for this decrease, although `hpg` has a more complex memory access pattern than the other benchmarks, so that could be a factor.) However, for the remaining benchmarks, dependence chain scheduling had little impact, either due to no effect on the vector length, or due to resource utilization already being maximized as mentioned above. Chain register insertion also did not typically affect the vector length for the particular benchmarks we evaluated. This resulted in little or no impact on performance, in spite of the fact that in some cases a large percentage of the register accesses used chain registers (which is still useful to save energy). We are continuing to refine the compiler's cluster assignment and instruction scheduling phases, which impact each other's effectiveness as well as the effectiveness of chain register insertion. Incorporating more advanced techniques in the scalar-to-VP code transformation phase—for example, when selecting loops to parallelize—should also improve our results.

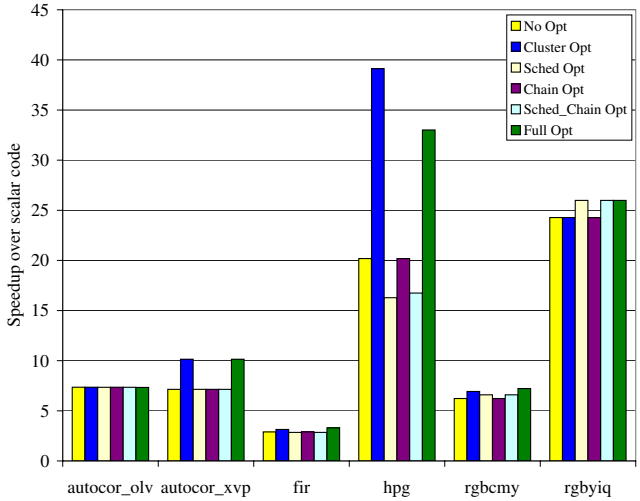

Figure 7: Comparison of the various speedups of the compiler-generated code when different optimizations are turned on. No Opt = No optimizations; Cluster Opt = Scale-specific clustering; Sched Opt = Schedule dependence chains together; Chain Opt = Target chain registers; Sched_Chain Opt = Schedule dependence chains together and target chain registers; Full Opt = Turn on all optimizations.

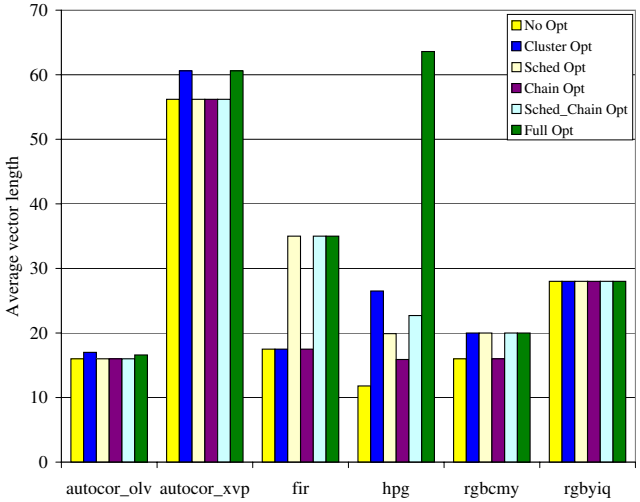

Figure 8: Comparison of the average vector lengths when different optimizations are turned on.

Figure 9 provides a comparison between the speedups of the compiler-generated code and the speedups of the hand-written assembly code. This comparison shows that there is still significant room for improvement in our compiler infrastructure. The hand-coded benchmarks have been highly optimized and take advantage of algorithmic transformations as well as Scale features that we do not yet support such as segment-strided memory accesses. As we further develop our compiler, we expect to narrow the performance gap.

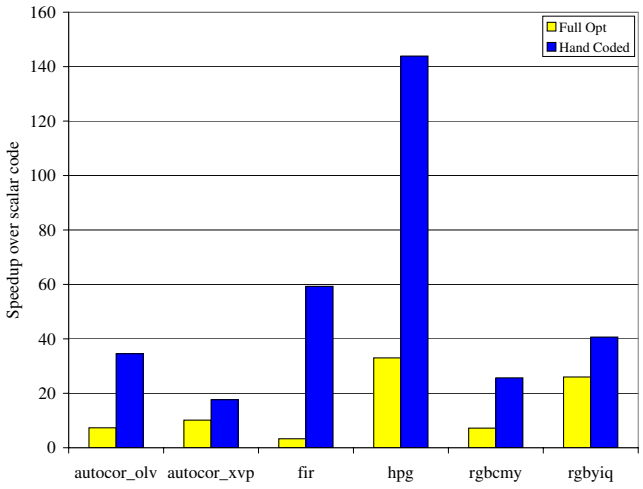

Figure 9: Comparison of the speedups of the compiler-generated code with all optimizations turned on and the hand-coded assembly.

5. RELATED WORK

The TRIPS architecture [27] can also exploit multiple forms of parallelism. However, unlike Scale, which exploits DLP, TLP, and ILP simultaneously, TRIPS only targets one form of parallelism at any given time and explicitly "morphs" between modes of execution. Speculation is used to find parallelism. The TRIPS compiler focuses on forming blocks full of useful instructions [31] and mapping instructions to ALUs [25, 12]. The Scale compiler has somewhat different priorities, as it deals with issues more typically encountered in vectorization, and it also focuses on non-speculative execution.

Stream processing [18] targets DLP by mapping computation kernels—typically the body of a loop—to a Kernel Exection Unit (KEU), which is a co-processor that contains clusters of ALUs. ILP can also be exploited within each kernel. While this approach shares some similarities with Scale, there are some significant differences between the Scale compiler and the stream processing compiler [13]. The KEU can only communicate with memory through a Stream Register File (SRF). A major job of the compiler is to manage the utilization of the SRF. By contrast, this is much less of a concern for the Scale compiler due to Scale's cached shared memory model and decoupled cache refills [7]. Additionally, the stream processing compiler performs a binary search to determine the best strip-size when strip mining, while this is not an issue for Scale. Finally, the stream processing compiler uses a standard list scheduling approach to reduce latency of a kernel, while the Scale compiler focuses more on improving throughput as the Scale hardware provides decoupling to tolerate latency.

Eichenberger et al. [14] describe the compiler for the Cell processor, which exploits multiple forms of parallelism. They provide multiple programming models for Cell depending on how involved the programmer wants to be with the low-level details of the architecture. The paper focuses on many issues which are not present in Scale: Cell uses SIMD functional units for data-level parallelism, so alignment is a significant concern; the Synergistic Processor Elements (SPEs) have local non-coherent memories, so the compiler has to handle transfers between the system memory and the local memories using DMA commands; the SPEs have no branch prediction and assume all branches to be not-taken, so the compiler has to insert branch hints and schedule them appropriately. The Scale compiler has different priorities due to the significantly different architecture.

The XIMD architecture [35] extends a VLIW architecture by providing an instruction sequencer for each functional unit. This allows the XIMD to behave like a multithreaded machine, while still retaining the ability to function like a VLIW. The compiler schedules loops for the XIMD by using a technique called *iteration mapping* [26], which attempts to balance the exploitation of fine- and medium-grained parallelism in order to fully utilize processor resources. Since Scale hardware time-multiplexes VP threads onto the physical resources and makes extensive use of decoupling, there is less of a burden on the compiler to fully utilize resources, although the cluster assignment phase can have a significant impact.

Microthreading [16] shares some similarities with VT. A microthread executes a single loop iteration and its execution can be interleaved with that of other microthreads sharing the same physical resources. The compilation infrastructure required for microthreading is discussed in [8], although no actual compiler implementation is presented. It is the compiler's responsibility to insert *swch* commands to force a context switch after issuing loads which could miss in the cache. This is done to prevent potential stalls. The Scale compiler aims for a similar goal by attempting to place VP operations which depend on loads on a different cluster than cluster 0, so that decoupling will be possible.

Multi-threaded vectorization [9] tries to span the gap between vector and VLIW machines in order to target code that would not be traditionally vectorizable. The compiler starts with a schedule that has been software-pipelined and builds on that to determine the issue time and issue period for each vectorized instruction. However, the compiler has to know the functional unit latencies, and the work is restricted to a single vector lane.

Tian et al. [33] exploit parallelism at both the thread-level and instruction-level on an Intel platform that supports Hyper-Threading Technology. The Intel compiler can perform parallelization that is guided by OpenMP directives or pragmas. It also supports automatic loop multithreading to exploit medium-grained parallelism. For automatic parallelization, the compiler builds a loop hierarchy structure, performs data dependence analysis to find loops without loop-carried dependences, and determines whether it will be profitable to generate multithreaded code for the loop. The compiler can also perform "intra-register vectorization" by generating SIMD instructions. This technique can be combined with the above multithreading approaches. Scale's

advantages include the ability to handle loop-carried dependences, and more efficient parallelization of loops, especially with respect to vector memory operations and the spawning of threads.

Superword-level parallelism (SLP) [22] can be used to improve performance in machines that support SIMD operations. Shin et al. [30] discuss how SLP can be applied even in the presence of control flow by using if-conversion. The disadvantage of this approach is that all possible control paths have to be executed. This is addressed with the concept of generating *branches-on-superword-condition-codes* (BOSCCs) [29, 28]. BOSCCs allow a vector instruction or group of vector instructions to be bypassed if the guarding vector predicate contains all false values. Scale is more flexible in its handling of control flow because each VP thread—which corresponds to a single superword element in the BOSCC approach—can direct its own control flow without being tied to the other threads.

6. CONCLUSION

Vector-thread architectures can exploit multiple forms of parallelism simultaneously, making it possible to target a wide variety of applications. However, this is dependent on the programmer or compiler being able to exploit the features of the architecture. In this paper, we have presented a compiler for the Scale vector-thread architecture. The compiler can parallelize several types of loops, and it is able to achieve significant speedups over a single-issue scalar processor. We are continuing to improve the compiler and incorporate techniques that will enable us to increase performance further and also support a greater number of applications.

7. REFERENCES

[1] EEMBC. http://www.eembc.org/.

[2] GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[3] Scale Home Page. http://www-ali.cs.umass.edu/scale/.

[4] J. R. Allen et al. Conversion of control dependence to data dependence. In *POPL-10*, pages 177–189, January 1983.

[5] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers, 2001.

[6] K. Asanović et al. Energy-exposed instruction sets. In *Power Aware Computing*, chapter 5. Kluwer Academic/Plenum Publishers, June 2002.

[7] C. Batten et al. Cache refill/access decoupling for vector machines. In *MICRO-37*, pages 331–342, December 2004.

[8] T. Bernard et al. A microthreaded architecture and its compiler. In *Proceedings of the 12th International Workshop on Compilers for Parallel Computers*, pages 326–340, January 2006.

[9] T. c. Chiueh. Multi-threaded vectorization. In *ISCA-18*, pages 352–361, May 1991.

[10] L. N. Chakrapani et al. Trimaran: an infrastructure for research in instruction-level parallelism. *Lecture Notes in Computer Science*, 3602:32–41, 2005.

[11] M. Chu, K. Fan, and S. Mahlke. Region-based hierarchical operation partitioning for multicluster processors. In *PLDI 2003*, pages 300–311, June 2003.

[12] K. Coons et al. A spatial path scheduling algorithm for EDGE architectures. In *ASPLOS-12*, pages 129–140, October 2006.

[13] A. Das, W. J. Dally, and P. Mattson. Compiling for stream processing. In *PACT-15*, pages 33–42, September 2006.

[14] A. E. Eichenberger et al. Optimizing compiler for the CELL processor. In *PACT-14*, pages 161–172, September 2005.

[15] M. M. Islam et al. Limits on thread-level speculative parallelism in embedded applications. In *INTERACT-11*, pages 40–49, February 2007.

[16] C. Jesshope. Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines. In *Proceedings of the 6th Australasian Conference on Computer Systems Architecture*, pages 80–88, January 2001.

[17] A. Kejariwal et al. Challenges in exploitation of loop parallelism in embedded applications. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pages 173–180, October 2006.

[18] B. Khailany et al. Imagine: media processing with streams. *IEEE Micro*, 21(2):35–46, March/April 2001.

[19] R. Krashinsky et al. The vector-thread architecture. In *ISCA-31*, pages 52–63, June 2004.

[20] R. Krashinsky et al. The vector-thread architecture. *IEEE Micro*, 24(6):84–90, November 2004.

[21] R. M. Krashinsky. *Vector-thread architecture and implementation*. PhD thesis, Massachusetts Institute of Technology, June 2007.

[22] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI 2000*, pages 145–156, June 2000.

[23] S. Larsen, R. Rabbah, and S. Amarasinghe. Exploiting vector parallelism in software pipelined loops. In *MICRO-38*, pages 119–129, November 2005.

[24] D. B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24(1):121–145, January 1977.

[25] R. Nagarajan et al. Static placement, dynamic issue (SPDI) scheduling for EDGE architectures. In *PACT-13*, pages 74–84, September–October 2004.

[26] C. J. Newburn, A. S. Huang, and J. P. Shen. Balancing fine- and medium-grained parallelism in scheduling loops for the XIMD architecture. In *Proceedings of the IFIP WG10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 39–52, January 1993.

[27] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *ISCA-30*, pages 422–433, June 2003.

[28] J. Shin. Introducing control flow into vectorized code. In *PACT-16*, September 2007.

[29] J. Shin, M. Hall, and J. Chame. Evaluating compiler technology for control-flow optimizations for multimedia extension architectures. In *6th Workshop on Media and Streaming Processors*, December 2004.

[30] J. Shin, M. Hall, and J. Chame. Superword-level parallelism in the presence of control flow. In *CGO 2005*, pages 165–175, March 2005.

[31] A. Smith et al. Compiling for EDGE architectures. In *CGO-4*, pages 185–195, March 2006.

[32] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, June 1972.

[33] X. Tian et al. Exploiting thread-level and instruction-level parallelism for Hyper-Threading Technology. *Intel Developer Update Magazine*, January 2003.

[34] R. P. Wilson et al. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[35] A. Wolfe and J. P. Shen. A variable instruction stream extension to the VLIW architecture. In *ASPLOS-4*, pages 2–14, April 1991.