Exploring the Limits of Sub-word Level Parallelism

Kevin Scott and Jack Davidson
University of Virginia, Charlottesville, VA
{jks6b,jwd}@cs.virginia.edu

Abstract

Multimedia instruction set extensions have become a prominent feature in desktop microprocessor platforms, promising superior performance on a wide range of floating-point and integer signal processing, multimedia, and scientific applications. But the question remains whether or not these multimedia extensions can be applied to improve the performance of general, integer intensive applications. The answer to this question is important and could be used to direct research and development of compiler algorithms and refinements to multimedia architectures. In this paper we answer the question of whether integer programs exhibit enough sub-word level parallelism (SLP) to facilitate performance improvements through use of multimedia extensions. Using a highly optimizing compiler and a simulator for an aggressive SLP architecture, we measured available SLP in a range of integer benchmarks. Our measurements show that these applications exhibit significant levels of SLP. Using the most aggressive simulator settings, dynamic instruction count reductions of 17 to 36 percent were observed. However, detailed examination of the data indicates that much of this parallelism is equivalent to instruction-level parallelism (ILP) and could just as easily be exploited by a traditional ILP architecture. Our findings indicate that researchers should focus their efforts on exploiting SLP in floating-point intensive and multimedia applications.

1. Introduction

Most microprocessors used by desktops, workstations, and servers have multimedia extensions that can be used to improve the performance of applications exhibiting at least some degree of fine-grained SIMD parallelism. These multimedia extensions provide integer and floating-point instructions that operate simultaneously on several data elements packed into a single word. We call these packed data elements sub-words; accordingly, we use the term sub-word level parallelism[1] (SLP) to refer to the fine-grained SIMD parallelism that these extensions were designed to exploit.

While multimedia extensions can provide impressive performance increases on hand-tuned applications, the more general problem of automatically leveraging SLP across a broad range of integer and floating-point applications remains open. For certain types of programs (e.g., vectorizable integer and floating-point codes), the outlook is promising. Recent research at MIT has produced significant speedups on single-precision floating-point intensive codes by using a simple basic block SLP compilation technique [10].

However, it remains unknown whether there is sufficient SLP in integer programs to effectively use SIMD instruction set extensions, and what the necessary compilation techniques are if the SLP exists. Even if significant SLP exists in a wide range of programs, it must be in a form that makes it clearly preferable to ILP execution on wide-issue processors in terms of performance, power, complexity or some other appropriate metric.

This paper addresses the question of whether or not SLP that can be effectively exploited by existing multimedia extensions exists in typical integer applications. We have simulated multimedia and integer benchmark programs from SPECint95 and MediaBench to determine the degree and character of available SLP. Our simulation study in many ways emulates the early ILP studies performed by Wall [8, 20, 21]. We propose a hypothetical SLP machine which can dynamically synthesize SIMD instructions from an executing stream of Alpha EV56 instructions [1]. From these simulations we gather statistics such as dynamic instruction count reduction due to SIMD instruction formation, SIMD instruction mixes, and the type and sizes of synthesized SIMD operands.

1. Larsen and Amarasinghe [9] have used the term superword level parallelism in their work to refer to the same type of parallelism. We prefer the term subword level parallelism as used by Lee [12] as it more accurately reflects the fact that sub-word length operands are the atomic units being combined to realize speedup.

Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques
0-7695-0622-4/00 $10.00 © 2000 IEEE

Our results indicate that while integer applications can realize significant dynamic instruction count reductions due to SIMD execution, most SIMD instructions were formed from only one or two Alpha instructions, resulting in SLP that very much resembles ILP. While this is true of most of the benchmarks that we studied, a few of the multimedia codes do show impressive instruction count reductions and achieve this with SIMD instructions that have no ILP equivalents. Hopefully this finding will promote further investigation on the nature of sub-word level parallelism and focus research on developing tools that can uncover and exploit the available SLP in the types of programs where it is present.

The remainder of the paper is organized as follows. Section two provides background on multimedia instruction set extensions and SLP. Section three discusses the method we used to measure SLP and section four presents our results. Section five presents related work and section six presents our conclusions and suggests future directions for studies of SLP.

2. Background

2.1. Multimedia Instruction Set Extensions

Computer architects have long recognized that SIMD (single instruction, multiple data) computation can be an effective way to achieve high performance. SIMD architectures have typically been realized as large scale multiprocessors and date back to the early 1960's and the Illiac IV [3]. The fundamental idea behind SIMD architectures has remained unchanged throughout four decades and across dozens of research and commercial machines; a single instruction operates on many data elements at once using many functional units, perhaps located on many different processors.

Recently microprocessor vendors have applied the idea of SIMD computation to instruction set extensions and processor organizations specifically designed to improve the performance of certain multimedia applications. Among these multimedia extensions are Intel's MMX and SSE [14, 18], AMD's 3DNow! [13], Sun's VIS [19], Motorola's Altivec [7], HP's PA-MAX2 [12], and Digital's Alpha MVI [15].

A typical multimedia extension provides SIMD instructions that operate on 2 to 16 sub-word operands or operand pairs simultaneously. Most multimedia extensions include SIMD instructions for integer arithmetic, logical operations, and data movement. Some multimedia extensions, such as Intel SSE and Motorola Altivec, provide SIMD operations for single-precision floating point arithmetic. Multimedia extensions often contain application specific SIMD instructions. For instance most multimedia extensions include a SIMD instruction designed specifically to improve the performance of MPEG2 motion estimation for video playback.

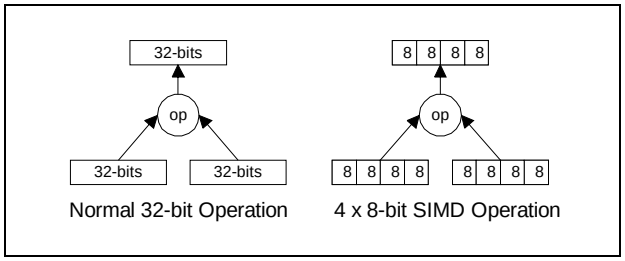


Figure 1: Normal vs. SIMD instructions.

In Figure 1 we depict the difference between a normal 32-bit operation and a 4 x 8-bit SIMD operation. In the normal case, an operation is performed on two 32-bit source registers and the result is placed in a 32-bit destination register. In the SIMD case each of the 32-bit SIMD registers is partitioned into four 8-bit sub-word elements. An operation is performed on each pair of sub-word elements and placed into a partitioned SIMD destination register.

It is the responsibility of the programmer or compiler to place sub-word data into SIMD source registers in the correct order and to extract sub-word data (if necessary) from the SIMD destination register. If the data on which the SIMD instructions operate are arranged correctly in memory, a single SIMD load or store may be all that is required to get data into or out of SIMD registers. If data is in the wrong order in memory, or is coming from non-SIMD registers, then some data manipulation will be required to correctly pack and unpack the SIMD registers.

In the example above we can consider 4 x 8-bit to be the *type* of the SIMD operation. For most fine-grained SIMD architectures, including commercial processors with multimedia extensions, and the research vector microprocessors from Berkeley [2] and Toronto [11], the types of SIMD operations are explicitly encoded in the instruction stream. The burden then is on software and/or the programmer to uncover the fine-grained SIMD parallelism, or sub-word level parallelism (SLP), and to produce the appropriate code to exploit it on one of these architectures.

One alternative to compile-time exploitation of SLP has been proposed by Brooks and Martonosi [5]. They describe an architecture which dynamically examines operand widths and uses this information to combine instructions in the issue queue into SIMD operations. The Brooks/Martonosi architecture handles integer arithmetic and logical operations, but excludes floating point and memory operations. Using this dynamic approach to SIMD operation synthesis, instructions operating on data whose bit-width requirements change over the course of execution may still benefit from SIMD execution. Using detailed micro-archi-

tectural simulation, Brooks and Martonosi demonstrated 4.3-10.4% speedups using dynamic SIMD instruction synthesis.

2.2. SLP vs ILP

At first it might not be obvious what advantages fine-grained SIMD architectures have over typical wide-issue processor organizations. If *n* operations can be executed in SIMD parallel fashion, why not execute the *n* operations on *n* functional units of a superscalar or VLIW machine? There are two good answers to this question.

Interconnect design is one of the most challenging problems in the engineering of modern processors. More interconnect means higher power consumption, larger die areas (and hence higher manufacturing costs), potentially longer cycle times, and more constraints in general on placement of other processor components [4]. Functional units in most processors are designed to accommodate operands as large as the native machine word size. If the native machine word size is *m*, and the processor executes *n* operations simultaneously on *n* functional units, *n x m* wires are needed to communicate each operand from the register file to the functional units. If it turns out that the operands only required *k* bits, then *n x (m - k)* interconnect lines were unused per operand. If on the other hand we can pack *n k*-bit operands into a word-sized register and use these packed registers as operands to SIMD instructions, then no interconnect is wasted. This more effective use of interconnect can be used either to reduce the amount of interconnect required on a processor, perhaps lowering cost and power consumption, or it can be used to increase the number of simultaneous operations possible, potentially improving performance.

Fine-grained SIMD architectures with explicit encoding may also make more effective use of the instruction stream and register file. For *n* operations to execute on a modern wide issue-processor, each of the *n* instructions must be fetched, decoded, have its operand registers renamed, and entered into a buffer from which it will be issued to the execution units. A similar SIMD instruction performing *n* simultaneous operations on packed registers requires $1/n^{th}$ the issue bandwidth and as few as $1/n^{th}$ the renaming registers. Note that the Brooks and Martonosi architecture is practically the same as an ordinary wide-issue processor through the issue stage [5]. Accordingly their architecture would not make any more effective use of the instruction stream or register file than would any other wide-issue processor.

Fine-grained SIMD architectures do have their drawbacks when compared with general-purpose wide-issue organizations. Wide-issue processors perform a substantial amount of dynamic optimization based on runtime behavior of a program. In particular, wide-issue processors are able to use dynamic scheduling and register renaming to hide operation latencies and to achieve higher functional unit utilization than would otherwise be possible through static optimization. Modern wide-issue processors are also able to accurately predict branch outcomes and speculatively execute along predicted paths. The current crop of fine-grained SIMD architectures have no hardware mechanisms for dynamically improving fine-grained SIMD execution.

3. Measuring Sub-word Level Parallelism

Implicit parallelism in source code and instruction streams have allowed compiler writers and computer architects to make impressive advances in high-performance computer architectures and microprocessors. Instruction-level parallelism (ILP) for example, is the driving force behind wide-issue processor organizations. A compiler may be able to recognize parallelism statically and produce code that exposes that parallelism to the microprocessor by careful arrangement of instructions. The processor may be able to detect further parallelism in an executing instruction stream, and perform additional transformations to expose that parallelism to the execution pipelines.

ILP is a measure of independence among the instructions of a program. This independence itself is the property that enables simultaneous execution of a program's instructions. We can also define instruction-level parallelism in terms of an abstract wide-issue machine that we give special capabilities, like unbounded register files and unit latencies for all operations [20]. By simulating execution of programs on this abstract machine we can study the effects of hardware and software approaches and trade-offs for improving ILP.

Sub-word level parallelism (SLP) is another type of implicit parallelism in programs. A program must exhibit SLP in order to achieve high performance on fine-grained SIMD architectures. This description yields very little insight into the nature of SLP. Larsen and Amarasinghe have asserted that SLP is just a more restricted form of ILP [9]. In addition to being independent, instructions to be combined into SIMD instructions must be of the same type. If combining instructions into a SIMD load or store, the memory elements loaded or stored must be contiguous. But this is not a complete description of what SLP is.

The degree to which instructions can be combined together to form SIMD instructions is affected by the width of their operands. For instance we may only be able to combine several add instructions when their operands have the same width. So determining the narrowest possible operand widths for instructions is necessary to maximize

SLP. This does not necessarily fall under the rubric of constrained ILP.

Our goal in this paper is to characterize the available SLP in programs. To do this we simulate execution of code compiled with the DEC C compiler on an abstract SLP machine. In the remainder of this section we describe our reference SLP architecture and the benchmark programs that we simulated.

3.1. Reference SLP Architecture

Our reference SLP architecture (SLP-REF) is based on the Alpha EV56 ISA and implemented in SimpleScalar 3.0 [6]. Alpha instructions are dynamically combined to form SIMD operations. SLP-REF's SIMD instruction set includes load, store, integer arithmetic (including 32-bit multiplies), floating-point arithmetic (including single-precision multiplies), logical, and comparison operations. SLP-REF does not have SIMD instructions for integer or floating-point division, double-precision floating-point multiplication or arithmetic operations such as square root. SLP-REF also does not provide application-specific SIMD instructions like sum of absolute differences (useful for MPEG motion estimation) or saturating integer arithmetic. SLP-REF supports packing of up to 128-bits worth of sub-word operands into a single SIMD register.

Figure 3 is a depiction of the organization of SLP-REF. The SimpleScalar *sim-safe* functional simulator was modified to gather traces of Alpha instructions with branch outcomes, register values and addresses fully resolved. Executed instructions along with their actual operand values are entered in order into a staging queue. As instructions are entered into the staging queue, Alpha registers are renamed to SIMD sub-word registers based on actual bit-width requirements of the register values. Renaming achieves two effects: it removes antidependencies and it uses the actual operand value to choose the narrowest sub-word register width that is still large enough to hold the value. Possible sub-word widths are 8, 16, 32, and 64-bits. When the staging queue fills, control is transferred to code which simulates the issue of SIMD instructions on SLP-REF.

SLP issue works by moving operations in order from the staging queue and placing them in a queue of pending SIMD instructions. A new instruction can be combined with a partially-packed pending instruction if a compatible pending instruction can be found. If no compatible instruction is found, the new instruction is placed at the end of the pending queue. As each instruction is added a search is conducted, beginning with the last scheduled instruction, to find a conflicting instruction. An instruction conflicts with the instruction being added if there is a register or memory carried data dependence between them. The new instruction can't be placed into the pending queue earlier than the first conflicting instruction found. If the conflicting instruction participates in a true dependence with the new instruction then we can place the instruction no earlier than the instruction following the conflict.

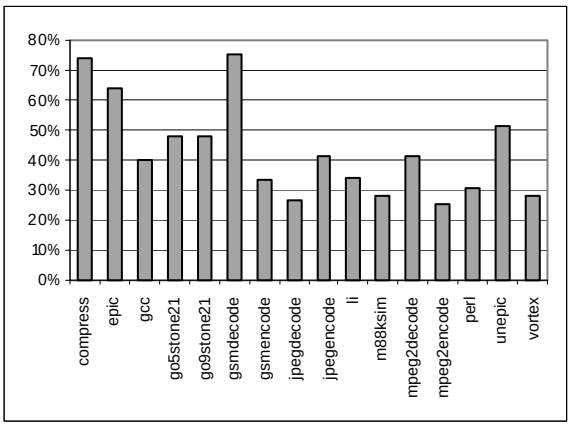


Figure 2: Percent of SIMD candidate instructions combined past control flow boundaries

Once a conflicting instruction is found, we search the instructions following the conflict for an appropriate place to put the new instruction. We can combine the new instruction with a partially packed SIMD instruction under the following circumstances:

- The partially-packed instruction is the same type of operation as the new instruction, e.g., both are adds.
- The partially-packed instruction has room in its SIMD operand registers to accommodate the addition of the new instruction sub-word registers.
- Loads and stores can only be combined if the resulting SIMD load or store is to a contiguous chunk of memory.

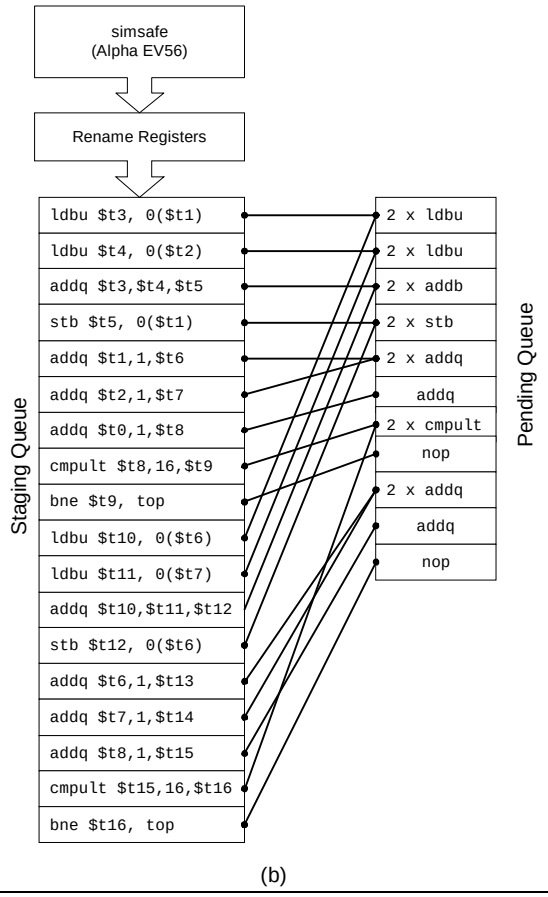
For instructions meeting these criteria, we keep track of which instruction would be brought closest to fully packed by the addition of the new instruction. When the search reaches the end of the pending queue, we place the new instruction into this "best" partially-packed instruction. If no compatible partially-packed instruction is found, the new instruction is placed at the end of the pending queue.

This process is repeated until the instructions are exhausted from the staging queue, or until we can add no more instructions to the pending queue. If we can add no more instructions to the pending queue, it is emptied and the process continues. When the staging queue is empty, we return control to SimpleScalar.

There are several issues worth pointing out about SLP-REF. Optimally combining instructions in the pending queue is an NP-hard online bin packing problem. The tech-

```
while(not staging_queue.empty())
{
  insn = staging_queue.get_next_instruction();
  best_cycle = NONE;
  if (is_slp_candidate(insn))
  {
    conflict_cycle = pending_queue.find_conflict(insn);
    for cycle = conflict_cycle to pending_queue.last_cycle()
    {
      if (cycle.will_hold(insn))
        if (cycle.free_slots() < best_cycle.free_slots())
          best_cycle = cycle;
    }
  }
  else
  {
    best_cycle = pending_queue.next_empty_cycle();
  }
  if (best_cycle == NONE) {
    if (pending_queue.full())
    {
      pending_queue.flush();
      best_cycle = pending_queue.first_cycle();
    }
    else
    {
      best_cycle = pending_queue.next_empty_cycle();
    }
  }
  best_cycle.add_instruction(insn)
}
```

(a)



(b)

Figure 3: Organization of SLP-REF. A modified *sim-safe* collects traces of Alpha EV56 instructions. Register operands are sized, renamed and entered into a staging queue. The pseudo-code describes how instructions are removed from the staging queue and placed in the pending queue to create SIMD instructions.

nique that we've used is a greedy approximation to an optimal solution. A better approximation or an optimal solution might lead to more combined instructions.SLP-REF allows SIMD operands to be synthesized from sub-word registers regardless of the order in which the sub-word registers were loaded from memory. In existing fine-grained SIMD architectures it is expensive to rearrange the sub-words in a SIMD register once it is loaded.

All simulations conducted with SLP-REF use perfect branch prediction since we know all branch outcomes at the point of SIMD operation packing. This was consistent with our goal to determine the "ideal" amount of SLP in a program. If we were actually designing hardware or software to do SIMD instruction packing, we would have to take control flow operations into account. Figure 2 shows the actual percentage of SIMD-candidate instructions that were combined past the boundaries of some control flow operation. On average 43% of the instructions combined into SIMD instructions were combined past a control flow boundary. This indicates that compilers generating SIMD code, or architectures dynamically synthesizing SIMD operations would unlikely be able to achieve the dynamic instruction count reductions that we achieve using perfect branch prediction (see Section 4.)

The choice of Alpha EV56 as an ISA to simulate has a definite impact on the results of our experiments. Since Alpha is a 64-bit architecture, certain address computations will require 64-bit arithmetic despite the fact that none of our benchmarks require a 64-bit address space.

Our goal with SLP-REF was to design an architecture that could aggressively uncover SLP, but that remained similar enough to familiar SIMD extensions that our results would have relevance. While SLP-REF is aggressive, there are probably many more complex variations that would yield higher degrees of SLP. For example limiting ourself to finite staging and pending queues leads to potential missed opportunities for SIMD instruction combining. We could have given SLP-REF the ability to combine non-contiguous loads and stores, but we chose not to since a realis-

tic implementation of this capability is infeasible given the organization of typical microprocessors.

3.2. Benchmark Programs

Since we are trying to determine how much and what kind of SLP exists in typical integer applications, we chose a variety of integer intensive benchmarks to use in our simulation study. The benchmark programs were selected to reflect a wide range of multimedia and traditional integer applications. Table 1 lists the benchmarks that we studied along with the number of cycles that we simulated to produce our results. The programs from SPECint95 were run on subsets of their reference inputs to keep the simulation times reasonable, and for the same reason, programs were only allowed to run for 100,000,000 instructions or until completion. For the MediaBench programs that we examined, both encode and decode variants were simulated.

All benchmarks were compiled with the DEC C compiler version 5.8 with *-O5, -ifo,* and *-arch ev56* enabled. At the O5 optimization level, DEC C does loop unrolling and attempts to vectorize loops operating over char and short arrays. We forced the compiler to use the Alpha BWX extensions so that byte and halfword load and store operations would be more obvious in the instruction stream.

Name	Description	Instructions Simulated
compress	compression utility	100M
gcc	GNU C Compiler	100M
go	strategy game	100M
li	Lisp interpreter	100M
m88ksim	CPU simulator	100M
perl	Perl interpreter	100M
vortex	DBMS	100M
epic	Pyramid image codec	10M (decode) / 65M (encode)
gsm	GSM audio codec	71M (decode) / 100M (encode)
jpeg	JPEG image codec	4M (decode) / 16M (encode)
mpeg2	MPEG2 video codec	100M (decode/encode)

Table 1: Benchmarks simulated. The top group of benchmarks are from SPECint95. The bottom group are from MediaBench. Simulations were halted after 100 million instructions.

4. Results

4.1. Available SLP

Available SLP is indicated by dynamic instruction count reductions due to SIMD instruction combining. Our simulation framework considers all of a program's instructions, including time spent in external library routines. The dynamic instruction count reductions that we report should, in a crude way, reflect the speedups that one could expect from SIMD combining.

To determine SLP we varied three simulation parameters—pending queue size, packing style and SIMD word length. Packing style has two settings—homogenous and ideal. Homogenous packing requires that all sub-word registers in a SIMD operand have the same size. This reflects the reality of multimedia extensions found in existing processors. Ideal packing removes this constraint and allows sub-word registers to combine arbitrarily up to the size limits imposed by the SIMD word size setting. Figure 4 illustrates the difference between operations created with homogenous versus ideal packing.

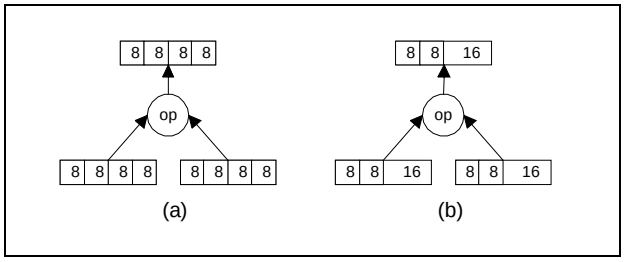


Figure 4: Homogenous vs. Ideal packing. (a) a 4 x 8-bit SIMD operation created with homogenous packing, (b) a 2 x 8-bit + 1 x 16-bit SIMD operation created with ideal packing.

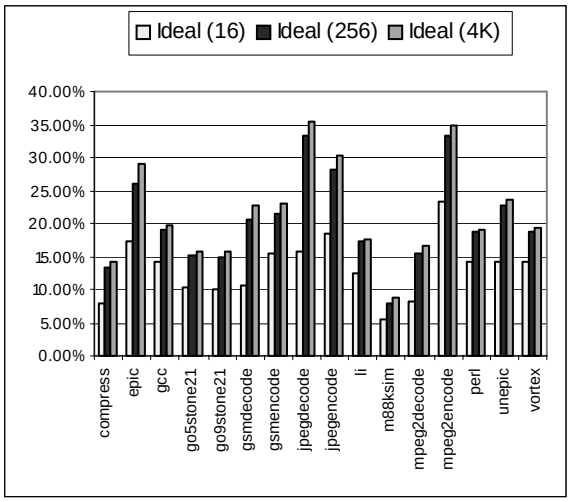


Figure 5: Dynamic instruction count reduction for ideal packing, 128-bit SIMD words and pending queue sizes of 16, 256 and 4096.

To determine the effect of pending queue size on available SLP, we ran a series of experiments using ideal packing and 128-bit SIMD word lengths, while varying the pending queue size from 16 entries to 4K entries. Figure 5 shows the results of these experiments. The largest dynamic instruction count reduction (35.6%) occurred in jpegdecode with 4K pending queues, and the smallest

dynamic instruction count reduction (5.6%) in m88ksim with 16 entry pending queues.

The effect on dynamic instruction count reduction from reducing the pending queue size was not as dramatic as we anticipated. Since reduction of the size of the pending queue reduces the scope of possible instructions with which new instructions might combine, we expected larger decreases in dynamic instruction count reduction. Instead we went from an average reduction across all benchmarks of 21.6% with 4K pending queues, to an average across all benchmarks of 13.3% with 16 entry pending queues—a difference of only 8.3 percentage points.

Our next set of experiments explored the trade-offs between ideal and homogenous packing. While ideal packing increases the opportunities for SIMD instruction combining and should result in larger dynamic instruction count reductions than homogenous packing, current multimedia extensions don't support the single SIMD instructions with mixed sub-word lengths that ideal packing would require. To determine the extent to which dynamic instruction count reduction is reduced by forcing homogenous packing, we ran a series of experiments with pending queue sizes fixed at 4K, SIMD word length at 128-bits, and either homogenous or ideal packing. Figure 6 shows the results of these experiments.

With homogenous packing enforced, the average dynamic instruction count reduction across all benchmarks is 18.6% versus 21.6% for ideal packing, a difference of only 3 percentage points. Again these findings are somewhat surprising. Homogenous packing imposes non-trivial additional constraints on how instructions can combine beyond the constraints of ideal packing. We anticipated substantially larger decreases in dynamic instruction count reduction due to these additional constraints. The fact that homogenous packing doesn't significantly affect dynamic instruction count reductions implies that current multimedia extensions, which only have homogenous operations, are probably adequate.

Current multimedia extensions have SIMD word lengths of 64 or 128-bits. Longer SIMD word lengths allow more operations to be packed into a single instruction which potentially leads to increased dynamic instruction count reduction. We ran a series of experiments where SIMD word length was increased beyond SLP-REF's default SIMD word size of 128-bits. Figure 7 shows dynamic instruction count reduction with ideal packing, 4K pending queues and SIMD word sizes of 128, 256, 512 and 1024-bits. Dynamic instruction count reduction averages 24.3% with 128-bit SIMD words and 25.4% with 1024-bit SIMD words. Increasing SIMD word lengths by a factor of eight only yielded an average improvement in dynamic instruction count reduction of one percentage point. This finding is somewhat surprising and indicates that the 128-bit SIMD word sizes common in the latest generation of multimedia extensions (Intel SSE and Motorola AltiVec) are adequate for the types of integer applications that we studied.

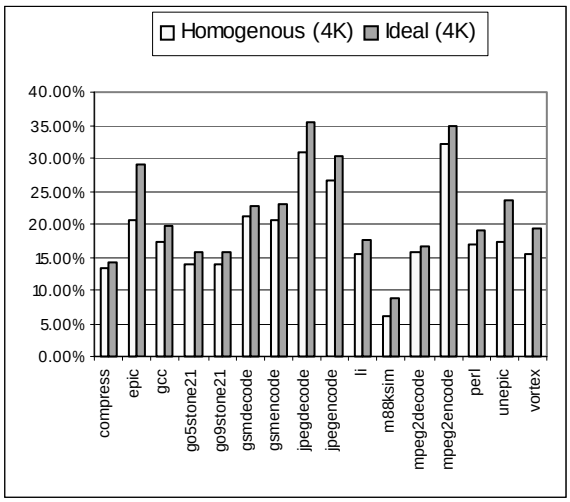


Figure 6: Dynamic instruction count reduction for homogenous and ideal packing with 4K pending queues and 128-bit SIMD words.

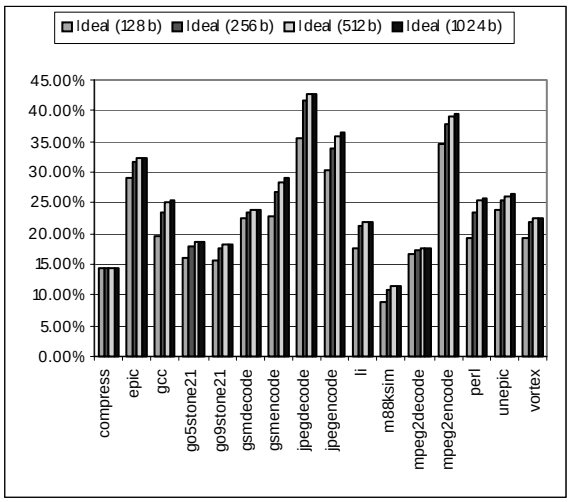


Figure 7: Dynamic instruction count reduction for ideal packing, 4K pending queues, and SIMD word sizes of 128, 256, 512 and 1024 bits.

Finally, it is important to note that we tried more pending queue sizes than the 4K, 256 and 16 entry data points in Figure 5 indicate. Lowering the pending queue size to 2K and even 1K produced no appreciable decrease in dynamic instruction count reduction. This suggested to us that pend-

ing queues with 4K entries were probably large enough for our experiments and that further increases in pending queue size would yield diminishing increases in dynamic instruction count reduction.

4.2. Instruction Mix

The available SLP experiments in the previous section use dynamic instruction count reduction, a metric that weights all instructions equally, to measure SLP. In this section we examine the actual mix of instructions that contribute to dynamic instruction count reduction. Knowing the mix of instructions can be helpful when trying to determine which kinds of operations to include in a multimedia extension. Instruction mix also lets us see what percentage of high-latency instructions are combined into SIMD operations.

We gathered statistics on the mix of SIMD instructions created in the ideal, 4K pending queue, 128-bit SIMD word experiments. We classify SIMD instructions as to whether they are loads, stores, comparison operations, floating-point operations, logical operations or integer arithmetic operations. Figure 8 shows the percentage of *total* dynamic instruction count reduction due to combining instructions into SIMD operations in a given category. For instance, in compress 64% of the total dynamic instruction count reduction results from combining instructions into SIMD arithmetic operations.

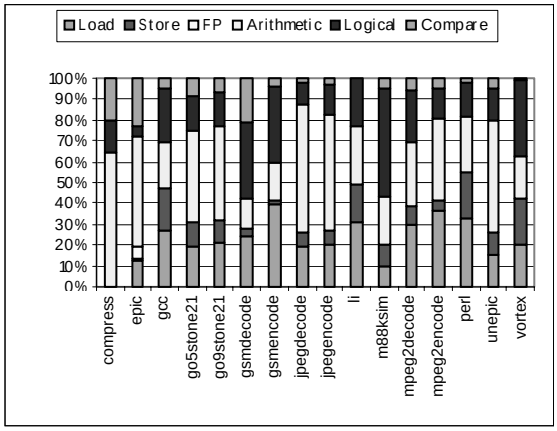


Figure 8: Instruction mix. This graph illustrates the percentage of *total* dynamic instruction count reduction due to combining instructions into SIMD operations of a given category. Simulations conducted using ideal packing, 4K pending queues and 128-bit SIMD words.

The category percentages in Figure 8 are relative to total dynamic instruction count reduction. Since loads are higher latency than other SIMD candidate instructions, and account for an average of 22% of *total* dynamic instruction count reduction, we decided to take a closer look at the actual dynamic instruction count reductions due to SIMD load combining alone. Figure 9 shows the results of these measurements. We observed that the actual dynamic instruction count reduction that is attributable directly to SIMD load combining ranges from 0% in compress to 12.6% in mpeg2encode. For most applications SIMD load combining accounts for less than 6% of the dynamic instruction count reduction and averages 5%. Although this is a relatively small dynamic instruction count reduction in most applications, the fact that loads are high latency suggests that SIMD loads could have a significant impact on actual execution times.

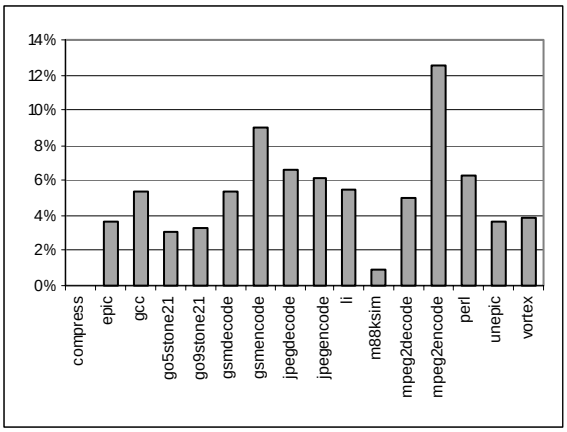


Figure 9: Load effect. Percent dynamic instruction count reduction from SIMD load combining.

4.3. SIMD Instruction Types

Homogenous SIMD instructions have a characteristic type given by the number of sub-word registers packed and the size of the individual sub-words. Knowing the distribution of characteristic types in typical applications is very useful. This information can help multimedia instruction set designers determine which types of SIMD registers and instructions to include in their extensions, and can help us understand how available SLP is different from available ILP in the same applications.

We looked at the distribution of characteristic types in the homogenous, 4K pending queue, 128-bit SIMD word simulations of all benchmarks programs. Figure 10 shows the percentage of total instructions that were combined into SIMD instructions of a given type. For example, in compress, of all instructions that were simulated, 14% of them became part of 2 x 8-bit SIMD instructions.

In Figure 11, we combine the data in Figures 10a-d into a single plot which shows the relative weights of SIMD

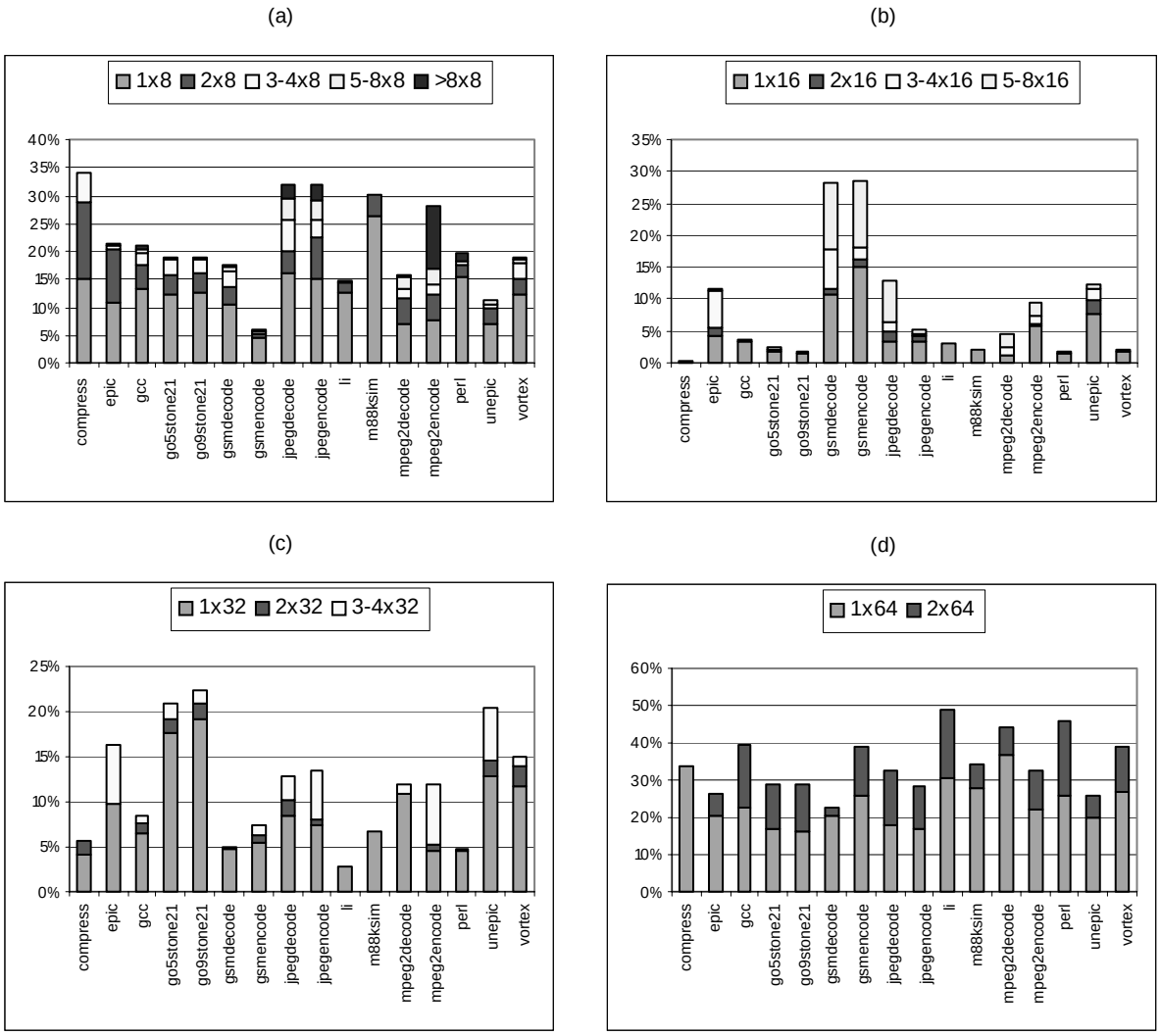


Figure 10: Percent of total instructions combined into SIMD operations of type (a) k x 8-bits, (b) k x 16-bits, (c) k x 32-bits, and (d) k x 64-bits, using 4K pending queue, ideal packing, and 128-bit SIMD words.

instructions according to the number of sub-word operands (of any size) packed into SIMD words.

Surprisingly, there are very few instances where combining three or more instructions into single SIMD instructions has a significant impact on dynamic instruction count reduction. An exception might be mpeg2encode, in which 11.2% of all instructions were combined into 16 x 8-bit SIMD operations. On average though, 16% of instructions are combined into 2-by SIMD operations, 10% are combined into greater than 2-by SIMD operations, and only 4% are combined into greater than 4-by SIMD operations.

With only 10% of instructions combined into greater than 2-by SIMD operations, it is unlikely that SIMD execution will be able to deliver performance improvements over traditional ILP execution on the types of integer benchmarks that we've studied.

5. Related Work

Despite the fact that multimedia extensions have been available in production microprocessors for several years, very little has been published on automatic exploitation of the SIMD features of these processors. All commercial processors with multimedia extensions require the compiler or programmer to uncover available SLP and to generate the appropriate SIMD code to exploit it. There are two primary SLP compilation approaches suggested by the literature.

By treating packed SIMD registers as short vectors, vector compilation techniques can be used to extract the SLP in certain types of programs. Vector compilation subsumes a variety of program transformations that can be used to rearrange and simplify loops so that vector instructions can be used to execute loop body statements [22]. On

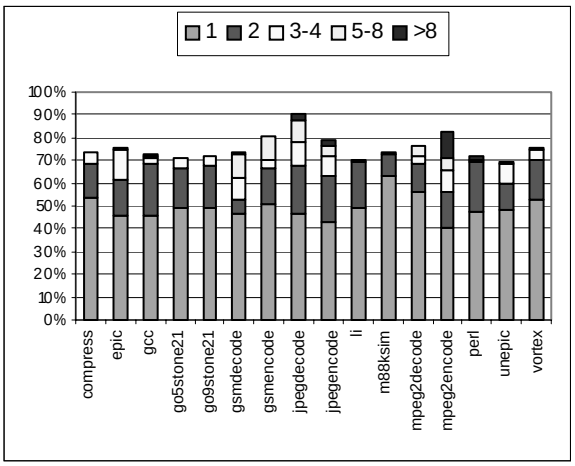


Figure 11: Percent of total instructions combined into SIMD operations with either 1, 2, 3-4, 5-8, or greater than 8 sub-words of any size packed into SIMD operands.

many vector architectures loops must be arranged so that elements of a vector are arranged contiguously in memory and accessed in order by the loop's statements. A loop so transformed is said to be vectorized. Vectorization techniques are most effective on loops where the control variable's upper limit is statically determinable, or at least loop invariant. Loop bodies with complex inter-statement control and data dependences or vectors which are heavily aliased may not be vectorizable [17].

On fine-grained SIMD architectures vector lengths tend to be small, from 2 to 16 elements, and vector instructions themselves tend to have very low latencies. On vector supercomputers, vector lengths are large (often greater than 1024 elements) and setup for execution of vector instructions is expensive. On these machines it is important to arrange loops so that vector lengths are maximized. While this cannot hurt performance on a fine-grained SIMD architecture, arranging loops for long vector lengths is not necessary.

On programs that spend a significant amount of their execution time in vectorizable loops, vector compilation can yield excellent performance on fine-grained SIMD architectures. But the fact that a program contains few or no vectorizable loops does not mean that good performance cannot be had.

Larsen and Amarasinghe have proposed a non-vectorizing compilation approach for fine-grained SIMD architectures that attempts to synthesize SIMD instructions from the statements of single basic blocks [9]. Basic blocks do not typically exhibit enough SLP to generate good SIMD code, so their approach must be combined with loop transformations that boost SLP within basic blocks. Loop unrolling can effectively boost SLP by increasing the number of similar statements in a basic block, and is easy to implement. Larsen's and Amarasinghe's approach has proved effective on floating-point intensive kernels and some of the applications from SPECfp.

6. Conclusions

In this paper we have studied available SLP in a set of representative integer intensive programs. Our studies have demonstrated that features such as non-homogenous operations (ideal packing), SIMD word sizes larger than 128-bits, and large dynamic traces (pending queues) do not substantially improve available SLP.

More importantly though, our studies have revealed that the type and degree of SLP in many of the integer benchmarks studied is substantially similar to ILP when considering performance only. Even multimedia benchmarks like mpeg2encode, which intuitively should benefit substantially from SIMD execution, has only 11% of its instructions combined into 16 x 8-bit SIMD operations. On average less than 10% of all instructions simulated were combined into SIMD instructions with more than two sub-word operands per SIMD word. Under these conditions SIMD execution appears no better than normal execution on a wide issue machine with two or more integer execution units.

The single exception to this conclusion might be SIMD loads. We found significant opportunities for SIMD load combining in the integer benchmarks. Combining high latency loads may yield improvements in application performance not attainable through normal ILP execution.

This finding should help to focus the efforts of compiler researchers on finding and exploiting available SLP in floating-point and multimedia codes. It may also suggest that compiler researchers need to attack the difficult problem of detecting the use of saturating arithmetic and other multimedia idioms so that application-specific SIMD instructions can be used to improve the performance of applications.

While available SLP might not be distinguishable from ILP in terms of performance of integer intensive programs, this does not mean that SIMD instructions and SIMD architectures do not merit our attention. Fine-grained SIMD architectures may offer smaller program sizes with improved instruction stream performance, and provide opportunities for power savings that would not be possible with a traditional wide-issue processor organization.

Finally, our study of available SLP is far from the final word on the subject. Available SLP that our infrastructure can detect can be impacted by compile-time analyses and transformations. One such compile-time approach that we

are eager to explore is the aggressive static bit width analysis techniques proposed by Stephenson, et al. [16]. It is not immediately obvious that this will improve SLP in the benchmark programs that we studied, but we are hopeful.

7. Acknowledgments

This work was supported in part by NSF Grant ASC-9612756 and by a generous grant from Panasonic AVC American Laboratories. We also thank Kevin Skadron for allowing us to use the LAVA cluster for our simulations and the anonymous reviewers whose comments have improved this paper. We are especially grateful to Sally McKee for giving us detailed feedback on this paper.

8. References

[1] *Alpha Architecture Handbook*. Digital Equipment Corporation, 1992.

[2] Krste Asanovic and David Johnson. Torrent architecture manual. Technical Report CSD-97-930, University of California, Berkeley, January 24, 1997.

[3] George H. Barnes, Richard M. Brown, Maso Kato, David J. Kuck, Daniel L. Slotnick, and Richard A. Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, C-17(8):746–757, August 1968.

[4] Mark Bohr. Silicon trends and limits for advanced microprocessors. *Communications of the ACM*, 41(3):80–87, March 1998.

[5] David Brooks and Margaret Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. *HPCA-5*, 1999.

[6] Douglas C. Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[7] Sam Fuller. Motorola's altivec technology.

[8] Norman P. Jouppi and David W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 272–282, 1989.

[9] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. Technical Report LCS-TM-601, MIT Laboratory for Computer Science, Cambridge, MA, November 1999.

[10] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[11] C. G. Lee and M. G. Stoodley. Simple vector microprocessors for multimedia applications. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 25–36, Los Alamitos, November 30–December 2 1998. IEEE Computer Society.

[12] Ruby Lee. Subword parallelism with max-2. *IEEE Micro*, 16(4):51–59, August 1996.

[13] Stuart Oberman, Greg Favor, and Fred Weber. 3dnow! technology: Architecture and implementations. *IEEE Micro*, 19(2):37–48, March 1999.

[14] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):24–38, January 1997.

[15] Paul Rubinfeld, Bob Rose, and Michael McCallig. Motion video instruction extensions for alpha. http://www.europe.digital.com/semiconductor/alpha/papers/pmvi.ps, 1996.

[16] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[17] P. Tang and N. Gao. Vectorization beyond data dependences. In ACM, editor, *Conference proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain, July 3–7, 1995*, Conference Proceedings of the International Conference on Supercomputing 1995; 9th, pages 434–443, New York, NY 10036, USA, 1995. ACM Press.

[18] Shreekant Thakkar and Tom Huff. The internet streaming simd extensions. *IEEE Computer*, 32(12):26–34, December 1999.

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

[20] David Wall. Limits to instruction level parallelism. *4th Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 8-11, 1991.

[21] David W. Wall. Limits of instruction-level parallelism. Technical Report 93/6, DEC, Palo Alto, CA, November 1993.

[22] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.