Superword Processors

22 June 2001

Abstract

This paper introduces the concept of a *superword* processor, a style of computer-architecture design in which a traditional processor datapath is replicated in SIMD fashion, transforming each machine instruction into a SIMD equivalent. Superword techniques are appealing because they require minimal changes to an existing design, offer backward compatibility with an existing ISA, and offer large potential for performance improvements.

This paper also describes new compiler techniques that are key to the successful use of a superword processor. Automatic parallelization of general-purpose, sequential applications is needed in order to provide a seamless interface to the user. One of the most important tasks of the compiler is the grouping of multiple memory operations into a single wide load or store. We will show that this can greatly increase memory bandwidth.

We will show that a superword-extended VLIW machine can outperform a clustered VLIW with similar resources by a factor of 2.43. Even compared with an ideal VLIW machine we achieve speedups of 1.84.

1 Introduction

The current level of performance seen in today's microprocessors is due to the aggregate contributions of several different innovations. Among these are out-of-order execution, multiple instruction issue, and branch prediction. Most of these structures are providing diminishing returns as they are scaled in size and complexity. An orthogonal technique that remains largely unexplored is *superword extension*. In a superword-extended processor, the datapath is widened in SIMD fashion to provide a more highly parallel design. In essence, each machine instruction is transformed into a SIMD equivalent, including memory operations. This technique works in conjunction with existing architectural components, and can be added to a superscalar or VLIW architecture.

A superword processor has several appealing aspects. First, it is a relatively simple and straightforward way to take advantage of an increasing number of transistors. Second, a superword processor can be made backward compatible with an existing design by providing a mode in which only one *slot* in the superword is active. Next, widening the datapath does not necessitate changing the complexity of existing structures. For example, the register file requires no additional read or write ports. Furthermore, the instruction fetch and decode units remain unchanged since the instruction set is virtually unchanged. Because superword instructions provide a compact method for specifying multiple operations, we attain a higher effective bandwidth to instruction memory with the same hardware.

One of the most important aspects of a superword processor is its ability to seamlessly integrate a wide path to data memory. In the same way that the datapath and register file are extended, so is the bus to data memory. By banking the memory such that each *lane* — a portion of the superword datapath — has fast access to a single bank, memory latency can remain the same. However, if we take advantage of the wide memory bus, we can provide much higher bandwidth compared to the base machine. Together with a more compact instruction encoding, superword processors offer the opportunity to significantly ease the memory bottleneck.

In order to take full advantage of the superword extension, we require a complementary set of compiler technologies. Acceptance of the architecture relies on automatic parallelization of sequential code. As suggested, a key component is the compiler's ability to efficiently use a wider path to memory.

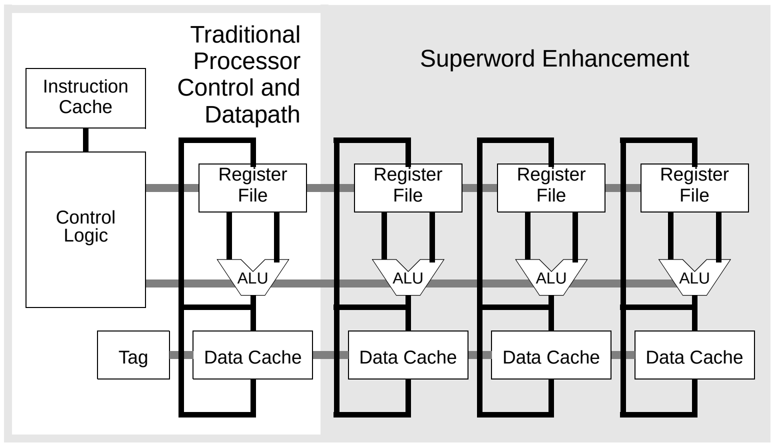

Figure 1: Block diagram of a 4-lane superword-extended processor.

This paper makes the following contributions:

- A description of the superword processor paradigm, and how it can be used to provide another axis for improving microprocessor performance.
- An overview of the compiler algorithms we have implemented to successfully exploit the parallelism available on a superword machine.
- A qualitative and quantitative analysis of the performance of a superword processor versus a conventional VLIW design with the same resources.

2 Superword Processors

In existing designs, the machine's datapath width is typically the same as the basic word width. A word is sized according to addressing and accuracy considerations, and is usually 32 or 64 bits. In a superword architecture, the datapath is extended by duplicating the functional units, register file, and data cache. In this scheme, the machine's datapath holds several pieces of word-sized data.

In theory, a superword processor's datapath could be made arbitrarily wide as transistor counts increase. However, there is a physical problem of maintaining clock speed if control is distributed across the die area. We believe that a superword is best used in small configurations, typically of size 4 or 8. The amount of parallelism offered by such configurations allows us to use simple compiler techniques to parallelize general-purpose applications.

Superword extension affects many architectural components. It increases the effectiveness and efficiency of some elements, enables the expansion of others, and introduces the opportunity for new architectural developments. Taking advantage of many of these components requires the cooperation of the compiler. In Section 3 we describe compiler techniques we have developed to take advantage of the most important superword enhancements.

2.1 Memory Operations

Perhaps the most important component of a superword processor is its ability to ease the memory bottleneck experienced by existing microprocessors. This is partly achieved by extending the path to data memory along with the datapath and register file. A wide path to memory allows us to load and store multiple data words by issuing a single superword memory operation.

In this scheme, the data cache is built using banks such that each lane of the machine has a direct path to one bank. This provides very low memory latency as long as data are used in the proper lane. Communication to another lane will require a slower communication channel. This can be achieved with hardware, or can be done in software using the packing and unpacking operations described below.

To further simplify the memory system, we believe memory operations should be aligned to a superword boundary. This ensures that each memory operation will be contained within the same cache line. As a result, it is not necessary to handle the case when a load or store spans a cache line or page boundary. At first this may seem overly restrictive. However, in Section 3, we will describe the compiler techniques that make data placement and alignment possible.

2.2 ALU Operations

In a superword extension, ALU units are extended by replication such that each word within a superword is operated upon in parallel. This has the effect of transforming each instruction into a superword equivalent. Extending the processor in this way is natural since the same control logic can be used for each new unit, or can be duplicated if cycle time degrades beyond acceptable allowances.

Most current general-purpose microprocessors have recently added multimedia extensions, providing SIMD instructions that operate on subwords. These operations are analogous to superword ALU operations since multiple subword sized data are contained within a single machine word. Support for subword operations can be added to a superword machine seamlessly by replicating these units as well. In essence, this provides support for subword operations with a much higher degree of parallelism.

In Section 3.1, we introduce a novel compiler technique that constructs SIMD operations from sequential code. Our algorithm uses the same mechanism for extracting both superword and subword level parallelism.

2.3 Pack and Unpack Operations

The compiler may choose to shuffle data in order to maximize the benefits of superword execution. Therefore, a mechanism is required for moving a data word to a different lane within the superword datapath. This packing and unpacking can be viewed as scatter and gather operations within the register file.

Fast packing and unpacking operations will help processor performance in situations where data are frequently moved across lanes. The scope and range of these operations will vary among architectural implementations. At a minimum, however, a set of simple shuffle and permute instructions should be added to move data with reasonable efficiency. These types of operations are supported on the subword level in designs such as AltiVec [4] and MicroUnity [6].

The most aggressive machines could use a single instruction to pack together data words from multiple lanes and multiple registers. Such an implementation will require a large number of register file ports and tight integration between lanes, leading to a potential degradation of cycle time. Our compiler algorithm attempts to limit the amount of communication between lanes. We believe that a simpler design can provide the necessary functionality.

2.4 Predication

We incorporate predication into a superword architecture by expanding the traditional one-bit predicate registers to a multi-bit mask. Typically, each mask bit would correspond to one word within the superword. However, if subword operations are also supported, it may be desirable to provide mask bits to each subword as well.

Superword predicates introduce a host of opportunities. Predication can be used to reduce the frequency of packing and unpacking operations by selectively updating only selected words within the superword. Predication can also be used as a power-saving mechanism since portions of the datapath can be switched off when the mask indicates that they are inactive.

The common use of predication is to follow multiple branches of control simultaneously instead of executing control transfer instructions. This is beneficial in architectures that provide multiple issue slots when some of the slots may be unused anyway. A compiler must be careful when predicating a piece of code since the unnecessary computation that is introduced can quickly outweigh the benefits of removing control flow. However, a superword processor using masks is much less sensitive to this phenomenon. This is due to the compact SIMD instruction encoding. If an instruction is not utilizing all lanes within the superword, there is no extra instruction overhead to fill these extra slots with potentially useless computation.

Superword masks introduce new opportunities for predicate manipulation instructions beyond what is provided for traditional one-bit registers. For example, consider mapping multiple iterations of a *while* loop into different superword lanes. In this case, the loop condition must be checked independently for each iteration. The computation in one iteration must not execute if any of the loop conditions from previous iterations have failed. The exit conditions for multiple iterations can be computed simultaneously in a SIMD manner and a mask register set accordingly. However, in order to provide the correct functionality, a predicate bit in the mask register is dependent on all the bits that correspond to previous iterations. A masking operation that clears all predicate bits to the right of the first zero would be very useful in this situation.

We believe other interesting uses of predication exist in the context of superword processors. However, these are beyond the scope of this paper and we leave them to future work.

2.5 Data Address Generation

The proposed superword architecture shown in Figure 1 does not duplicate the cache tag and does not require a complex and impractical multi-ported cache. One result of this design is that only one address is generated for a wide memory operation, meaning that all but one lane will be inactive during address calculation. This should not be viewed as an inefficiency since a wide memory operation substitutes multiple address calculations with a single address calculation and multiple memory operations with a single memory operation.

However, there are some limited opportunities for multiple address generation using more than one lane. We also relegate this to future work.

2.6 Control Transfer Operations

Since only a single branch instruction is executed in any given cycle, most of the lanes will typically go unused during computation of a branch condition. However, these lanes could be used to compute several branch conditions in parallel. A predicate mask could then be used to select one of the conditions before branching. This could be used to improve the performance of logical *and* and *or* operations, multi-way branches, and virtual method dispatch. Such mechanisms require compiler support and are not discussed in this paper.

2.7 Processor Control Logic

Our proposed superword extensions have minimal impact on the control logic of an existing base architecture. In theory the same control could be distributed to all replicated units. In practice, however, physical design constraints will require that some additional action must be taken. At the very least, driver strengths will have to be increased to accommodate the increased fan-out and wire lengths. For some designs it may be necessary to duplicate control logic in order to maintain a high clock speed. The main expense in this scheme is the extra transistors required for these redundant structures. However, this should not present a large problem since control logic typically only uses a small portion of the total die area.

2.8 Instruction Fetch

In addition to improving data cache performance , a superword processor also improves performance in the instruction cache and instruction fetch unit. This again is due to the compact SIMD instruction encoding. Since a superword instruction specifies multiple operations with a single base operation, instruction bandwidth is increased without any changes to the instruction memory or instruction fetch unit.

Typical parallelizing compilers usually trade increased executable size for better performance. The classic example of this is the use of loop unrolling which is used to provide the compiler with larger basic blocks. As we will explain in Section 3, we also use loop unrolling to expose parallelism to the compiler. However, after the combination of multiple sequential operations into a single superword operation, code size is reduced. In the best case, an unrolled loop can be compacted to its original size.

2.9 Instruction Set Architecture

A superword-extended processor can easily be made backward-compatible with the existing base ISA. Since each instruction opcode is the same, it is simple to provide a mode in which only one superword lane is active. As discussed earlier, the remainder of the datapath can be powered down in order to conserve power.

To fully support the new superword model, a few new opcodes will need to be added. These include the packing and unpacking operations as well as the predicate mask manipulation instructions. These changes are minimal and will have a small impact on the existing ISA.

3 Superword Compilation

This section describes the compiler algorithms we have implemented in order to successfully exploit the parallelism available in a superword processor. The key technique is drawn from our previous work in compiling for multimedia extensions [9], in which we describe a novel method for extracting superword level parallelism (SLP). This algorithm is highlighted in the following subsections. We also describe two compiler analyses we have developed in order to make SLP extraction possible.

3.1 SLP Extraction

We have developed a simple and robust compiler algorithm that parallelizes a sequential program for execution on SIMD hardware. The algorithm works at the level of basic blocks in the compiled code and is therefore scalable and applicable to a wide range of programming styles, source languages, and application domains. Our analysis does not rely on any high level program information.

One of the key innovations of superword processors is the ability to cleanly exploit wide memory operations to ease the memory bottleneck. In order to do this, our compiler must combine several short memory operations into a single wide load or store. Superword extraction therefore begins with an n^2 search of all memory operations in the basic block. When we find homogeneous operations that reference adjacent loca-

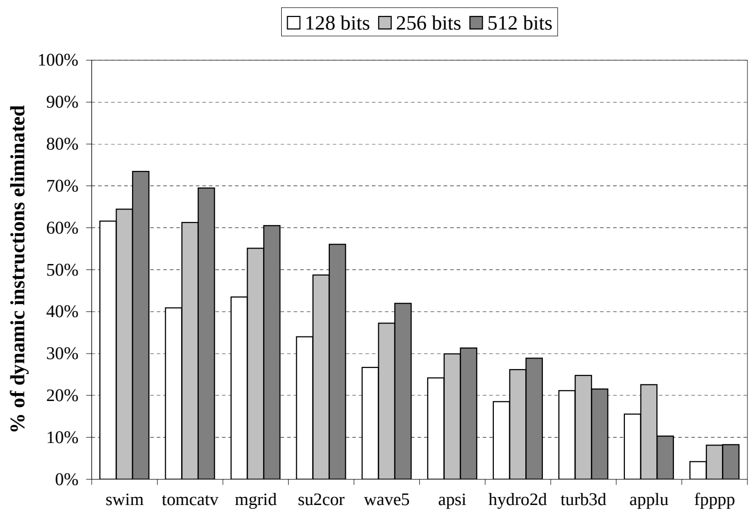

Figure 2: Percentage of dynamic instructions eliminated on a variety of different superword sizes.

tions in the same cache line, we merge them into a wide memory operation. Detection of adjacent memory references is handled with a combination of alignment analysis and distance analysis, which are described later.

Once an initial set of packed memory operations has been built, the next step is to build a set of logical and arithmetic operations that can use or produce the current set of packed data. To do this, we follow def-use and use-def chains from packed operations. If these lead to new operations that are independent and isomorphic, they too can be combined into SIMD operations and added to the set of packed operations. This process repeats until no new SIMD operations are discovered.

Figure 2 details the success of SLP extraction on the SpecFP95 benchmark suite. Our results are presented from the perspective that superword parallelization replaces several homogeneous operations with a single wide operation. This figure shows the percentage of dynamic instructions eliminated from parallelization for three different superword widths.

Conventional wisdom has held that a vectorizing compiler should be used to extract SIMD parallelism. This may be true for vector supercomputers which exploit huge amounts of parallelism in the form of long vectors. However, we believe that the smaller degree of parallelism offered by a superword machine

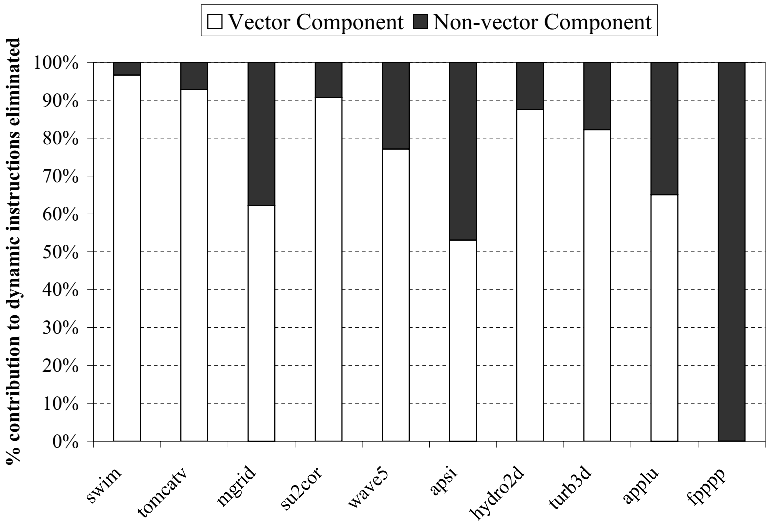

Figure 3: The amount of SIMD parallelization obtained from vectorizable loops versus non-vectorizable loops.

warrants a fresh approach. In fact, we discovered that a notable amount of parallelism extracted by our system was in fact non-vectorizable. Figure 3 breaks the results shown in Figure 2 into vectorizable and non-vectorizable components. These numbers were obtained by observing that a vector compiler targets loop nests alone. Vector parallelism was therefore measured by counting the SIMD operations that were built only from identical operations in different loop iterations.

Scientific benchmarks are typically considered to be highly vectorizable. The ability of our algorithm to extract parallelism in places other than vectorizable loops strengthens the idea that SLP exists in more application domains.

3.2 Alignment Analysis

The most straightforward way to lay out data on a banked-memory architecture is to use low order interleaving. In this scheme, the bank assignment for a given address can be determined by the address's low order bits, provided that the number of banks is a power of two. We consider a static memory reference to be *aligned* if it accesses the same bank for every dynamic instance. Alignment analysis attempts to prove

```
for (i=0; i<N; i++) {
  A[i] = 0;
}
```

(a) A simple loop with a single memory reference.

```
for (i=0; i<N; i += 4) {
  A[i+0] = 0;
  A[i+1] = 0;
  A[i+2] = 0;
  A[i+3] = 0;
}
```

(b) The same loop after unrolling.

```
for (i=0; i<N; i++) {
  if (&A[i] % width == 0)
    break;

  A[i] = 0;
}

for (; i<N; i += 4) {
  A[i+0] = 0;
  A[i+1] = 0;
  A[i+2] = 0;
  A[i+3] = 0;
}
```

(c) A pre-loop inserted to guarantee alignment in the unrolled loop body.

Figure 4: Techniques to increase the alignment in inner loops.

this property for each static reference in the program. When successful, the analysis is also able to compute the bank a memory operation will access. This information allows us to simplify the memory system of a superword processor since we can ensure that the address sent to each bank is identical for any given wide transaction.

Notice that in practice the majority of memory operations will not be aligned. Consider a simple example of a loop that steps through the elements of an array. An example is shown in Figure 4(a). Here, the store instruction will access consecutive banks on each iteration.

In order to increase the percentage of aligned memory operations, our compiler performs a series of alignment-increasing transformations. One of the most important is loop unrolling. The code in Figure 4(b) shows the result of unrolling the original loop. For simplicity of illustration, assume that the number of loop iterations is always a multiple of the unroll factor. After unrolling the loop by a factor consistent with the number of banks, we can guarantee that each memory operation in the loop only accesses a single bank. This is the case in our example assuming a machine with four banks, each with width equal to the size of the elements of A.

Since inner loops comprise the majority of dynamically executed instructions, it is very important that memory references within loops are aligned as much as possible. Loop unrolling is effective for array references when the array is a local or global variable. For global arrays the compiler can control the alignment of the array base directly. For local arrays, the compiler can control the alignment of the base relative to the current stack frame. As long as stack frames are allocated in sufficiently large sizes, the alignment of local arrays is also known. However, when an array is passed as an argument to the enclosing function, loop unrolling alone can not guarantee aligned references since the base of the array is unknown. Even if this information could be uncovered through whole-program analysis, it might be the case that the base of a passed array is aligned differently for different calls to the function.

To overcome this limitation, our compiler inserts a pre-loop that executes the original loop until the references within the body reach a known alignment. From here, control is transferred to an unrolled version of the loop where the alignments of references are now guaranteed. An illustration of this is shown in Figure 4(c). As long as the pre-loop only executes a few iterations, the majority of dynamically executed memory operations will be aligned. This technique was first proposed by Ellis and Fisher [5, 7] for use in a VLIW with banked memory. Our analysis extends theirs by adding profile-driven feedback to help choose the best set of conditions to begin execution of the unrolled loop. This is necessary when the loop body contains many loop references with different and sometimes conflicting alignments. In these situations, it may be impossible to guarantee alignment for all the references in the loop. Using profiling, we can find the best set of non-conflicting references.

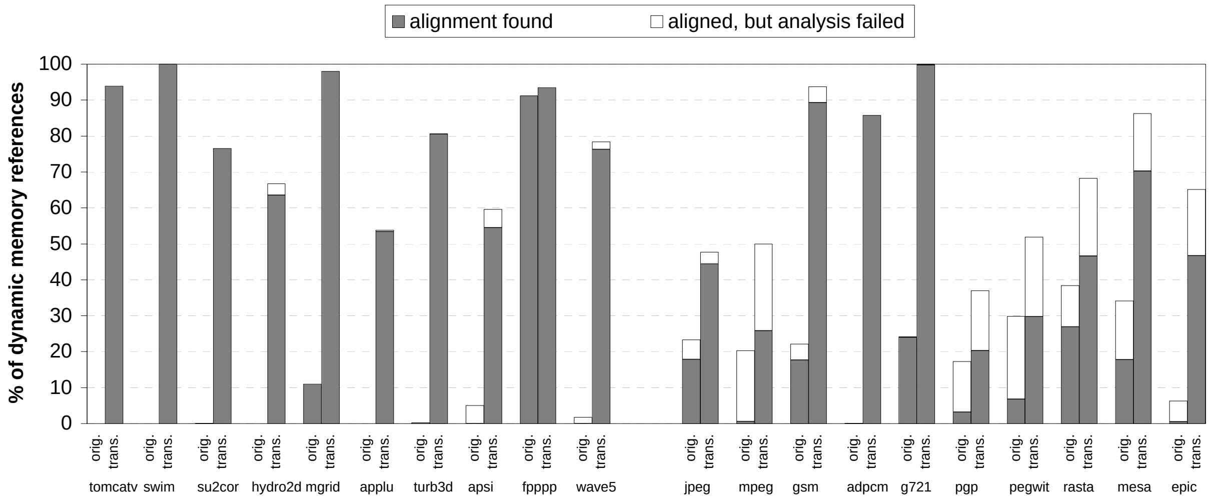

Figure 5: Results of alignment transformations and analysis on the SpecFP95 and Mediabench benchmark suites.

Figure 5 shows the results of the alignment transformations and alignment analysis for the SpecFP95 and Mediabench benchmark suites. The lower portion of each bar shows the percentage of dynamic memory references that the analysis is able to uncover. The upper portion shows the number of additional references that are actually aligned at run-time, but for which our analysis is unable to extract any information. The two bars for each benchmark show these numbers before and after applying the transformations described above. Clearly, these techniques are needed in order to provide a large number of aligned memory references. Once the transformations are applied, our analysis is able to detect the majority of aligned references.

3.3 Distance Analysis

Distance analysis attempts to determine the distance in bytes between two static memory references. This information is used to determine whether two memory operations access adjacent memory. The algorithm is implemented as a dataflow analysis that operates on low-level address calculations. For each memory reference, we attempt to build a symbolic linear equation describing the address calculation. For example, consider the following two array references:

$$A[j][i+3]$$

$$A[j][i+2]$$

Assume A is a K$\times$10 integer matrix mapped to a linear array, and the word size is 4 bytes. After dismantling the array references into address calculations, the analysis will assign the following linear equalities to each memory operation:

$$A + 40j + 4i + 12$$

$$A + 40j + 4i + 8$$

To determine the distance between these references, we subtract their linear equations. If the result of the subtraction is a constant, then we know with certainty the distance between the two operations. In this example, the result of the subtraction is 4, indicating a distance of 4 bytes between these two references. In other words, they access adjacent locations in memory.

Distance analysis is able to operate when address calculations have been converted into three-address form, or when they have been optimized by previous transformations. This is essential since SLP extraction operates at a low level. The algorithm is also able to uncover distance information across control flow. While this is not currently important for the local analysis, we expect to extend SLP extraction beyond basic blocks in the future.

4 Comparison with Other Architectures

In this section, we compare superword processors to similar designs. The architectures listed below can be used to exploit parallelism in much the same way that a superword architecture exploits parallelism. However, we feel that there are fundamental differences in design philosophy that make them less suitable for executing general-purpose applications.

4.1 Vector Architectures

Vector supercomputers have been used with huge success in the scientific computing arena. Scientific applications are usually characterized as having a massive amount of parallelism in the form of large vectorizable loops. When these loops are well-formed, a vector machine is ideal since the loops are easily mapped to the vector hardware.

More recent work has proposed building vector microprocessors in order to bring vector capabilities to the desktop. In addition to the obvious advantage of shrinking a computer system to a single die, vector microprocessors offer other benefits that make them applicable to a wider spectrum of application domains. For example, the T0 vector microprocessor [2, 8] did not incur the start-up and dead-time penalties associated with vector supercomputers. This means that the amount of parallelism needed to make a vector operation profitable is smaller since there is less overhead that needs to be amortized. As a result, there has been some work in compiling multimedia applications for execution on a vector microprocessor [10].

Despite the success of vector architectures, vector computing has remained a niche field. We believe this is for several reasons. First, vector processors are typically implemented with long vector lengths, meaning that a large amount of parallelism is required in order to efficiently use the vector hardware. This is a problem if the application simply does not contain the required level of parallelism. Even if it does, it may be difficult to extract using existing compiler technology. Vector compilers map very specific loop structures to the vector hardware. If the available parallelism does not exist within a loop, or if the loop does not match the required structure, vectorization will fail. We propose building superword processors with short SIMD parallelism. As we showed in Section 3, this allows us to look for parallelism where it is scarce, such as within a basic block.

Another reason why traditional vector computers are not well-suited for general purpose computation is the division between the vector and scalar units. In the standard design, there is a clear demarcation between the two units, which enforces a division between the computation that is scheduled on them. This organization is by design, as it is assumed that there will be little communication between vector and scalar computation. On a superword machine, this separation does not exist since all computation executes on the same integrated unit. We believe that many non-scientific applications can benefit from tighter integration. One example of this is in the use of wide memory operations. Computation that is customarily assigned to the scalar unit will undoubtedly benefit from a higher bandwidth connection to memory.

4.2 Multimedia Architectures

Multimedia extensions recently have been added to most general-purpose microprocessors [4, 11, 14, 13]. These extensions mainly add short SIMD instructions to the ISA in order to exploit subword parallelism. In the past, use of multimedia instructions was limited to applications using small data types. New extensions such as Motorola's AltiVec add full 128-bit superwords to the architecture. These designs also add SIMD floating point operations, making it possible to use the extensions to perform more general-purpose computation. In previous work [9], we described a compiler algorithm that was able to automatically parallelize sequential code for execution on these extensions.

The limitation of these architectures is that the SIMD hardware is still treated as a separate coprocessor. In AltiVec for example, there is no direct path from the vector register file to the scalar register file. All communication must be routed through memory, making it very costly. One of the main difficulties we encountered when targeting our compiler algorithms for AltiVec was the task of separating the computation between the vector and superscalar units [9].

Rather than separating a design into scalar and SIMD modules, we propose building an architecture with a single unified unit. This tight integration allows us to take advantage of wide memory operations and SIMD ALU operations for all types of computation. In addition, a single-unit design simplifies the task of extracting superword level parallelism, leading to a more effective and efficient compilation system.

	I	II	III	IV	V
	Baseline		Comparable Resources		Impractical Upper bound
Mode	Single issue	Superword	Superword VLIW	Clustered VLIW	VLIW
Clusters Lanes	1 1	1 4	1 4	4 1	1 1
Functional units	1 LOAD/ ALU/ FLOAT	4x1 LOAD/ ALU/ FLOAT	4x1 LOAD 4x1 ALU 4x1 FLOAT	1x4 LOAD 1x4 ALU 1x4 FLOAT	1 LOAD 4 ALU 4 FLOAT
Integer resisters Floating point resisters	32 32-bit 32 64-bit	32 32x4-bit 32 64x4-bit	32 32x4-bit 32 64x4-bit	32x4 32-bit 32x4 64-bit	128 32-bit 128 64-bit
Maximum number of ports per register file	2 read 1 write	2 read 1 write	4 read 2 write	4 read 2 write	10 read 5 write
Required instruction bandwidth	24 bits/cycle	24 bits/cycle	72 bits/cycle	288 bits/cycle	216 bits/cycle

Table 1: The five architecture configurations.

Instruction	LOAD	STORE	BRANCH	ALU	MUL	DIV / MOD	COMP	FADD / FMUL	FDIV	Intercluster COPY
Latency	3	1	1	1	5	20	1	4	12	1

Table 2: Instruction latencies for the tested architectures.

5 A Case Study

In the next section we analyze the performance of superword architectures — as well as our superword compilation techniques — by comparing them against several VLIW architectures. This section presents the machine models used in the comparison.

A VLIW machine provides an ideal target for comparing the trade-offs of superword extensions. A *clustered VLIW* and a superword-extended VLIW require a similar amount of hardware, and are comparably complex. We believe that these two choices would result in equivalent development and manufacturing costs. The difference in performance will highlight the suitability of the two architectures for general-purpose computing.

The main difference between the two architectures is the way they manage instruction and data bandwidth. By issuing identical operations in each slot, a VLIW machine can simulate a superword processor. However, compared to the superword model, the VLIW instruction encoding requires a larger instruction fetch bandwidth.

Superword processors also relieve data memory bandwidth problems. While wide loads and stores are very natural in a superword design, they are awkward to incorporate into a VLIW architecture. For instance, loading multiple words from memory on a VLIW would probably require using multiple destination operands. Alternatively, these destinations could be hard-coded in the architecture. These solutions are unsatisfying and, as a result, current VLIW architectures do not allow such a high bandwidth connection to data memory.

Because of such differences between processor models and compiler strategies, it is difficult to provide a fair comparison between different architectures. That said, we tried in earnest to fairly compare the machine models. Besides appropriating equal resources for machines of interest, we also use the same simulator and set of basic compiler optimizations in all cases.

The following list describes the five architectures that this paper surveys:

I. A simple five-stage pipeline with 32 general-purpose registers and 32 floating-point registers. The functional unit latencies that we assume are listed in table 2. This model is useful for baseline comparisons.

II. A superword processor with four lanes that is otherwise identical to the baseline processor. It is important to note that the datapath, and thus the memory bandwidth, is four times wider than that of the baseline architecture. Although this model is too simple to be of any practical value, it provides an uncluttered view of superword capabilities.

III. A superword-extended VLIW processor, where each of the three functional units has four lanes. It has one integer ALU unit and one floating-point unit. Because these units are superword-extended, this model is able to process four integer operations and four floating-point operations per cycle. Like machine II, this model has a memory unit that is able to access a superword from memory.

IV. A clustered VLIW processor with four clusters, each of which has three units. Each of the symmetric clusters has an integer ALU unit, a floating-point unit, and a memory unit. Since we use the same memory model as machine III, this processor will not be able to issue more than one load per cycle. In addition, each cluster can communicate one integer or floating point value with any other cluster per cycle.

V. A single clustered VLIW with nine total functional units. There are four integer ALU units, four floating-point units, and one memory unit. This machine also models the same computation resources of machines III and IV, but without the limitations imposed so that one can construct such a machine with reasonable effort. For example, this machine design requires a register file with 10 read ports and 5 write ports. However, this model provides a good upper bound.

Table 1 numerically shows the resources allocated to each machine model. As shown in the table, the first two processor models are simpler and have fewer resources than the others. Nevertheless, they are useful for baseline comparisons. For instance, comparing model II with model I demonstrates the potential performance improvements of superword-extending an architecture.

Machine III and machine IV are very similar in resource requirements and hardware complexity, providing a head-on comparison between VLIW and superword architectures. The last architecture is somewhat impractical. It is a VLIW machine that has the same resources as machines III and IV, but suffers no communication penalties.

As already noted, one advantage of superword architectures is compact instruction encoding. Unfortunately, our simulator does not take instruction compactness into account, benefitting the VLIW models that we surveyed. However, because instruction bandwidth is an important characteristic, it is worthwhile to analyze SLP's potential benefits in this area. Assuming it takes 24 bits to represent a RISC-type instruction, a fully utilized 9-functional unit VLIW has instruction widths up to 216 bits. The superword equivalent requires only 72 bits.

5.1 Experimental Methodology

This section describes the methodology used to collect simulation results. Because the goal of this project is to compare two very different architectures, our tool chain is highly flexible.

The compilation process begins in the SUIF front-end [15]. The front-end carries out many traditional as well as parallelism enhancing optimizations such as loop unrolling. Most of the compiler passes are identical for all machine models. However, for the superword-machine targets, we apply an SLP detection and extraction pass. This process was described in Section 3.

We then convert the intermediate format to the MachSUIF [1] machine representation. We have developed a parameterized machine back-end that is capable of handling all our machine models[12]. The back-end also performs register allocation. After register allocation we schedule the code using the algorithms presented in the next section. Finally, we generate a compiled simulator that is used to collect results. The simulator is parameterized for each machine, accurately modeling the occupancy and latency of each execution unit. We assume fully pipelined functional units, and a cache memory with a perfect hit rate.

5.2 VLIW scheduler

We have implemented a VLIW scheduler that schedules instructions for all machine models. For the superword machines, scheduling is performed after superword extraction. Using this scheme, superword instructions appear as normal operations, meaning the scheduler requires no knowledge of the superword extensions.

Our algorithm leverages the work done by [3]. For every instruction in a basic block, the algorithm determines the *critical path length (CPL)* from the instruction to the end of the basic block. A one-cluster schedule is computed by filling functional units with the instructions that have the highest CPL. The one-cluster schedule is then used to determine *sub-traces*. Sub-traces are critical sequences of instructions linked by data dependences that we try to assign to the same cluster. They are built using the *Partial_components* algorithm described in [3].

Next, sub-traces are mapped to clusters using a simple heuristic that tries to balance workload. In our current implementation, we move traces among clusters without taking into account the cost of eventual communication. This is done to enforce utilization of all clusters. After determining the assignment of instructions to clusters, we add inter-cluster communication where necessary.

An invariant of our scheduling algorithm is that all live registers are available in the first cluster at the beginning and end of every basic block. Thus, at the beginning of each basic block communication is inserted for all data needed in the other clusters. Similarly, at the end of each basic block, communication is inserted to copy live values back to the first cluster. Finally, our algorithm recomputes the CPL and the final schedule for all instructions.

6 Results

In this section we present some initial performance results for the machine models described in the previous section. At the time of this writing, we are only able to obtain preliminary results for nine benchmarks. Six of these are multimedia kernels, and the remaining three are from the SPECfp95 benchmark suite. These benchmarks are listed in the table below:

fir	Finite Impulse Response Filter
iir	Infinite Impulse Response Filter
dct	Discrete Cosine Transform
matrix	Matrix-Matrix Multiply
vector	Vector-Matrix Multiply
yuv	RGB-YUV conversion
tomcatv	Fluid Dynamics / Geometric Translation
swim	Weather Prediction
su2cor	Quantum Physics

Our compiler infrastructure is actually able to handle a much broader range of applications. However, integration of the backend, linker, loader, simulator and debugger is still in its infancy. We expect to obtain end to end results for a broad range of benchmarks in the near future. Furthermore, we expect the results to improve for both the VLIW and superword machine models as our infrastructure matures. As such, we believe the current comparison is valid.

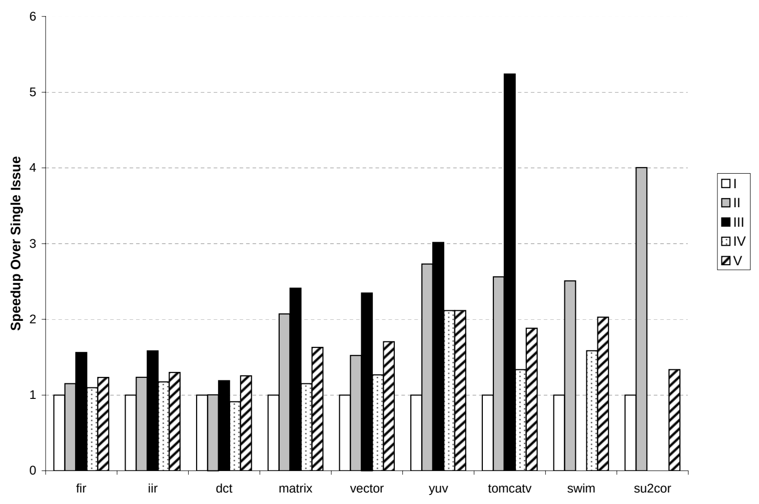

Figure 6: Speedup of each machine model relative to the base architecture.

Figure 6 shows the overall speedup of our benchmarks for each of the five processor models. For every benchmark, the superword-extended VLIW outperforms the similarly equipped clustered VLIW. In fact, this machine performs exceptionally well for tomcatv. This is due to a high level of parallelization in both the superword and VLIW axes. The superword processor even outperforms the ideal VLIW processor in 6 out of 7 benchmarks. We believe this is mainly due to the high memory bandwidth offered by the superword processor. However, further investigation is required to verify this hypothesis.

For su2cor, we were unable to obtain results for all machine models. However, the partial results allow us to conclude that the superword machines outperform the VLIW machines for this benchmark. This follows because the performance of the superword-extended VLIW machine III will always be greater than that of the simple superword machine II. Also, machine V will always outperform machine IV because of the absence of inter-cluster communication. The results show that machine model II outperforms machine model V, thus we can infer that the missing superword machine model III will also outperform the missing VLIW machine model IV.

Figures 7, 8, and 9 show the number of instruction bits fetched compared to the simple machine model. Figure 7 shows these results for all operations and Figures 8 and 9 separate the results for ALU and memory operations, respectively. Superword instructions were counted once for each instance, whereas VLIW

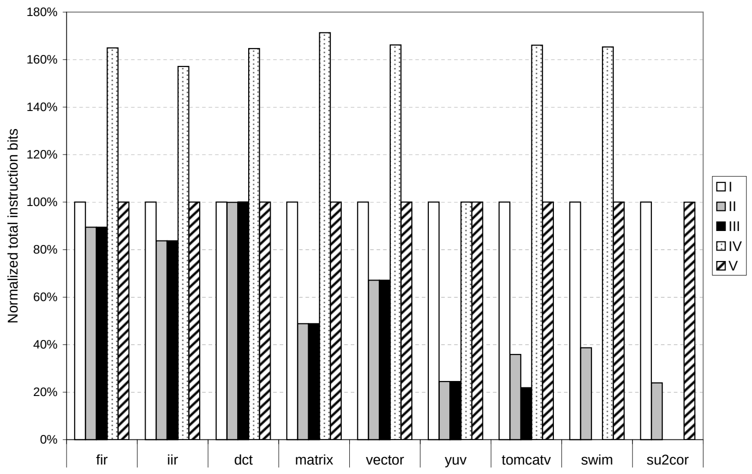

Figure 7: Normalized number of operations executed for each machine model.

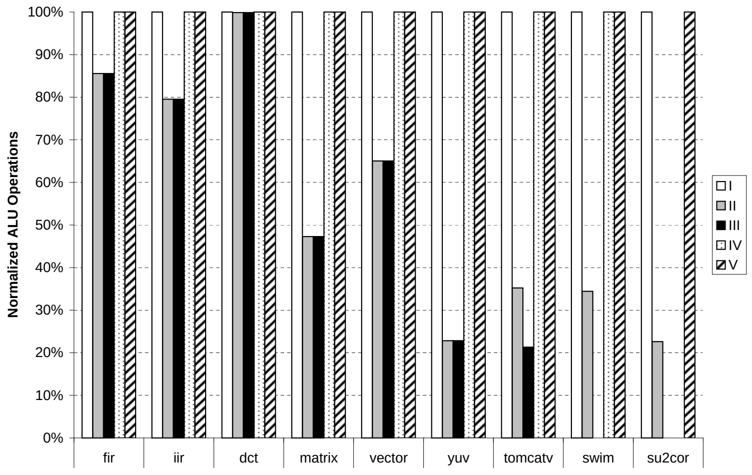

Figure 8: Normalized number of ALU operations executed for each machine model. Superword instructions are counted once, whereas VLIW operations are counted once for each used slot.

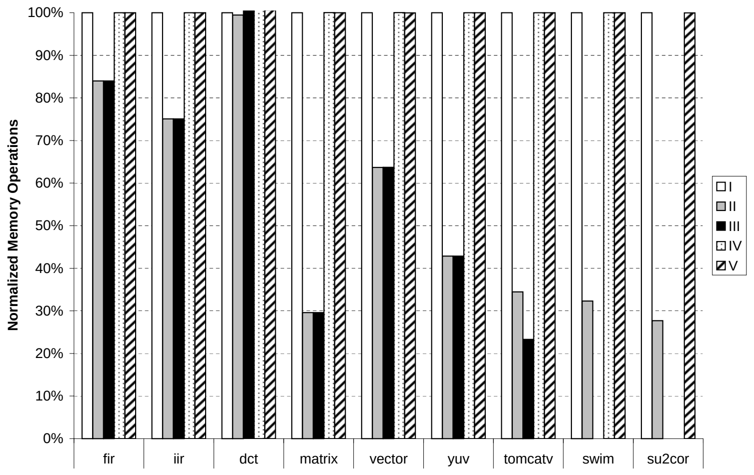

Figure 9: Normalized number of memory operations executed for each machine model.

instructions were counted by the number of filled issue slots. In the graphs, we assume a compressed VLIW instruction encoding, i.e. nop instructions are not counted.

These figures are meant to illustrate the relaxation on the instruction cache and instruction fetch unit when a SIMD encoding is used. The reduction in memory operations is particularly important because they have the added affect of improving data memory performance. Recall that this reduction is possible because of the banked memory configuration of a superword processor. We believe this has the greatest impact on the high performance attained by the simulated superword machines.

Figure 10 shows the instruction mix normalized to the total number of operations executed on the base machine. The key point in this figure is the high number of inter-cluster communication operations for the clustered VLIW machine versus the low number of pack and unpack operations on the superword machines. This indicates that our superword extraction algorithm is very effective at minimizing communication across lanes. This is important since communication is typically a high-latency operation.

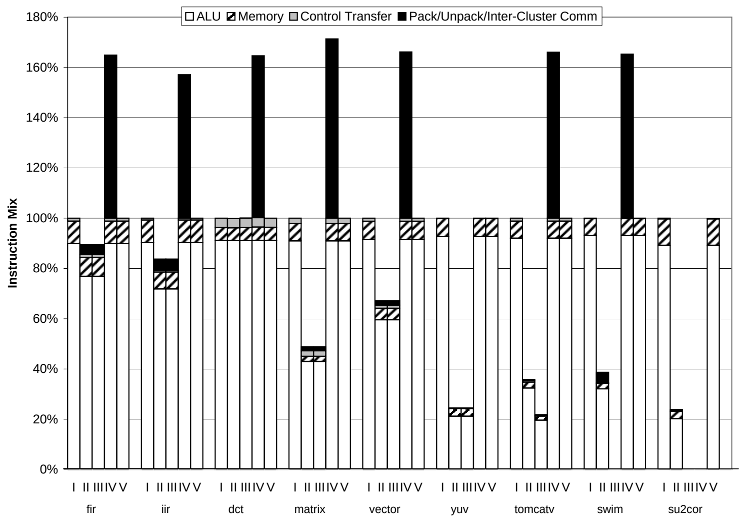

Figure 10: Instruction mix for each machine model.

7 Conclusion

This paper introduced superword processors, an architectural innovation in which a traditional processor datapath is replicated in SIMD fashion to provide a machine with a higher degree of parallelism. In addition to the promise of increased performance, superword processors have many desirable attributes. These include simple design, increased memory bandwidth, and backward compatibility with an existing base architecture. Furthermore, the superword enhancement is orthogonal to existing architectural structures, and can in fact be added to designs as varied as superscalars and VLIWs.

This paper also presented a set of compiler analyses that are necessary to take full advantage of the potential offered by the superword design. We have shown the effectiveness of the compiler and the architecture together through a comparison of a superword-enhanced VLIW to a clustered VLIW with comparable resources and complexity. Our initial results show that the superword processor outperforms the clustered VLIW by a factor of 2.43. Even compared to an ideal VLIW with identical functional units, the superword processor is still able to outperform the VLIW by a factor of 1.84.

Superword processors exhibit simplicity of design in both the compiler and the hardware, and still provide potential for performance gains in a large class of applications. As a result, we believe superwords can become a universally adopted architectural enhancement.

References

[1] http://www.eecs.harvard.edu/hube.

[2] Krste Asanović, James Beck, Bertrand Irissou, Brian E. D. Kingsbury, Nelson Morgan, and John Wawrzynek. The T0 Vector Microprocessor. In *Proceedings of Hot Chips VII*, Aug 1995.

[3] Giuseppe Desoli. Instruction assignment for clustered vliw dsp compilers: a new approach. Technical Report HPL-98-13, Hewlett Packard Laboratories, January 1998.

[4] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, Mar 2000.

[5] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel Processing: A Smart Compiler and a Dumb Machine. In *ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 37–47, June 1984.

[6] Craig Hansen. MicroUnity's MediaProcessor Architecture. *IEEE Micro*, 16(4):34–41, Aug 1996.

[7] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Yale University, 1985.

[8] Krste Asanović. *Vector Microprocessors*. PhD thesis, University of California at Berkeley, 1998.

[9] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 145–156, June 2000.

[10] Corina G. Lee and Mark G. Stoodley. Simple Vector Microprocessors for Multimedia Applications. In *Proceedings of the 31st Annual International Symposium on MicroArchitecutre*, pages 25–36, Dallas, TX, December 1998.

[11] Ruby Lee. Subword Parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, Aug 1996.

[12] David Maze. Compilation infrastructure for vliw machines. Master's thesis, Massachusetts Institute of Technology, September 2001.

[13] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing Streaming SIMD Extensions on the Pentium III Processor. *IEEE Micro*, 20(4):47–57, Jul 2000.

[14] Marc Tremblay, Michael O'Connor, Venkatesh Narayanan, and Liang He. VIS Speeds New Media Processing. *IEEE Micro*, 16(4):10–20, Aug 1996.

[15] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. 29(12):31–37, December 1994.