Width-Sensitive Scheduling for Resource-Constrained VLIW Processors

Tarun Nakra, Bruce R. Childers and Mary Lou Soffa
Department of Computer Science
University of Pittsburgh
{nakra, childers, soffa}@cs.pitt.edu

Abstract

As the width of processor instruction words increases, so do the opportunities for optimizations that exploit the widths of operands in instructions. This paper presents a feedback-directed technique, called width-sensitive scheduling, that packs operations on a functional unit, thereby enabling sharing of a functional unit among multiple operations. We target this technique as a static optimization to VLIW processors that are are resource-constrained, and use profile data to guide the optimization. We first discuss the significant factors in optimization using operand widths. We then describe and evaluate various approaches to optimizing operand widths on a realistic VLIW model. We find that there is sufficient potential, upto 13% speedup, for performance improvement using our technique. Packing of homogeneous operations on a functional unit is unable to exploit most of this available potential. An approach to pack heterogeneous operations on the same functional unit with minimal hardware changes is discussed. This approach exploits most of the available potential in performance savings. We also identify important factors that affect the efficacy of our technique.

1 Introduction

As processors have evolved to support wider instruction words, there has been an increase in the opportunity to optimize data widths. Although newer processors support up to 64 bits of data, the applications running on these processors rarely require the entire data width. There has been some work with compiler optimization and computer architecture to exploit data width by packing several operations together to execute on a single functional unit (FU) [2, 4]. Most of this work has focused on superscalar processors utilizing sub-word parallelism to pack similar (homogeneous) operations with narrow operands. This paper evaluates packing multiple narrow operands on resource-constrained VLIW processors. In particular, we present a static technique, *Width-Sensitive Scheduling (WSS)*, that packs operations of different types (heterogeneous operations) to execute on a FU.

Several processors have added support for small operand widths by enhancing their instruction set with operations for sub-word parallelism (SIMD instructions). Examples of these instructions sets include Intel MMX and SSE, AMD 3DNow!, Motorola Altivec, and HP MAX-2. Although these enhanced instruction sets encode sub-word parallelism in the form of SIMD instructions, most of the burden of detecting the parallelism is left to the programmer. Vectorization techniques have been proposed in the context of vector processors to exploit loop-level parallelism within scientific code [5]. In the case of applications running on fine-grained architectures, there has been recent work in hardware and compiler support for synthesizing SIMD-like instructions without user intervention. Brooks et al. [2] proposed an architecture that dynamically

packs narrow integer operations on an FU, similar to a parallel sub-word operation. Compiler support has been proposed by Larsen et al. [4] to synthesize SIMD instructions from basic block statements. Both of these recent studies are oriented toward superscalar processors. In this paper, we exploit smaller operand widths for VLIW processors.

One of the observations in previous studies is that multimedia benchmarks benefit greatly from SIMD synthesis because they typically operate on 16-bit data. Since multimedia applications are common for embedded systems, packing operations to synthesize SIMD-like instructions is very promising in this domain. However, as embedded systems move toward an EPIC-style Very Large Instruction Word (VLIW) computing model, previous work with SIMD synthesis for superscalars is not directly applicable and there is a strong need to address scheduling for narrow data types on VLIWs. The importance of optimization for embedded VLIWs is underscored by the popularity of recent architectures such as Starcore's SC140, Texas Instrument's TMS320C6201, and Transmeta's Crusoe. These processors are resource-constrained due to the stringent cost limitations of embedded systems, and by packing narrow width operations to execute on a single FU, we can take better advantage of hardware resources. Our width-sensitive scheduling technique directly targets embedded VLIWs that have limited hardware resources.

In the context of VLIW processors, exploiting narrow data widths has several unique challenges. Among these challenges is how to detect the width of operands that can be exploited by the compiler during scheduling. The detection can be performed either by static analysis or using run-time tools, such as profiling, which provide dynamic information. When profile information is used for predicting operand widths, the hardware needs to have the capability to recover from mispredictions. If the misprediction penalty is low, then the number of predictions can be increased to allow more operations to be packed on a FU. Another challenge is how to efficiently schedule operations using operand width information, particularly with the limited fetch bandwidth of embedded VLIW processors. A final challenge is whether there are enough opportunities to make statically scheduling narrow width operations worthwhile. In previous work related to operation packing [4, 2], only operations of the same type may be combined together for parallel execution on a FU. For VLIW architectures, we believe there is an unique opportunity to combine narrow width operations of different types without making significant hardware modifications. By adding support for heterogeneous operations, width-sensitive scheduling can uncover more opportunities for improving performance.

This paper discusses and addresses the challenges related to width-sensitive scheduling for embedded VLIWs. We present a scheduling technique that uses profile data to pack operations on a functional unit with minimal hardware changes. In the next section, we describe our scheduling technique, including a discussion of the challenges related to combining narrow data types. In Section 3, we evaluate the impact of width-sensitive scheduling on performance. Section 4 summarizes and concludes the paper.

2 Width-Sensitive Scheduling

2.1 Determining Operand Widths with Feedback

One of the important issues in packing operations with narrow operands is to determine the width of an instruction's operands. One approach to determine width uses dataflow analysis to compute bounds on the operand widths. Static bitwidth analysis has recently been proposed in the context of synthesizing reconfigurable architectures to identify the width of operands [7]. Although this information is accurate, it is conservative and cannot consider run-time information such as program input within

the analysis. In WSS, a feedback-directed approach is used to predict the width of operand values and speculatively pack narrow operations on a FU.

In VLIW processors, profile techniques have been widely used to increase parallelism by speculating operations. Using profiling for WSS is particularly well suited for embedded VLIWs where the applications are known and can be analyzed beforehand. Although static scheduling in a VLIW compiler allows aggressive scheduling over a larger window than the scheduling window of a dynamic schedulable processor, accurate run-time information is not available at compile-time. The compiler schedules operations conservatively preserving all control and data dependencies. Some of the conservative dependencies are overcome by speculating operations using profile information. Control speculation is used to move an operation above a branch by predicting the path to be taken after the branch. Data speculation is used to move loads above potentially non-conflicting stores (memory disambiguation), and to predict operand values to break true data dependencies (value prediction) enabling parallelism. We use speculation to pack narrow width operations in the same instruction. In each case, profile information is used to guide speculation by estimating the program behavior in terms of either branches taken or computed values.

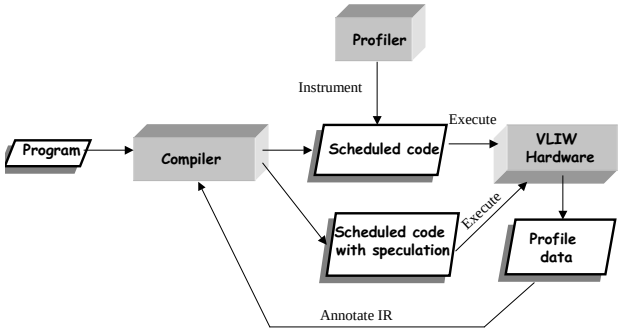


Figure 1: High-level view of profiling

The profiling procedure for WSS is shown in Figure 1. In this figure, the program is initially instrumented to collect profile information about the width of instruction operands. The program is executed with training inputs to gather the profile information, and its intermediate representation (IR) is annotated with the collected information about data width. Next, the compiler uses the annotated IR to generate an instruction schedule that packs operations with narrow widths. Instructions are packed on the same FU if the profile data indicates their operands to be narrow enough to share the FU.

2.2 Exploiting Fetch Bandwidth

As mentioned before, resource constrained VLIWs are limited by the amount of parallelism they can exploit. An example demonstrates such a case. Consider a five-wide VLIW processor with two integer FUs, and assume that the fetch unit is tightly coupled with the FUs. Hence, there is a one-to-one correspondence between the FUs and the fetched operations. Some VLIW processors with limited resources use this tight coupling to simplify issue logic. In Figure 2(a), we show part of a schedule for a region from the *_compress* function in the *compress* benchmark. Each operation is assigned the FU that would execute the operation. Note that although the instructions are independent, they need two cycles to be scheduled since only two integer operations can be accommodated per cycle. However, profile information indicates that each of the integer operations have operand widths that are at most 16 bits. This information can be used to pack two add operations on a single ALU and fetch them as a single SIMD operation. Performing this SIMD synthesis yields the schedule shown in Figure 2(b). In order to pack the two adds as a SIMD operation, the instruction set architecture (ISA) needs to have the capability to specify pairs of registers in an add operation. The reason is that most of embedded VLIW ISAs do not have vector registers.

One observation from Figure 2(a) is the processor has the capability to fetch five operations

Memory	Integer	Integer	FP	Branch
L_H <r 49> <r 15>	CMPP <pr 2> <r 4> i< 0>	ADD <r 20> <r 4> i<-1>		
	ADD <r 18> <r 25> i<2>	ADD <r 6> <r 4> i<–2>		

(a) Original Schedule

L_H <r 49> <r 15>	CMPP <pr 2> <r 4> i< 0>	ADD <r 6> <r 4> i<–2> <r 20> <r 4> i<-1>		
	ADD <r 18> <r 25> i<2>			

(b) Schedule after homogeneous packing

Figure 2: (a)Example of homogeneous operation packing

Memory	Integer	Integer	FP	Branch
L_H <r 49> <r 15>	CMPP <pr 2> <r 4> i< 0>	ADD <r 20> <r 4> i<-1>		
	ADD <r 18> <r 25> i<2>	ADD <r 6> <r 4> i<–2>		

(a) Original Schedule

L_H <r 49> <r 15>	CMPP <pr 2> <r 4> i< 0>	ADD <r 6> <r 4> i<–2>	ADD <r 20> <r 4> i<-1>	
	ADD <r 18> <r 25> i<2>			

(b) Schedule after homogeneous packing using fetch bandwidth

Figure 3: Example of homogeneous packing exploiting fetch bandwidth

per cycle, yet two of these slots, for branch and floating-point operations, remain unused in both cycles shown. Instead of coalescing similar operations together as a SIMD operation, the operations can be fetched separately using the empty slots. In this way, the ISA does not need to be modified to support SIMD operations. At runtime, the processor steers operations to an appropriate FU for execution as determined by the compiler. Using available fetch slots prevents an increase in register ports used per cycle. The steering involves redirecting the source operand values to the FU and the output value to the port that will write the value to the register file. This steering logic is not complex and similar logic is already provided in several VLIW processors that have more FUs than fetch bandwidth. Examples are Starcore's SC140 and Trimedia's TM-1000 processor. Although operand steering has some latency cost, we expect the additional latency would not affect the machine cycle time. The reason is that typically register file accesses determine machine cycle time and other pipeline stages scale proportionate to the register file cycle time. Previous studies make this observation in context of superscalar processors and expect similar performance limits for VLIW processors [3]. The unused fraction of the cycle time in the issue and execution stages can be used to perform the steering of the operation values.

To assign FUs to operations, the compiler annotates the operations with a FU identifier and the issue logic directs a fetched operation to its corresponding FU. This technique permits packing operations without generating SIMD instructions. Figure 3(b) shows the schedule of the previous example after packing operations using the FP slot. We expect that adequate unused fetch bandwidth can be made available for fetching additional ALU operations. From the analysis of several benchmark programs, we found that floating point and branch operations

Memory	Integer	Integer	FP	Branch
L_H <r 49> <r 15>	CMPP <pr 2> <r 4> i< 0>	ADD <r 20> <r 4> i<-1>		
	ADD <r 18> <r 25> i<2>	ADD <r 6> <r 4> i<–2>		

(a) Original Schedule

L_H <r 49><r 15>	CMPP <pr 2><r 4>i< 0>	ADD <r 20><r 4>i<-1>	ADD <r 6><r 4>i<–2>	ADD <r 18><r 25>i<2>

(b) Schedule after heterogeneous packing

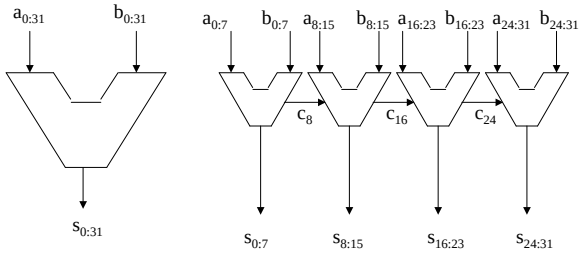
Figure 4: Example of heterogeneous packing

comprise less than 1% of dynamic instructions. This observation indicates that fetch bandwidth is severely under-utilized, and by scheduling narrow operations in unused slots, we can take better advantage of the available bandwidth.

2.3 Narrow Width Operations

In the example above, packing operations did not reduce the schedule length since one add operation had to be delayed by a cycle. One of the restrictions of packing operations in previous work is that only similar operations are allowed to execute together on a functional unit. A study by Scott et al. [6] observes that most of the benefits of packing homogeneous operations into SIMD-like instructions can be achieved by instruction-level parallelism on a machine with two or more ALUs. If heterogeneous operations are packed on a FU, the performance savings would potentially be higher. Packing heterogeneous operations permits scheduling all of the operations in the example above in one cycle, as shown in Figure 4(b). In this example, the add and compare operations can be scheduled together because they use at most 16 bits. The compiler assigns the position within the ALU for each operation allowing the issue logic to steer these operations to an appropriate ALU sub-component.

In order to pack heterogeneous operations together, a FU needs to execute operations of different types in parallel. It is possible to consider a 32-bit ALU as a group of 8-bit sub-ALUs executing the same operation and connected through carry lines. A pictorial view of a 32-bit ALU partitioned into sub-components is shown in Figure 5. Since an ALU is a combinational circuit of cascaded logic gates, this kind of partitioning can be made. If the sub-components were allowed to execute independently, each one would need a separate mode select to determine the type of operation to execute. Our work targets simple functional unit design applicable to processors limited by power and area constraints. There may be some design restrictions for partitioned FUs in multi-GHz processors. We are currently investigating the issues associated with next generation processor FU design and how our technique would address these issues.



(a) Normal 32-bit ALU (b) A 32-bitALU partitioned into 8-bit slices

Figure 5: ALU structure

Note that it is not possible to pack heterogeneous operations on an ALU using static SIMD synthesis because each combination of operations requires a separate SIMD opcode. For VLIW processors, the compiler can assign heterogeneous operations to the same ALU along with their placement positions within the

ALU. For superscalars, performing heterogeneous packing at run-time incurs additional complexity to select the operations to be packed. If 8-bit and 16-bit heterogeneous operations were interleaved on a 32-bit ALU, a processor with a central issue window would require additional control logic to find and select the operations to pack on an ALU. Packing heterogeneous operations works well for VLIWs because we can statically schedule the operations to avoid the run-time overhead of detecting narrow width operations.

2.4 Prediction and Recovery

One of the observations in the previous example is that scheduling using profile information about operand width decreases schedule length. Because this scheduling relies on profile information, we must predict the data width of operands, which requires a mechanism to recover from mispredictions at run-time. If we define a *prediction threshold(b)* for an operand to be the percentage of times that the operand is less than b bits wide, then we can vary the prediction threshold to control when to apply narrow width scheduling. By lowering the prediction threshold, more operations can be packed together. However, one of the drawbacks of lowering the threshold is that the misprediction rate increases. In case the operand width is more than its static prediction, the operation may not be computable with its assigned portion of the FU in the same cycle. Such a misprediction can be identified at run-time using zero detection logic that checks whether the upper 8 or 16 bits of an ALU result are zero. This detection logic is proposed by Brooks et al. [2] for their dynamic packing scheme and already exists in several modern processors.

One mechanism for recovery from data width mispredictions is to replay the mispredicted operation on a FU. In this case, when an operand's width is mispredicted, the operation is replayed through the same FU using the FU's entire width during the following cycle. During the replay of a mispredicted operation, subsequent operations are stalled in the pipeline. In the case of multiple mispredictions on a single FU, each mispredicted operation is replayed independently and the pipeline is stalled during the entire replay sequence. In this paper, we study the impact of varying prediction threshold on misprediction rate and its impact on performance.

3 Evaluation

3.1 Methodology

We studied the performance of WSS using the Trimaran compiler and simulator [1]. The simulator was extended to instrument programs and annotate the program IR with profile information. We also implemented our scheduling technique in the Trimaran compiler. To evaluate WSS, we used several benchmarks from the SPEC95 and MediaBench suites. Our machine model is a five-wide VLIW with two ALUs, one floating point unit, one memory unit, and one branch unit. This machine is similar to the Transmeta Crusoe TM5400 processor which is a four-wide VLIW with one ALU. However, the Crusoe processor can use its memory unit as an adder when no memory operations are scheduled, effectively making the processor five-wide. We use a fairer model of a five-wide VLIW that is less constrained by allowing two ALUs. Our machine model has 64 general-purpose registers and a 32-bit data width. The Trimaran compiler does several aggressive code optimizations including dead code removal, redundancy elimination and other classical optimizations before performing WSS. Only ALUs were packed with more than one operation using WSS. The profile data included how often an operand width of an instruction was (1) less than 8 bits and (2) less than 16 bits. This data was analyzed by the compiler to identify instructions whose operands were within 8 or 16 bits. For example, if profile data indicated an instruction to always have operands within 16 bits, the ALU as-

signed to the instruction was considered as 50% busy. The other half of the ALU was available to other instructions with narrow operands of 8 or 16 bits. The compiler used a prediction threshold of 100% for both 8-bit and 16-bit operands. In case an instruction's operand spilled over its predicted width, the processor stalls a cycle to re-execute the instruction.

3.2 Fetch Bandwidth and Register Pressure

For our initial studies, we were interested in analyzing the potential performance of packing operations without bandwidth limitations. Hence we assumed there was always enough bandwidth to supply operations to fully pack an ALU. Also operations packed on an ALU were allowed to be of different types. Later in this paper, we remove these assumptions. Figure 6 shows the percentage speedup for packing operations. The first column in the figure assumes perfect register allocation with no register spills and the second column incorporates register spills. The savings in absence of register spills are in the range 1-13% with an average of 5.7%. Another observation is that register spills become important and affect performance considerably. Since width-sensitive scheduling increases parallelism, register pressure also increases. This reduction is especially noticeable in case of *ijpeg*. Speedups for our benchmarks after considering register spills are in the range 1-8% with an average of 2.6%. The impact of spills is particularly significant in our machine model because there is only one load/store unit.

From the above observations, we infer that in order to gain performance benefits from width-sensitive scheduling, the register allocator needs to be sensitive to operand widths. One intuition is that 16-bit values can be spilled in consecutive locations and fetched in a single load operation. In our current implementation, width-sensitive scheduling only packs ALU operations, so spills and reloads of narrow width data are performed

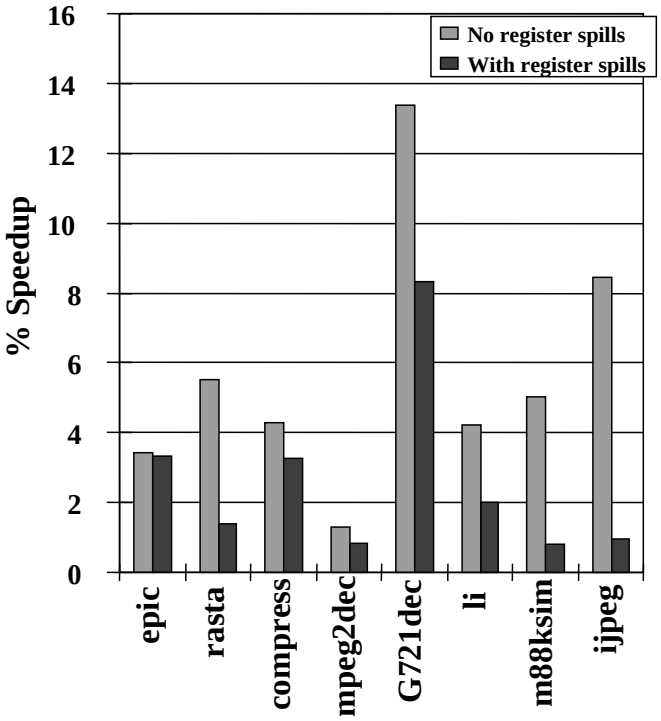
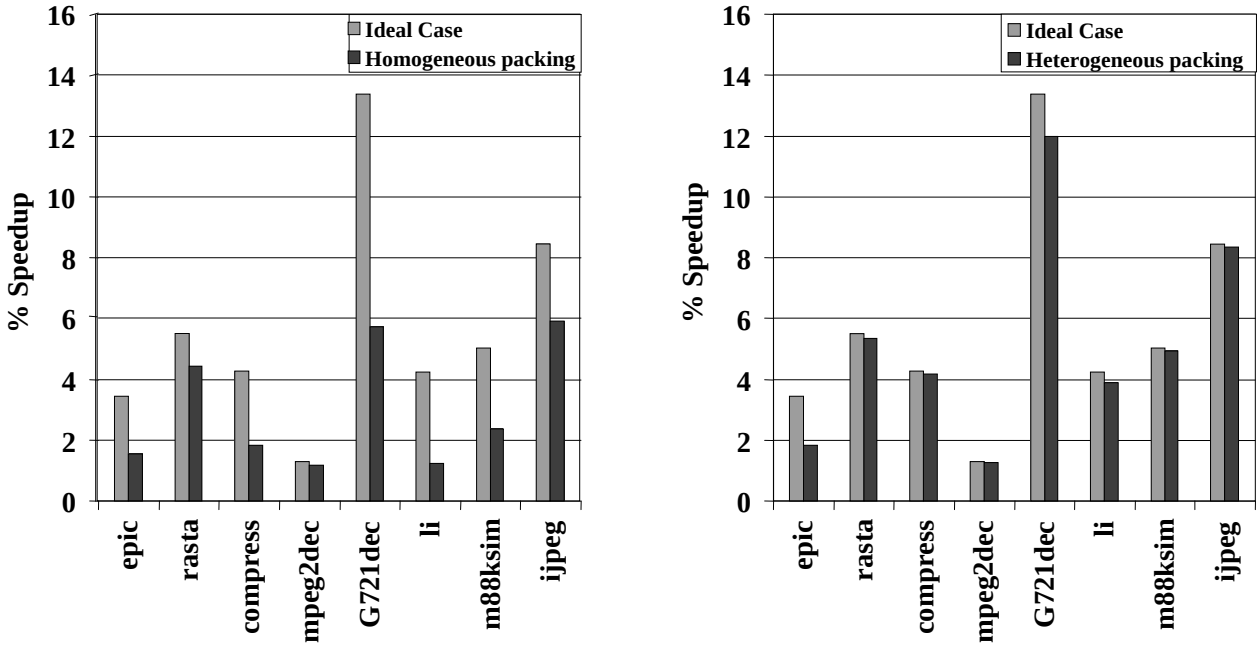


Figure 6: Performance with no limitation on fetch bandwidth

with separate store and loads. By packing narrow operands for spills, we expect the impact of spilling to reduce. This packing is similar to SIMD synthesis of load/stores [4]. However, since register allocation is a static technique, we can also modify the allocation phase to accommodate spills of 16-bit values to the same register, which will significantly reduce the number of spills. This width-sensitive register allocation can be incorporated within a compiler independent of the underlying architecture. We are currently investigating methods to reduce the impact of register spills using techniques similar to the ones proposed here. For our analysis in this paper, we were interested in evaluating the scope of width-sensitive scheduling for performance improvement. To avoid the impact of spills on this performance study, we assume perfect register allocation with no spills for the rest of our experiments.



(a) Homogeneous packing

(b) Heterogeneous packing

Figure 7: Performance for limited fetch bandwidth

3.3 Homogeneous vs. Heterogeneous Packing

In the next set of experiments, we wanted to analyze how constraining bandwidth affects performance. Hence the fetch bandwidth was fixed to be the width of the machine. Also, we wanted to study how well packing heterogeneous operations improves performance over SIMD-like synthesis. Two sets of experiments were carried out. First, only homogeneous operations were packed on an ALU. This technique is similar to the synthesis of SIMD instructions. In the second set, we allowed heterogeneous operations to be packed on an ALU. Figure 7 shows performance for both experiments. Each case is compared with the performance of the case with no limitation on fetch bandwidth (labeled as ideal case). From Figure 7(a), we observe that speedups in the case of packing homogeneous operations are considerably reduced from the ideal case. This reduction relates to the fact that fetch bandwidth is limited and only similar operations are allowed to share an ALU. For the case of heterogeneous packing, Figure 7(b) shows that performance improvements are close to the ideal case. This observation implies that heterogeneous packing uncovers significant parallelism that is not exploitable with homogeneous packing. Also, in the latter case, the unused fetch bandwidth can be used to exploit most of the possible benefits of the ideal case.

3.4 Prediction Threshold and Misprediction

As discussed previously, prediction threshold can have an impact on detecting opportunities to pack operations. To determine whether this was the case, we varied the threshold and measured the impact on performance. We found that as the prediction threshold was lowered, the sched-

ule length improved but the misprediction rate also increased. It was observed that the improvements in schedule length were minor unless the prediction threshold was below 60%. However, for such low thresholds, the cost of recovering from mispredictions mitigated any improvement in schedule length. With our current recovery mechanism, each mispredicted operation is replayed independently and contributes a cycle to the misprediction penalty. In case of complex arithmetic operations with latency of several cycles this penalty can be more than one cycle. In our experiments we observed that less than 1% of packed operations had latency higher than one cycle (except in case of *G721dec* with 3.3%). Accounting for multi-cycle misprediction penalty would have a negligible effect on performance.

It is possible to achieve higher speedups with lower prediction thresholds by reducing the misprediction penalty. One way to reduce the penalty is by recovering multiple predictions in parallel with execution of the next VLIW instruction, if resources are available. Also more sophisticated prediction techniques may be able to predict more accurately reducing the instances of misprediction. We are currently investigating both compiler and hardware techniques to reduce the misprediction penalty.

3.5 Machine Width

In our next experiment, we varied the machine width to determine in what context width-sensitive scheduling is most appropriate. Three different machine models were considered: (1) a four-wide VLIW with one load/store, one ALU, one floating-point and one branch unit, (2) a five-wide VLIW with one load/store, two ALU, one floating-point and one branch unit and (3) an eight-wide VLIW with two load/store, four ALUs, one floating-point and one branch unit. These machine models were selected based on a range of embedded processors from a simpler model (four-wide Transmeta TM5400) to a relatively complex one (eight-wide TI TMS320C6201).

Performance analysis for the different machine models is shown in Figure 8. From this figure, the narrow VLIW machine has considerable improvement, ranging from 11-25% with an average of 15.7%. However, as more ALUs are added to the machine model, these savings reduce. Performance improvements for the eight-wide machine model were insignificant, ranging from 0.1-3%. The reason is that this machine model has sufficient integer units to exploit most of the parallelism available by packing narrow operands. This observation confirms the statement that width-sensitive scheduling is beneficial for resource-constrained VLIWs, similar to models 1 and 2. In our experiments, a 32-bit data width is used and a wider data path would allow more operations to be packed together on a functional unit. From Figure 8, our techniques substantially improve the performance for narrow VLIWs, and we expect the performance improvements would be even greater for VLIWs with a wider data width.

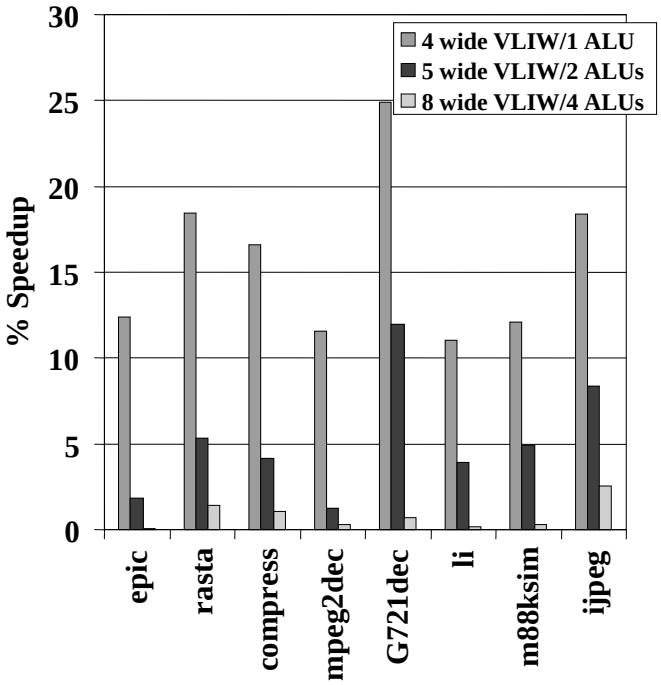


Figure 8: Performance for different machine models

4 Summary and Future Work

In this paper, we present a scheduling technique, called width-sensitive scheduling, that allows sharing of a functional unit among operations that have narrow width operands. The technique is proposed as a static scheduling technique and applied to resource-constrained VLIW processors. The important factors associated with our scheduling technique are presented and evaluated on a realistic VLIW machine model. The performance savings due to width-sensitive scheduling are very encouraging, with speedups of 1-13%. Packing operations of different types on the same functional unit has better performance than packing operations in a SIMD-like fashion. We find our technique to be most suitable for VLIW processors that are resource-constrained, such as those used in embedded systems. Our experiments indicate that register allocation plays an important part in speculation of operand widths, and our techniques could be used along with allocation of live ranges to registers based on width of the live range value. We are currently analyzing the improvements due to this combination. We also believe our technique would have even better performance with an improved misprediction recovery mechanism. In a future paper, we will describe our work with bit-sensitive live range analysis and improved misprediction recovery.

5 Acknowledgments

We would like to thank the anonymous reviewers whose comments and suggestions helped improve the paper.

References

[1] Trimaran compiler research infrastructure. In *Tutorial Notes, http://www.trimaran.org*, Nov 1997.

[2] David Brooks and Margaret Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 13–22, Orlando, FL, January 9–13, 1999.

[3] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. Register file design considerations in dynamically scheduled processors. In *WRL Research Report 95/10, Western Research Laboratory, Compaq*, 1995.

[4] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 145–156, Vancouver, BC, June 18–21, 2000.

[5] C. G. Lee and D. J. DeVries. Initial results on the performance and cost of vector microprocessors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 171–183, RTP, NC, December171–182 1997.

[6] Kevin Scott and Jack Davidson. Exploring the limits of sub-word parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Philadelphia, PA, October 15–19, 2000.

[7] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 108–120, Vancouver, BC, June 18–21, 2000.