

Instruction-Level Parallel Processors— Dynamic and Static Scheduling Tradeoffs

Kevin W. Rudd*
Michael J. Flynn

Stanford University Computer Systems Laboratory
E-mail: {kevin, flynn}@Umunhum.Stanford.EDU

Abstract

Recently, high-performance computer architecture has focused on dynamic scheduling techniques to issue and execute multiple operations concurrently. These designs are complex and have frequently shown disappointing performance. A complementary approach is the use of static scheduling techniques to exploit the same parallelism. In this paper we describe some of the tradeoffs between the use of static and dynamic scheduling techniques and show that with appropriate scheduling, low-complexity designs using only static scheduling have significant advantages over high-complexity designs using dynamic scheduling in real systems.

1. Introduction

High-performance processor design has recently taken two distinct approaches. One approach is to increase the execution rate by increasing the clock frequency of the processor or by reducing the execution latency of the operations. While this approach is important, much of its performance gain comes as a consequence of circuit and layout improvements and is beyond the scope of this research. The other approach is to directly exploit the instruction-level parallelism (ILP) in the program and to issue and execute multiple operations concurrently. This approach requires both compiler and microarchitecture support.

Traditional processor designs that issue and execute at most one operation per cycle are often called *scalar* designs. Static and dynamic scheduling techniques have been used to achieve better-than scalar performance by issuing and executing more than one operation per cycle. While Johnson[7] defines a *superscalar* processor as a design that achieves better-than scalar performance, popular usage of this term

refers exclusively to those processors that use dynamic scheduling techniques. For clarity, we use *instruction-level parallel* processors to refer to the general class of processors that execute more than one operation per cycle.

The primary static scheduling technique uses the compiler to determine sets of operations that have their source operands ready and have no dependencies within the set. These operations can then be scheduled within the same instruction subject only to hardware resource limits. Since each of the operations in an instruction is guaranteed by the compiler to be independent, the hardware is able to issue and execute these operations directly with no dynamic analysis. These multi-operation instructions are very long in comparison with traditional single-operation instructions and processors using this technique have been called *Very Long Instruction Word (VLIW)* processors.

The primary dynamic scheduling technique uses special hardware to analyze the instruction stream at execution time and to determine which operations in the instruction stream are independent of all preceding operations and have their source operands available. These operations can then be issued and executed concurrently. Processors using dynamic scheduling techniques have been called *superscalar* processors.

Both of these techniques have advantages and disadvantages and both have demonstrated significant problems that limit their achievable performance despite claims of strong peak performance. Statically scheduled processors require that the latencies of all operations be fixed and known in advance. Because statically scheduled processors do not support dynamic dependency detection, this restriction limits the changes that can be made in an architecture to those which do not affect operation latencies. However, the absence of dynamic analysis hardware allows fast and simple implementations. Dynamically scheduled processors require a significant amount of hardware to perform the instruction stream analysis (which scales as $O(n^2)$ where n is the number of operations being analyzed). This analysis

*This work was supported in part by U.S. Army Research Office under grant DAAH04-95-1-0123

can rapidly become the critical path in the processor and can lead to an increase in the cycle time or the pipeline depth. However, the presence of dynamic analysis hardware does not require that the compiler know anything about operation latencies allowing significant flexibility in implementation changes and code generation techniques.

Although dynamic analysis is useful, in the typical superscalar processor there is a significant amount of redundant analysis performed by the compiler and the hardware. Ignoring issues of code compatibility, this redundancy is inefficient and undesirable. Finding a reasonable dividing line sharing the work between the compiler (static scheduling) and the hardware (dynamic scheduling) will lead to improved performance.

This paper examines some of the tradeoffs of using static and dynamic scheduling techniques in instruction-level parallel processors and examines the question as to where the dividing line should lay between hardware and software techniques. After discussing both of these scheduling techniques we define a universal ILP microarchitecture, describe our experimental framework, and introduce the design space that we are evaluating. We then present our results including some surprising trends.

2. Instruction-level parallel microarchitectures

Examining VLIW and superscalar architectures we see that both instances of instruction-level parallel architectures require the same instruction stream analysis in order to exploit the available ILP—they only differ in where this analysis occurs. There are two phases of this analysis that must be performed to properly schedule and issue the operations:

1. Identify and schedule independent operations
2. Issue ready operations for computation

The two analysis phases are fairly independent of each other and each can be performed using a range of static and dynamic techniques. Figure 1 represents the four combinations of schedule and issue approaches (both limited to static and dynamic for simplicity) and places both canonic and typical superscalar and VLIW processors in the matrix appropriately. Figure 1 also includes expanded instruction caches (Johnson [6]) as an instance of dynamically scheduled and statically issued microarchitectures.

In a VLIW architecture, the first phase is performed exclusively by the compiler and the second phase is a trivial task performed by the hardware. In a superscalar architecture, the first phase is likely performed by the compiler but both phases must be performed by the hardware.

In fact, neither VLIW nor superscalar processors rely completely on static or dynamic scheduling. VLIW processors often use limited dynamic behavior to improve performance, primarily focusing on the memory system: the

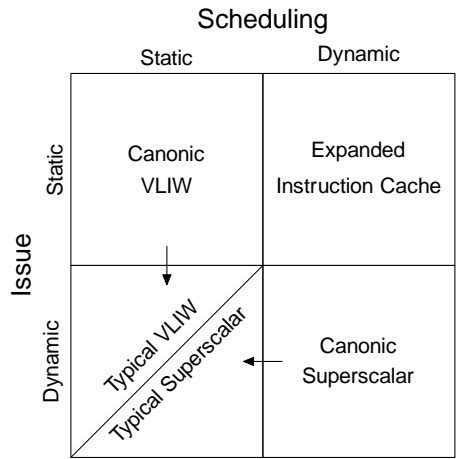


Figure 1. Instruction-level parallel schedule-issue regions

Cydra-5 processor [14] includes the ability to deal with adjustable memory system latencies under program control; the Philips TriMedia TM-1 processor [16] includes a high-performance cache subsystem that supports non-blocking references, memory regions locked in the cache (under program control), and dynamic stalling for delayed memory values. Superscalar processors do not always depend as much on dynamic analysis as one might suppose and many designs have simplified the instruction stream analysis to limit complexity: the HP PA-8000 [4] processor can schedule at most 4 operations per cycle despite being able to track a total of 56 operations and to manage 10 function units. Even so, the level of dynamic issue analysis in a superscalar processor is far greater than the level of dynamic issue analysis in a VLIW processor (which is typically limited to stalling the processor on a late result or use). There is a clear overlap in the area of dynamic issue analysis that recommends further study of the second analysis phase—that performed internal to the microarchitecture of an instruction-level parallel processor.

While VLIW and superscalar take different approaches to issuing and executing these operations, the source of these internal packets of operations does not have a direct effect. Whether the instruction stream consists of instruction packets of independent operations or a sequence of potentially dependent single-operation instructions, the microarchitecture itself remains relatively unaffected—it eventually sees a collection of independent operations and issues ready operations to available function units. Because we are looking at microarchitectural complexity, we are not interested in the specific technique used to construct packets of independent operations, we presume that the instruction stream is provided in a packetized form. This gives a best

case schedule since (in our case) the scheduling is done by the compiler with the entire program text available to it for analysis. This is in contrast to the traditional superscalar instruction window which can analyze only a few instructions (on the order of 4–8) for scheduling. We will review what effect this optimistic scheduling will bias the results in favor of processors using dynamic analysis hardware.

We have developed a trace-driven simulator that implements the execution model described in [15]. This model characterizes each operation as a triple consisting of the source operands, specific computations, and destination operands. It also distinguishes between the *architectural* (programmer visible) and *implementation* (as-built) specifications. A specification consists of a set of operation triples which define the behavior of the architecture. Despite differences between the architectural and implementation specifications, an implementation guarantees that the perceived behavior is precisely that of the architectural specification, thus providing the illusion of a virtual processor just as memory management hardware and the operating system provide the illusion of a virtual memory system. In both cases, the distinguishing characteristic between virtual and real systems is the actual performance—the program is unaware of any difference. This model is a general model that can be applied to a wide range of operation definitions—including complex operations that have indeterminate or iterative execution patterns!—and is based on earlier work by Rau [12, 13].

3. Simulation framework and methodology

As an evaluation platform, we define a “universal” ILP microarchitecture that we use as to evaluate processor configurations using a range of dynamic analysis techniques to issue operations to function units. Based on a number of studies of available ILP [18, 8] we have defined a nominal architecture that ensures that our results are not significantly affected by the use of an unreasonably narrow or wide configuration. Using this nominal architecture we are able to vary a number of aspects in order to explore the effects of these changes on performance.

For the nominal architecture we use a configuration consisting of two of each kind of function unit—integer, floating point, load/store, and branch—for a total of eight function units. This configuration allows the support of a reasonable level of ILP as well as allowing the exploration of the performance impacts of double and half wide—four and sixteen function unit—versions to understand how these changes affect performance.

We make a further simplifying assumption that the cycle time for all processor configurations remains constant and is unaffected by the complexity of any dynamic analysis hardware. While this choice simplifies the experiment,

it potentially complicates the data analysis—by not considering the complexity and cost of the analysis logic we are biasing the results in favor of processors using these techniques. We will revisit this bias in the conclusion and its impact on these results.

The architectures that we studied share a common set of features. The memory system consists of split instruction and data caches, a pipelined bus, and a main memory system. The instruction cache is 32 kB with 128 B lines (4 instructions per line); the data cache is 32 kB with 64 B lines. Both are direct mapped. Memory is reached through a pipelined 64 b bus using an 8 operation deep load/store queue. Branch prediction is performed using a 1024 entry direct-mapped counter-based history mechanism. Nominal function unit latencies are 1 cycle for integer operations and 2 cycles for floating point operations (8 and 15 cycles for floating point divide).

Around this common core we define five virtual architectures. One architecture has all operations completing in a single cycle and is used to represent a typical scalar processor instruction stream. The other four architectures are variations of the nominal architecture with the assumed/actual latency to the data cache scaled by multiples of 1 (the nominal architecture), 2, 4, and 8 times.

In order to explore the effects of the memory system on these architectures we use memory latencies of 4 cycles (approximating the effective latency when using a second level cache), 25 cycles (a nominal value for a moderate system), and 100 cycles (representing a system with a high-performance processor and a realistically slow main memory). To establish a measure of an upper bound on performance we also consider the case where the implementation has perfect branch prediction and perfect caches.

While the caches block on a miss (causing further references to stall until the miss is completed), this restriction only affects the relative cost of memory for the different applications. Non-blocking caches allow subsequent memory references to be serviced by the cache during miss processing and rely on the independence of memory references. Depending on the nature of the application’s memory reference pattern, the benefits of a non-blocking cache will vary with the worst case being equivalent to a blocking cache. Because the gain provided by non-blocking caches is program (and data) dependent, the performance for an architecture cannot be compared across the different benchmarks because each benchmark has different memory reference stream characteristics. However, the general characteristics of increasing memory latency still hold.

In order to see how performance varies with complexity, we vary the number of pending instructions analyzed in the hardware. In each cycle, there are a certain number of instructions available from which ready operations can be issued to function units—the greater the number of pending

| Benchmark | Static | | | Dynamic | | | Simulation samples |
|--------------|----------------------|---------------------|---------|---------------------|--------------------|---------|--------------------|
| | operations | packets | ops/pkt | operations | packets | ops/pkt | |
| 008.espresso | 104.47×10^3 | 51.22×10^3 | 2.04 | 10.12×10^6 | 3.82×10^6 | 2.65 | 50 |
| 022.li | 21.14×10^3 | 11.83×10^3 | 1.79 | 8.66×10^6 | 3.81×10^6 | 2.27 | 43 |
| 052.alvinn | 5.41×10^3 | 2.34×10^3 | 2.32 | 36.0×10^6 | 9.23×10^6 | 3.90 | 180 |
| 056.ear | 12.15×10^3 | 6.03×10^3 | 2.02 | 115.0×10^6 | 38.4×10^6 | 2.99 | 575 |

Table 1. Summary of benchmark characteristics

instructions, the greater the number of operations available for issue. This is analogous to varying the size of the instruction pool (reservation station, reorder buffer, *etc.*) in a superscalar processor.

We have simulated pending instruction window sizes from 1 to 64 pending instructions. These configurations cover the range from a simple in-order processor (with a window of 1 instruction) to a massively out-of-order processor (with a window of 64 instructions). Because the initial results have shown that the benefits from out-of-order issue rapidly saturate, we limit our results to architectures with window sizes up to 8 instructions.

We use four benchmarks from the SPEC-92 benchmark suite (two integer, two floating point) to analyze the performance of the different configurations. These benchmarks were chosen because they give a range of integer and floating-point as well as loop-based and table-search control-flow patterns. The basic characteristics for these benchmarks (nominal latency schedule) are shown in table 1.

4. Results

We have generated and analyzed data from simulations using the configurations and benchmarks described in the previous section. These have shown us a number of interesting and sometimes surprising results. The first result is the dependence of the performance of the code schedule technique on the level of dynamic analysis performed by the hardware. The second result is that as the mismatch between processor and system performance increases (as is the case with processor performance improving at a faster rate than memory system performance), the benefits of dynamic analysis hardware are reduced so that all configurations yield similar performance.

A third result observed is that there is little benefit to increasing the number of function units beyond that assumed by the schedule. Simulations with double the number of function units showed only slight improvements over the nominal architecture. On the other hand, reducing the number of function units to a half of nominal resulted in performance comparable to a half-wide schedule configuration. There is little benefit in having a configuration that exploits

more ILP than is available in the schedule (or program).

The correspondence between the schedule (virtual architecture) and the actual machine (implementation architecture) is described by a qualitative *matching* value that has three basic values. When the virtual and implementation machines agree we say that the schedule is *well matched*. Accordingly, a schedule that is not well matched is *poorly matched* and may be either *undermatched* (when the virtual latencies are shorter than the implementation latencies) or *overmatched* (when the virtual latencies are longer than the implementation latencies). While it is possible to have some virtual latencies that are shorter and other virtual latencies that are longer than the implementation latencies, we have not considered these configurations and do not define any particular naming for these cases.

The matching value is important because it qualitatively describes the nature of the code stream as seen by the processor. An undermatched schedule has operations scheduled *before* their source values become ready giving a densely packed schedule. A well matched schedule has operations scheduled *as* their source values become available giving a schedule that matches the expected level of ILP in the schedule. An overmatched schedule has operations scheduled *after* their source values become available giving a sparsely packed schedule. This can be seen in table 2 which shows the operations per packet for the instruction streams for the benchmark 022.li.

The effects of these three kinds of matching can be seen in figure 2. In this figure, the x-axis is the number of instructions analyzed at any given time to issue computations to the function units (a value of 1 is effectively a statically scheduled VLIW architecture which stalls whenever a source or destination operand is not available as scheduled or used); the y-axis is the achieved performance in operations per cycle (the limit for performance is 8 operations—the number of operations in an instruction).

The data in figure 2 is for the benchmark 022.li with perfect caches and branch prediction (figure 5 shows the same data for all four benchmarks). By using perfect caches we are able to focus on the variations due to schedule matching. This figure shows the general behavior that is seen throughout our results. The performance of the undermatched schedule increases dramatically up to the point

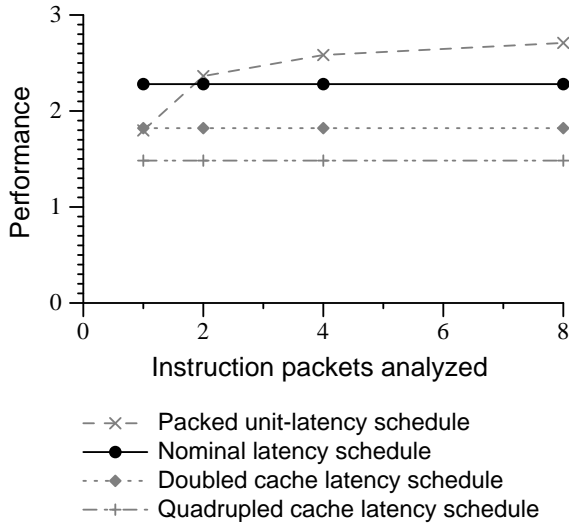


Figure 2. Achieved performance in operations per cycle of 022.1i (perfect cache and branch prediction)

that the number of dynamically analyzed instructions covers most of the implementation latency for the operations in the instruction stream; after this point the gain rapidly tops out. For an implementation with only a single dynamically analyzed instruction, any mis-scheduled operation will cause the processor to stall resulting in a performance loss. Since most operations are either 1 or 2 cycle latency operations, most of the gain is achieved by this point. The actual cross-over point will vary as the operation mix varies between programs. The well matched schedule has flat performance over the range of dynamically analyzed instructions considered. With this schedule, all operations complete as scheduled. The overmatched schedules with scaled memory access latencies—the doubled and quadrupled cache latency schedules—also show flat performance but have lower achieved performance due to the poor schedule utilization.

It is instructive to look more closely at these results to understand the performance of the three scheduling classes. To do this we must examine the concepts of operation density and schedule efficiency. Operation density is a potential measure of program performance—the average number of operations per instruction packet. Program efficiency is a measure of how well the program ran in contrast to how well it should have run—the ratio of the scheduled execution time to the actual execution time. Neither operation density nor schedule efficiency alone is sufficient to compare performance values: while the operation density is a measure of the available ILP, a schedule may stall frequently resulting in poor overall performance; similarly, while a schedule

| Scheduling approach | Operation density |
|-----------------------------------|-------------------|
| Packed unit-latency schedule | 2.75 |
| Nominal latency schedule | 2.28 |
| Doubled cache latency schedule | 1.82 |
| Quadrupled cache latency schedule | 1.49 |

Table 2. Instruction stream density (operations per packet) of 022.1i (perfect cache and branch prediction)

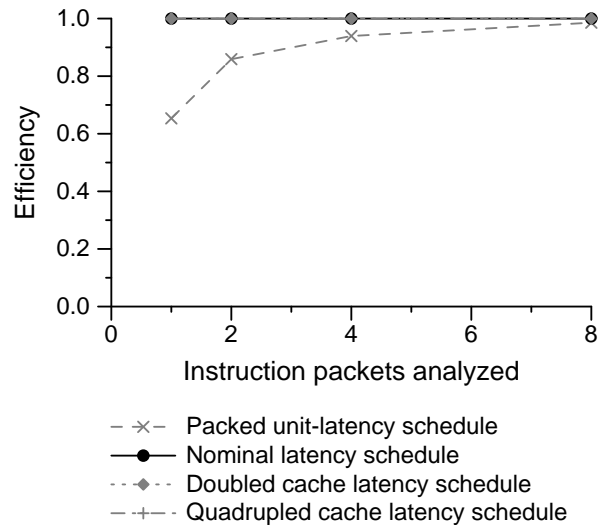


Figure 3. Schedule efficiency of 022.1i (perfect cache and branch prediction)

may run very efficiently, it may not utilize the exploitable ILP and again result in poor overall performance. However, the product of these two values yields program performance in terms of operations per cycle which is a useful measurement of performance between benchmarks, schedules, and configurations.

Table 2 shows the variations in the operation density for a range of schedules for 022.1i. As can be seen, the operation density is greatest for the undermatched schedule and decreases steadily from undermatched to well matched to overmatched. Figure 3 shows the differences in the efficiency of the processor for the same schedules. We see that the well matched and overmatched schedules have perfect efficiency. This should come as no surprise since the caches are perfect and all operations complete at or before their scheduled time—there is no delay due to the schedule. In addition to having poor efficiency (due to the frequent stalls from the optimistic schedule), the undermatched schedule

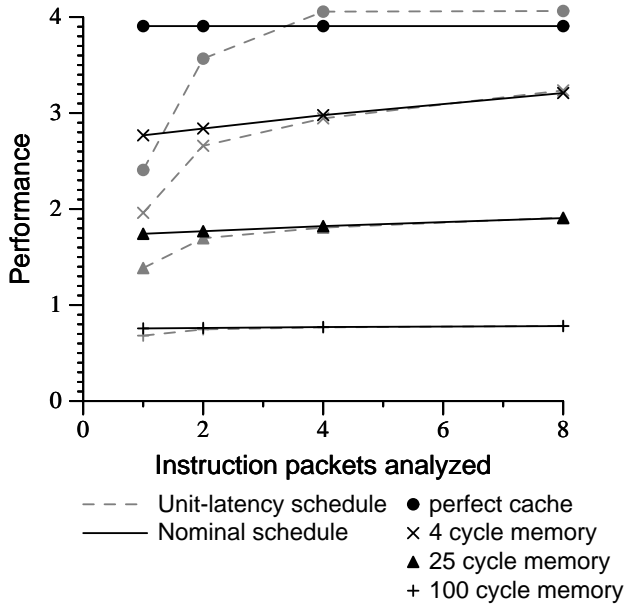


Figure 4. Comparison of performance of 052.alvinn (varying memory latencies)

requires a significant amount of analysis hardware to issue around the many delayed operations.

Even though the efficiency of the undermatched schedule never reaches that of the well matched schedule, the increased number of operations available for dynamic analysis more than compensates for the inefficient schedule and results in improved performance over the other configurations. However, although the effect of the increased operation density provides a large pool of operations available for issue, the performance of the undermatched schedule is not always better than that of a well matched schedule. This overall behavior also holds in the presence of cache misses since all schedules suffer the same cache misses but the undermatched schedule suffers stalls due to optimistic scheduling as well.

Next we examine the performance variations that occur in the presence of realistic memory latencies. Figure 4 shows the behavior of 052.alvinn as the memory system performance varies from a perfect cache to a latency of 100 cycles on a cache miss (figure 6 shows the same data for all four benchmarks). Just as in figure 2, the overmatched schedules show increasingly poor performance and are not shown here. Surprisingly, a memory latency of only 4 cycles to main memory shows that there is little benefit from the improved code density of the undermatched schedule even in configurations with much dynamic analysis hardware.

There is a small performance improvement as the dy-

amic analysis hardware is able to exploit out-of-order operation issue. As the number of dynamically analyzed instructions increases, there is a related increase in performance. This is most significant in the 4 cycle memory configuration (and is clearly absent from the perfect cache configuration!). As the memory system latency increases the benefit from this behavior diminishes to the point that at realistic main memory latencies there is again little to no advantage to the dynamic analysis hardware. The fact that the fraction of real time spent waiting for the memory system becomes greater and greater as the latency increases is a direct application of Amdahl's law [1]. In this case, it does not matter how fast we can execute operations in the processor if we are limited by the speed of the memory system.

5. Conclusions

These results lead to one basic conclusion—in the presence of significant memory latencies there is little difference in performance across a wide range of simple and complex processors. With memory systems performing 100 times more slowly than high performance processors, this is a significant consideration. There is limited benefit from dynamic dynamic analysis hardware in some benchmarks (as seen by 052.alvinn in figure 4), yet the benefit rapidly diminishes to the point that it is not clear that the performance improvements are worth the complexity. The ability to support two pending instructions gives a significant portion of the achievable benefit while limiting the complexity of the analysis hardware dramatically.

As was noted in earlier, we do not consider the complexity of the dynamic analysis logic and its cycle time impact. We need to consider both the assumption that the instruction stream consists of packets of independent operations as well as the assumption that the cycle time is fixed and unaffected by the number of outstanding instructions analyzed. What is the effect of this analysis hardware?

Dynamic analysis hardware that detects and schedules independent operations can produce no better performance than the compiler can with its static schedule—while compilers can look at the entire program, hardware has a very limited analysis window. Both the reduced quality of the schedule and the likely increase in cycle time combine to reduce overall performance.

Dynamic operation issue logic may affect the cycle time but may also result in improved performance by reordering around delayed operations—this reordering cannot be exploited by static scheduling. The effective performance impact of dynamic operation issue hardware is impossible to determine in the general case—although it is clear that the addition of this hardware will not reduce the cycle time! Again, any effect would be to reduce overall performance.

Our results show that the type of static schedule is im-

portant only with little to no dynamic analysis hardware—in this region the well matched schedule is vastly superior to both undermatched and overmatched schedules. When dynamic analysis hardware is added, the differences between the undermatched and well matched schedules are reduced and both are superior to the overmatched schedules. In the presence of realistic memory latencies the differences between schedules is reduced further. However, there is little observed performance improvement overall as the amount of dynamic hardware increases even without penalizing these configurations with increased cycle times.

These results recommend using little to no dynamic analysis hardware in the microarchitecture (and, consequently, well matched schedules)—and adding in a cycle time penalty will only serve to strengthen this result. The dividing line between the hardware and the software is clear: as little dynamic analysis as necessary should be performed in hardware. Using dynamic analysis hardware to issue ready operations to function units simply does not provide a significant performance benefit in realistic systems. As always, the compiler is still required to do as much analysis as is required to produce a good schedule.

There may yet be a need to use dynamic analysis hardware to schedule operations—as long as there is a demand for high performance processors that maintain compatibility with a scalar processor architecture there will be a need to perform dynamic scheduling. However, because a simple microarchitecture is desirable, this niche may best be filled by dynamically scheduled, statically issued processors. The dynamic object code translation performed by both Intel and AMD [10, 3] are both related to this area.

6. Acknowledgments

We would like to thank the Hewlett-Packard Company for the loan of a workstation and Micro Magic, Inc. for the use of their compute farm. We would also like to thank Wen-Mei Hwu and the University of Illinois Urbana-Champaign for the use of the IMPACT compiler suite and John C. Gyllenhaal for his technical support of the simulation tools. Finally, we would like to thank Bob Rau for his assistance with this research project.

References

- [1] D. G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings 1967 Spring Joint Conference*, volume 30, pages 483–5, Washington, D.C., Apr. 1967. Thompson Books.
- [2] P. Bannon and J. Keller. Internal architecture of Alpha 21164 microprocessor. In *Digest of Papers: COMPCON '95*, pages 79–87, Los Alamitos, 1995. IEEE Computer Society, IEEE Computer Society Press.
- [3] D. Christie. Developing the AMD-K5 architecture. *IEEE Micro*, 16(2):16–26, Apr. 1996.
- [4] N. B. Gaddis, J. R. Butler, A. Kumar, and W. J. Queen. A 56-entry instruction-reorder buffer. In *1996 IEEE International Solid-State Circuits Conference. Digest of Technical Papers, ISSCC*, pages 212–3, New York, Feb. 1996. IEEE.
- [5] D. Hunt. Advanced performance features of the 64-bit PA-8000. In *Digest of Papers: COMPCON '95*, pages 123–8, Los Alamitos, 1995. IEEE Computer Society, IEEE Computer Society Press.
- [6] J. D. Johnson. *Expansion Caches for Superscalar Machines*. PhD thesis, Stanford University, Stanford, Mar. 1996.
- [7] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, 1991.
- [8] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, New York, May 1992. Association for Computing Machinery.
- [9] D. Levitan, T. Thomas, and P. Tu. The PowerPC 620™ microprocessor: A high performance superscalar RISC microprocessor. In *Digest of Papers: COMPCON '95*, pages 285–91, Los Alamitos, 1995. IEEE Computer Society, IEEE Computer Society Press.
- [10] D. B. Papworth. Tuning the Pentium Pro microarchitecture. *IEEE Micro*, 16(2):8–15, Apr. 1996.
- [11] N. Patkar, A. Katsuno, S. Li, T. Maruyama, S. Savkar, M. Simone, G. Shen, R. Swami, and D. Tovey. Microarchitecture of HaL's CPU. In *Digest of Papers: COMPCON '95*, pages 259–65, Los Alamitos, 1995. IEEE Computer Society, IEEE Computer Society Press.
- [12] B. R. Rau. VLIW: Not your father's Oldsmobile. Invited talk, The 25th Annual International Symposium on Microarchitecture, Dec. 1992.
- [13] B. R. Rau. Dynamically scheduled VLIW processors. In *The 26th Annual International Symposium on Microarchitecture*, pages 80–92, Dec. 1993.
- [14] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer. *Computer*, 22(1):12–35, Jan. 1989. A version of this article appeared in The 22nd Annual Hawaii International Conference on System Sciences.
- [15] K. W. Rudd. Instruction level parallel processors—a new architectural model for simulation and analysis. Technical Report CSL-TR-94-657, Stanford University, Dec. 1994.
- [16] G. Slaveburg, S. Rathnam, and H. Dijkstra. The TriMedia TM-1 PCI VLIW media processor. In *Hot Chips 8: A Symposium on High-Performance Chips, Symposium Record*, Aug. 1996.
- [17] M. Tremblay and J. M. O'Connor. UltraSparc I: A four-issue processor supporting multimedia. *IEEE Micro*, 16(2):42–9, Apr. 1996.
- [18] D. W. Wall. Limits of instruction-level parallelism. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 176–88, New York, Apr. 1991.
- [19] K. C. Yeager. The Mips R1000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.

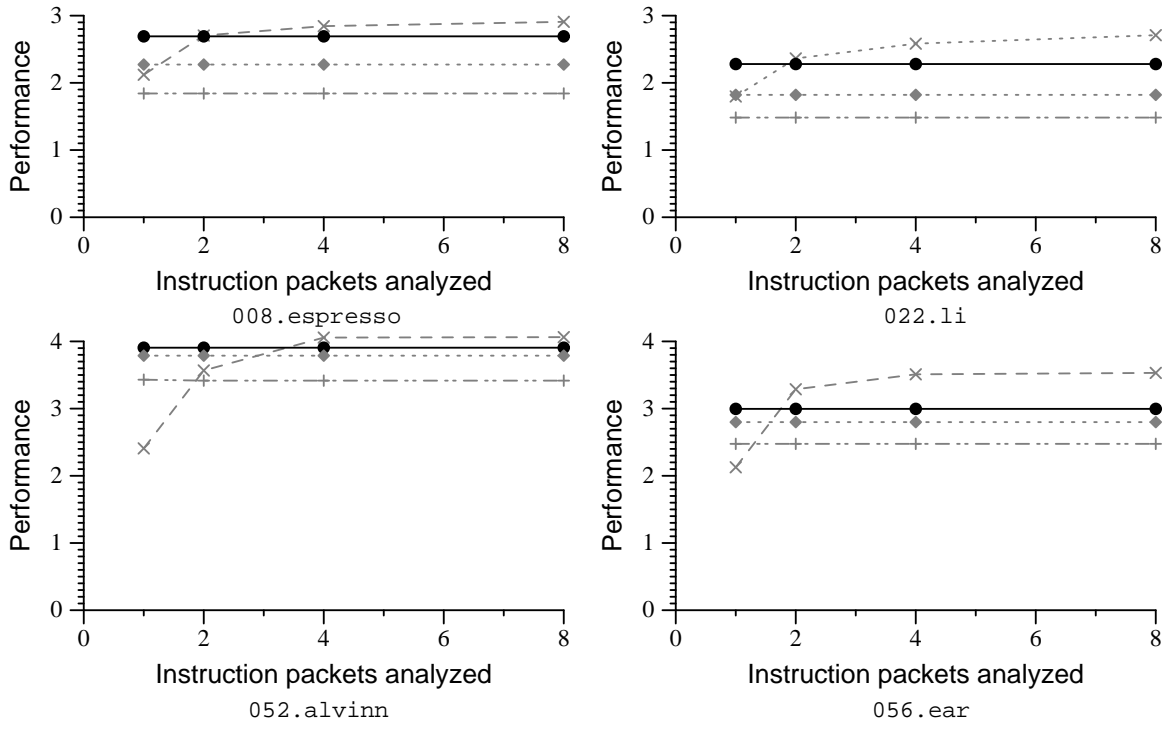


Figure 5. Achieved performance in operations per cycle for all benchmarks (perfect cache and branch prediction)

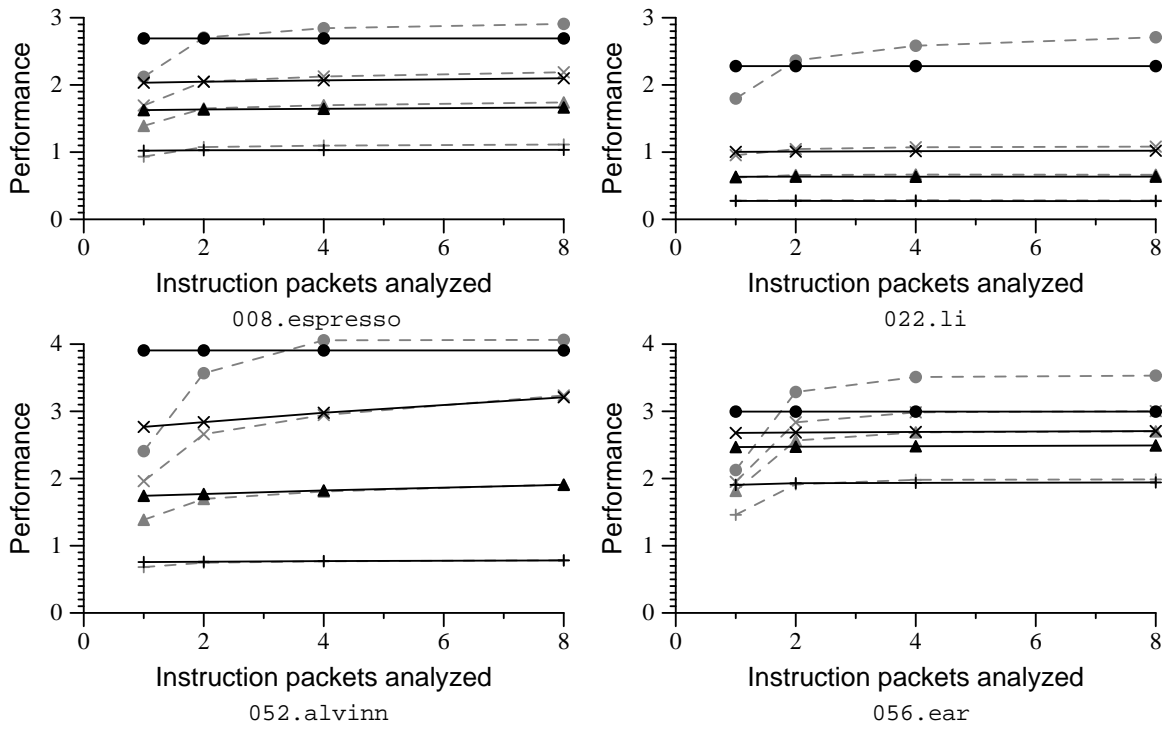


Figure 6. Achieved performance in operations per cycle for all benchmarks (varying memory latencies)