INSTRUCTION LEVEL PARALLEL PROCESSORS—A NEW ARCHITECTURAL MODEL FOR SIMULATION AND ANALYSIS

Kevin W. Rudd

Technical Report: CSL-TR-94-657

December 1994

This work was supported in part by NASA-Ames under grant NAG2-248.

INSTRUCTION LEVEL PARALLEL PROCESSORS—A NEW ARCHITECTURAL MODEL FOR SIMULATION AND ANALYSIS

by

Kevin W. Rudd

Technical Report: CSL-TR-94-657

December 1994

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305-4055

Abstract

Trends in high-performance computer architecture have led to the development of increased clock-rate and dynamic multiple-instruction issue processor designs. There have been problems combining both these techniques due to the pressure that the complex scheduling and issue logic puts on the cycle time. This problem has limited the performance of multiple-instruction issue architectures. The alternative approach of static multiple-operation issue avoids the clock-rate problem by allowing the hardware to concurrently issue only those operations that the compiler scheduled to be issued concurrently. Since there is no hardware support required to achieve multiple-operation issue (there are multiple operations in a single instruction and the hardware issues a single instruction at a time), these designs can be effectively scaled to high clock rates. However, these designs have the problem that the scheduling of operations into instructions is rigid and to increase the performance of the system the entire system must be scaled uniformly so that the static schedule is not compromised. This report describes an architectural model that allows a range of hybrid architectures to be studied.

Key Words and Phrases: super-scalar, superscalar, VLIW, instruction level parallelism, computer architecture, static scheduling, dynamic scheduling, processor simulation.

Contents

1 Introduction

High-performance processor design has recently taken two different (and often diametrically opposed) approaches. One approach used to design a high-performance processor is to increase the execution rate by increasing the clock rate of the processor or by reducing the latency of operations. These improvements are primarily through the application of circuit design and fabrication process enhancements and, while important, are outside the scope of this report. Another approach used to design a high-performance processor is to issue and to execute multiple operations concurrently. Traditional processor designs that issue and execute at most one operation per cycle have been referred to as "scalar" designs[1]. Processor designs that can issue and execute more than one operation per cycle will be referred to as "super-scalar" processors.

Historically, there have been two primary techniques used to achieve super-scalar performance. The first technique uses dynamic analysis of the instruction stream during execution to determine those operations that are independent. These independent operations can be issued and executed concurrently. The second technique uses static analysis performed by the compiler to schedule independent operations together into compound multi-operation instructions. All operations in a given instruction can be issued and executed concurrently with no dynamic analysis. Both of these approaches to processor design have advantages and disadvantages and both have demonstrated significant problems that have limited their performance. We have developed an architectural model that will be used as a research platform to evaluate a number of configuration and policy trade-offs in a design space that spans these two techniques and allows a variety of hybrid architectures to be compared.

Section 2 describes these two techniques in more detail and develops the need for exploring hybrid architectures to achieve scalable high-performance processor designs. Section 3 describes the execution paradigm that supports modeling a range of processors that spans these two techniques and provides several examples to clarify its organization and behavior. Section 4 describes the model in detail. This model is parametric not only in configuration (number and type of function units, architectural and implementation latencies) but also in several of the key policies that affect the behavior of the architecture. Section 5 describes several examples to reinforce the features and operation of the model.

2 Super-scalar issue techniques

As recent designs have demonstrated, neither dynamic nor static techniques alone have been sufficient to achieve high performance. The obvious consequence of this situation is that hybrid techniques should be considered as a general and effective solution to the problem of achieving high-performance in a super-scalar processor.

[1]Each operation in a "scalar" processor operates on single (scalar) data values in contrast to a "vector" processor which, while executing only a single operation at a time, execute the operation on multiple (vector) data values.

2.1 Dynamic super-scalar issue techniques—**dynamic multi-instruction issue** processors

The most popular technique today is the use of dynamic analysis to determine the operations that are available for concurrent issue and execution. Processors that use this technique are **dynamic multi-instruction issue** processors and are frequently referred to as **superscalar** processors—an unfortunate but not untrue nomenclature since this technique is but one of many techniques that achieve super-scalar performance. In this type of architecture, each instruction in the instruction stream consists of a single operation. In order to issue and to execute multiple operations concurrently, operations in the instruction stream must be analyzed for dependencies each time that issue is attempted. Compilers minimize the actual dependencies that an instruction stream will have but if the compiler cannot, or does not, prevent a dependency, the hardware must ensure that all dependencies are detected and enforced. This technique has significant advantages, one of which is code compatibility with other processors sharing the same architecture—a significant advantage in the marketplace if not in the research laboratory. Another advantage is that, since the analysis is based on the dynamic code stream, operations across branches and other changes of control flow are considered for concurrent issue and execution.

The major drawback of using dynamic analysis in a **dynamic multi-instruction issue** processor is that it requires a significant amount of hardware to analyze the instruction stream and to ensure that correctness is maintained. This approach lengthens the amount of time required to perform operation issue and can result in an increased cycle time (resulting in a reduced clock rate) or an increased pipeline length (resulting in increased branch penalties). Neither of these results is desirable and and the increase in performance from the concurrent issue and execution of multiple operations can be reduced or eliminated due to these problems. The complexity of the hardware required to perform the analysis is limited by both the number of operations within the analysis window (proportional to n^2 where n is the window size) and the maximum number of pending register results (proportional to k where k is the number of pending results). Existing designs have had difficulty with dynamically analyzing 2 to 4 operations and even with low issue widths have been limited to exploiting only those cases that match specific patterns of operations.

2.2 Static super-scalar issue techniques—**static multi-operation issue** processors

An alternative technique is the use of static analysis to determine the operations that are available for concurrent issue and execution. Processors that use this technique are **static multi-operation issue** processors and are frequently referred to **VLIW** processors—these processors have very long instruction words. In this type of architecture, each instruction in the instruction stream consists of multiple independent operations. No dynamic analysis is required to issue and execute these operations concurrently—the compiler performs any necessary analysis and schedules independent operations within a single instruction. Since no dynamic analysis is required during execution, the clock rate is not limited by the instruction issue logic and wide and fast processor designs are feasible. It is possible that there will be independent operations within several instructions—but, in order to issue and execute operations from multiple instructions concurrently, similar techniques to those used in dynamic multi-instruction issue processors would be required and similar complexity issues would arise.

Using static analysis in a static multi-operation issue processor is not without its problems. While there is no complex hardware to limit the performance, there is the requirement that the compiler have full knowledge of the hardware structure and latencies. This requirement is necessary since there are typically no interlocks on data dependencies and operations must not be scheduled until the source operands are ready. A side effect is that the hardware cannot vary from the specification that the compiler works with—all latencies (measured in cycles) must be constant. Thus, while the entire system can be improved if the speed-up is uniform, individual portions of the system cannot be improved upon independently without requiring that the program be recompiled in order to run on the new system. Another side effect is that since latencies are constant, operations with non-constant latencies cannot be supported. Load operations from a memory hierarchy that has multiple levels (and includes data caches) have multiple latencies depending on the location of the data within the memory hierarchy. This makes data caches unusable in a static multi-operation issue architecture and results in memory operations that have very long latencies. However, there are many programs that are able to be efficiently scheduled with long memory operation latencies. These programs are frequently loop-based floating point programs are able to be scheduled to hide the memory latency within loops and calculations. Existing implementations have been able to exploit instructions that are 28 operations wide [4] on this class of application. Unfortunately, there are also many programs that are not able to hide the long memory latency and these programs do extremely poorly on these processors.

2.3 Issues leading to a hybrid design

Clearly, both dynamic multi-instruction issue and the static multi-operation issue techniques can be used to achieve super-scalar performance and neither one has been proven to be superior to the other in all cases. However, these techniques are not mutually exclusive and some combination of static and dynamic techniques may result in an architecture that tolerates multiple latency operations and is efficient at wide instruction widths. In order to understand the aspects of these techniques that must be considered in a hybrid architecture, several issues must first be discussed.

One fundamental distinction that must be made is the separation of *architectural* and *implementation* specifications. An architectural specification determines the operation semantics that the compiler must schedule code for to ensure a correct schedule and defines the *virtual processor*. An implementation specification determines the operation semantics that are actually implemented—but this must be transparent outside the processor so that code that is scheduled to the architectural specification will run. For example, an add operation may have an architectural latency of 2˜ while a given implementation may have a latency of 3˜—this is not a problem as long as the implementation ensures that the instruction stream will see the result as if it had occurred in 2˜ as scheduled.

The notion of architectural and implementation specifications lead directly to the notion of *virtual time* and *latency* and *real time* and *latency*. Operations are scheduled according to the architectural specification in virtual time and appear to have virtual latency while they execute on the hardware according to the implementation specification in real time and have real latency. The implementation of the processor provides the illusion of a virtual processor that implements the architectural specification in much the same way that memory management hardware and operating system software provide the illusion of virtual memory—in both cases the reality is potentially very different from the illusion and yet there is no awareness of reality to the program.

In order to maintain the illusion of the virtual machine, when a result is generated earlier in real time than the virtual latency specifies, the result must be delayed so that it does not become visible to the instruction stream until the appropriate virtual time. Analogously, when a result is generated later in real time than the virtual latency specifies, any operations that use the result (as scheduled in virtual time) are delayed appropriately until the result is available. This distinction has been implicitly true within dynamic multi-instruction issue architectures (where operation reordering avoids data dependencies when possible and interlocks prevent data dependencies from occurring); there is no distinction between architecture and implementation in static multi-operation issue architectures since no operation variation is allowed[2]. The distinction between the architecture and the specification as it applies to dynamic VLIW architectures was noted in [5].

Most operations in a dynamic multi-instruction issue architecture have a virtual latency of $1^{\sim}$—that is, the hardware guarantees that dependent operations that are scheduled consecutively (and which, in the virtual machine, issue in consecutive virtual cycles) will produce the desired result. The few exceptions are typically memory load and control flow operations which often have a virtual latency of $2^{\sim}$ and are specified with a "load delay slot" or a "branch delay slot" that may or may not be able to be filled with a useful instruction by the compiler. For the case of a load that takes longer, interlocks typically ensure that the dependencies between operations are honored; for the case of a branch that takes longer, the pipeline control typically inserts null operations into the pipeline until the new instruction stream is available. A recent trend in architectural extensions is to deprecate or eliminate these "delayed" operations and to add in analogous operations that have a virtual latency of $1^{\sim}$ so that all operations have uniform virtual latencies. Shifting away from delayed operations allows implementations to more easily support varying issue widths and out-of-order execution. Unfortunately, this does not eliminate the need for the compiler to target its code schedule for a given architecture with knowledge of its implementation specification—in fact, this customized scheduling becomes more and more important as higher performance and wider issue designs are developed.

Even a simple machine that performs no concurrent operation issue is affected by poor code scheduling. A program that is scheduled using only the architectural latencies (of $1^{\sim}$ per operation) will run correctly but will exhibit poor performance on many implementations. With this code schedule, any time that there is a dependency (when the implementation latency exceeds the architectural latency) the pipeline will have to stall until the

[2]There have been cases where some results, primarily those from memory operations, may not be available when scheduled. These cases were handled by stalling the processor until the late result was available.

dependency is resolved. One of the goals of dynamic multi-instruction issue architectures is to analyze the instruction stream and to rearrange the operations so that better utilization of the pipeline is possible (this was a precursor to the goal of achieving concurrent issue and execution and is a similar but more limited problem). Thus, when a dependency arises, the dynamic scheduling will attempt to find another operation that can execute while the stalled operation waits for the dependent result to become available. But even if a program is scheduled efficiently for one implementation, when run on a different implementation it may no longer be scheduled efficiently and performance may suffer.

When concurrent issue and execution is considered, the problem becomes worse. The goal is no longer to find one available operation to issue but to find several available operations to issue. In this situation, there are more operands being analyzed and and more operations in progress within the pipelines to be checked against resulting in significantly more hardware requirements. This occurs even when the compiler anticipates which operations are independent and schedules them so as to avoid any dependencies on a given implementation—however, since this information is thrown away after the schedule is generated there is no way that the implementation can be aware of this. Additionally, operations that are analyzed but not issued are examined again during the next analysis cycle resulting in more redundant computation by the hardware.

To clarify this problem, consider the following example code sequence:

```
0:      r <- comp8(a, b)
1:      s <- comp2(c, d)
2:      t <- comp4(d, e)
3:      u <- comp2(s, f)  ; s available
4:      v <- comp4(s, s)
5:      w <- comp2(u, a)  ; u available
6:      x <- comp2(t, u)  ; t available
7:      y <- comp4(b, w)  ; w available
8:      z <- comp2(r, x)  ; r, v, x available
```

In this code sequence, `comp2` operations complete in 2˜, `comp4` operations complete in 4˜, and `comp8` operations complete in 8˜.

First, consider the case where the architectural latency is 1˜ for all operations. Now every operation scheduled completes "late" and all operations that are in any stage of execution must have their result operands compared with all operations being evaluated for issue to ensure that there are no dependencies that must be enforced. All operations that are being analyzed for issue until cycle 8 must be checked to ensure that they do not use result `r` (which is unavailable for cycles 1 through 7) even though the compiler scheduled the code with this knowledge and was able to avoid any true dependency. This is true of every operation that is in execution since all of these are "late" relative to the architectural specification.

Next, consider the same example where the architectural specification reflects the anticipated implementation specification. This means that values are known not to be available until their architectural (and implementation) latency and operations that are not "late" do not need to be checked while analyzing the instruction stream. Now, when every operation completes "on time" there are no checks required. In contrast, consider the same example where now all `comp2` operations actually take 3˜ to complete. In this case, the operation in instruction 1 does not complete in cycle 3 but in cycle 4. Since result `s` is not late until cycle 4 this cycle is the first cycle that this result needs to be checked while analyzing the instruction stream. Even if there are a number of operations that have an implementation latency that is greater than the architectural latency, it is clear from these cases that using non-unit architectural latencies that are matched closely to the anticipated implementation latencies results in few late operations. Since there are few late operations there are also few pending register results that need to be checked for dependencies during the issue process are reduced resulting in less complex issue hardware.

2.4 Hybrid architecture conclusions

As has been shown, the significant feature that a dynamic multi-instruction issue processor has is its ability to reschedule code to support unexpected latencies—either from a poor schedule or from a mispredicted operation latency (typically from the memory system). This is, unfortunately, an expensive feature to support given the liabilities of single-operation instructions and the 1˜ virtual operation latencies. There are two significant features that the static multi-operation issue processors have—multiple operation instructions (which are guaranteed independent) and accurate virtual operation latencies. These features have their downsides however. The exclusive use of static scheduling proscribes any scheduling errors—including mispredicted operation latencies—requiring that all implementation latencies match the virtual latencies.

All is not as bleak as one might expect—there are three key characteristics of these two processors that can be identified and must be considered in a hybrid architecture: the instruction stream must include as much dependency information as possible from the compiler; architectural latencies that correspond reasonably well to the anticipated implementation latencies must be used; and the distinction between architectural and implementation specifications and the illusion of a virtual machine. Providing these capabilities reduces the amount of dynamic analysis that must be performed to issue and to execute concurrent operations (the critical problem with dynamic multi-instruction issue processors) and reduces or eliminates many scheduled stalls (the critical problem with static multi-operation issue processors). This results in fast wide operation issue architectures that will achieve much higher performance and much greater scalability than either of the two original architectures.

3 The split-issue execution paradigm

The primary goal of creating a new architectural model is to explore high-performance instruction-level parallel processor architectures over a wide range of configurations. In order to meet this requirement, section 2.4 presented three architectural characteristics that are effective in achieving this goal. To include dependency information from the compiler we use statically scheduled instructions that contain independent operations; to minimize the differences between the architectural and implementation machines we use realistic latencies for all operations; and to maintain the illusion of a virtual processor we use split-issue techniques.

The first two features, instructions containing independent operations and the use of realistic latencies, are already characteristic of static multi-operation issue processors. This similarity is not surprising since both of these are efficient techniques for embedding dependency information into the instruction stream. Both the Horizon [1] and Cydra 5 [2] processors used variations on the use of realistic latencies for their operations to avoid unnecessary analysis or stalls.

In the Horizon processor, instead of using fixed operation latencies, each instruction embedded the distance to the nearest use of any result from the contained operations along any possible path. The use of this distance field allowed additional instructions to be issued while a given instruction was in execution—only when the distance value was exceeded and the instruction was still in execution would a stall occur. Since the Horizon instruction contained three operations, this distance value was a worst-case value of all three operations along all possible paths. One obvious extension to this technique would be to use a unique distance value for each operation. This extension would increase the Horizon instruction size by 5% to 10% and would increase the complexity of the execution logic slightly to support carrying a distance value for each operation.

In the Cydra 5 processor, instead of using fixed memory latencies a memory latency register contained the effective memory latency value that the current code region was scheduled for. Using the memory latency register allowed the schedule to take into account the memory access pattern of a region of code and to reduce the scheduled memory latency in those cases where the memory system could approach its minimum latency. When memory references took a different amount of time to complete than the value in the memory latency register, the results were either buffered (when available early) or the pipelines were stalled (when available late) so that results appeared to be available precisely as schedule. This approach could be easily be extended to support an operation latency register for each operation or class of operations. This would not affect the instruction size (although there would need to be new instructions added to read and to modify these values) but would increase the complexity of the execution logic to support variable scheduled latencies for all operations.

Neither of these extensions appears to have significant performance gains although both may reduce the stall penalty in some cases; both of these extensions increase the complexity of the processor to support operations that have non-fixed latencies during execution. Using statically scheduled instructions with fixed virtual latencies avoids both of these inefficiencies and embeds a significant amount of dependency information in each instruction. There may yet be better approaches and when an efficient approach for embedding the full dependency graph in the instruction stream is developed then the situation may improve further—but this development is neither likely nor expected to achieve dramatic improvements over the current approach.

The third feature, the use of split-issue techniques, is less traditional and the remainder of this section is devoted to the topic. In the split-issue execution paradigm the execution of an operation is broken up into three independent sets of events that correspond to the three phases of operation execution—the acquisition of source values from their origination (register file), the computation of result values (within execution resources), and the delivery of result values to their destination (register file). Each event in this scheme is scheduled at the virtual time that the architectural specification requires so that the implementation is able to maintain the illusion of the virtual processor. Temporary storage is used to hold the source and result values during this process which decouples the three phases from each other and allows them to be acted upon independently. There are a number of dependency issues that must be considered in order to ensure correctness but these are policy issues that are independent of (but allowed by) the model. This section will describe the split-issue paradigm in detail and will show how this model is directly applicable to the simulation of a wide range of high performance processors.

3.1 An introduction to split-issue

The execution of an operation typically consists of three distinct phases—the acquisition of source values, the computation of result values, and the delivery of the completed results. In a traditional pipelined processor, these are often described as "register fetch", "execute", and "write-back" and are considered to be inseparable steps in the execution of an operation in the pipeline. In the split-issue model, these phases are decoupled from each other so that they can be treated independently in the implementation. In order to maintain the correctness of the virtual machine, each event is scheduled to take effect at the appropriate time based on the architectural specification for each operation. Whereas in the traditional pipeline, source, result, and intermediate values are held in latches in pipeline stages until conditions are set to proceed, in the split-issue model these are held in temporary storage locations.

The original concept for split-issue was presented in [3] as a mechanism for supporting a dynamic execution model for a static multi-operation issue processor. In the original model, operations were split into two independent micro-operations that operated in a producer-consumer relationship using an internal temporary register file to buffer results between the two micro-operations. Each micro-operation is scheduled to occur in the virtual cycle specified by the architecture. Thus the operation $A \leftarrow f(B, C)$ with an architectural latency of $3^{\sim}$ would be broken out into the two micro-operations $T_1 \leftarrow f(B, C)$ @ 0 and $A \leftarrow T_1$ @ 2 with the advantage that slip between these two micro-operations is allowed. In this nomenclature, $x \leftarrow f(y)$ @ z indicates that the register (temporary or otherwise) x is assigned the value of the computation $f(y)$ at the virtual time z. If there is no time specified then the result will be generated at some arbitrary time when it is ready. Note that the result of this example operation is available at the end of cycle 2—and can be used $3^{\sim}$ from the start time as required by the architectural specification.

The original model supports the ability to dynamically schedule operations but limits these operations to those that can be supported in this simple two-step manner. However, the coupling of all source accesses with the operation may lead to unnecessary complexity when some source values are available and others are not. Reconsidering the events that occur during the execution of an operation leads to a revision of the two-step producer-consumer model to the three-step acquisition-computation-delivery model that provides additional capabilities.

In the split-issue model, operations are split into into three independent phases—ϕ^1 performs all acquisition events, ϕ^e performs all computation, and ϕ^2 performs all delivery events. The ϕ^1 and ϕ^2 events are similar in that they are access events performing read and write accesses to the register file. Using the same example as before, in this model $A \leftarrow f(B, C)$ results in the event triple

$$\begin{array}{lll} \phi^1: & T_1 \leftarrow B \ @ \ 0 & T_2 \leftarrow C \ @ \ 0 \\ \phi^e: & T_3 \leftarrow f(T_1, T_2) & \\ \phi^2: & A \leftarrow T_3 \ @ \ 2 & \end{array}$$

Note that the ϕ^e events do not need any timing information specified since they are able to commence once all operands are flagged as available.

With the exception of the separation of the acquisition events this approach accomplishes the same result as the original approach with the apparent addition of complexity. However, there are two benefits of this approach. First, there is only a single conceptual micro-operation that operates on the temporary storage values instead of two independent processing units—the acquisition and delivery of source and result values has been decoupled and is in many ways more like the traditional pipelined approach. Second, in addition to supporting basic operations like those supported in the original model, operations that allow events to occur at arbitrary cycles are also supported. One simple example is that of an operation with the late delivery of one operand—such as in a multiply-accumulate operation that forms the core of the SAXPY loop. This operation, $S \leftarrow A * X + Y$, can be specified such that only the two multiplicands (A and X) are required to start the operation and the summand (Y) is not required until later[3]. Using a $1^{\sim}$ delay for the summand and a $3^{\sim}$ operation latency, we have the triple

$$\begin{array}{llll} \phi^1: & T_1 \leftarrow A \ @ \ 0 & T_2 \leftarrow X \ @ \ 0 & T_3 \leftarrow Y \ @ \ 1 \\ \phi^e: & T_4 \leftarrow f(T_1, T_2) & T_5 \leftarrow g(T_4, T_3) & \\ \phi^2: & A \leftarrow T_5 \ @ \ 2 & & \end{array}$$

Note that in this example, the function $f(a, x)$ is used as the first stage of processing of the multiply-accumulate that takes the two multiplicands as arguments and immediately sets the flag T_4 (which could be an intermediate value from the computation) to represent that the computation has begun. The function $g(t, z)$ is then used to provide the summand to the process and the final result is placed in the temporary T_5.

Arbitrarily complex operations may be constructed in this manner and even the most complex CISC processor operation can be framed in this form. For example, even indefinite iteration can be specified as in $T_x, \cdots \leftarrow h(T_x, \cdots)$. One tangential use of split-issue techniques might be the emulation of CISC processors—this use will not be considered further in this report.

An observation of the representation used in the above two examples is that it bears a striking resemblance to the information contained in a machine specification for the operations. In a machine readable form, this is essential information that a compiler would

[3]Note that the multiply-accumulate operation is a reasonable example of the use of this approach and is *not* necessarily a recommendation for this specification!

need to schedule the operation and is also essential information that a processor designer would require to implement the same operation. This natural connection between specification and model is fortuitous—and the next section will demonstrate that there is a similar connection between the model and implementation as well.

One final point to make about the model concerns the timing of the events. It is clearly important to be able to model operations that take a single cycle to execute and to allow these to be supported back-to-back. Rather than to specify complex rules regarding the behavior of forwarding logic to deliver results and to provide values to execution events issuing the same cycle, this model adopts a simple read-execute-write sequence within a cycle. This is not a significant limitation since the problem can be resolved for many processor configurations by having the compiler schedule operations taking these delays into account and using operations that are specified to have the appropriate latencies for these cases. It is easy to conceive of a complex processor for which this approach would be inadequate but it is our belief that these processors are also too complex to reasonably consider building.

3.2 A tabular representation for **split-issue**

While the representation presented in the previous section is useful for specifying the behavior of operations, it is not directly useful for an implementation—either hardware or simulator—due to the tagged nature of each of the acquisition and delivery events. However, there is an analogous table-based representation that embodies the same information in a similar manner to what can be seen to be efficiently implemented. In the tabular representation there are three columns that represent the three phases in the execution of the operation—ϕ^1, ϕ^e, and ϕ^2. Thus the earlier example of the simple operation $A \leftarrow f(B,C)$ is represented in this model as

ϕ^1
$T_1 \leftarrow B$ $T_2 \leftarrow C$

	ϕ^e	
src	*op*	*dest*
T_1 T_2	f	T_3

ϕ^2
$A \leftarrow T_3$

In this representation, the first column corresponds to the ϕ^1 events and the third column corresponds to the ϕ^2 events. These operate as queues of events scheduled in virtual time with the bottom event being the current event. Thus, for this example, the acquisition events $T_1 \leftarrow B$ and $T_2 \leftarrow C$ are scheduled as the next events to be processed and the delivery event $A \leftarrow T_3$ is scheduled to occur two cycles later as specified. As events are processed, all events in the queues progress one cycle just as in a queue. The behavior is only half like a queue since events are inserted at their appropriate virtual time and not at the very top, but the queue analogy is a useful one in understanding the sequencing of event processing.

The second column is corresponds to the ϕ^e events and is used more as a scoreboard of pending operations than as a queue. This example reflects that the source values for the operation shown are in temporary storage T_1 and T_2 and that the result value will be placed in temporary storage T_3. Note that, although there is a required correlation between the positions in the ϕ^1 and ϕ^2 queues, there is no correlation between the positions in the ϕ^1 and ϕ^2 queues and the entries in the ϕ^e structure. While the ϕ^e structure may maintain the ordering of operations[4] there is no requirement that it do so.

In the examples that follow, all operations will be limited to a single source and destination operand and will avoid the use of temporary storage labels. This simplifies the examples and demonstrates the use of the model without any loss of generality—but with significantly less complexity than actually tracking multiple source and destination values using more realistic operations. Another simplification is that the tabular representation will be "preloaded" at the start of the example and will not show the sequential filling that would normally occur during a step-by-step execution process. However, all events are listed in the places that they would be relative to the first operation that is inserted so the events that are processed will be dealt with appropriately.

The following demonstrates the use of this representation for hypothetical code starting at cycle 1000:

Cycle 1000

ϕ^1
$g : \mathrm{r}(\mathcal{R}_1)$

	ϕ^e	
src	*op*	*dest*
$\surd$	(f)	—
—	g	
	h	—

ϕ^2
$h : \mathrm{w}(\mathcal{R}_1)$

$f : \mathrm{w}(\mathcal{R}_2)$

Register File	
$\mathcal{R}_1 =$	3
$\mathcal{R}_2 =$	?

This example represents three arbitrary operations labeled "f", "g", and "h". The parenthesis on operation f represent that the operation has started computation. In the state represented, the acquisition for operation f is completed (represented by the "$\surd$" mark) and the operation has started computation; however, the "—" in the destination for f indicates that there is a result value but it is not available yet. The delivery event for operation f is shown below the ϕ^2 event queue as "$f : w(\mathcal{R}_2)$"—this placement indicates that this event was scheduled to be executed before the operation actually completed computation and is "late"—this also is reflected in the "?" in register $\mathcal{R}_2$ indicating an unknown value.

Operation g is waiting for its source value (this is the next acquisition event to execute) and has no result and operation h has no source value but generates a result. Neither of these operations has started execution yet although it may be possible to start operation h depending on how dependencies are handled (operation h writes the same register that operation g is waiting to read). One key feature of this model is that a number of aspects of an implementation—such as how dependencies and late results are handled—are not a part of the model but can be specified independently.

[4] One use of operation ordering in the ϕ^e structure is to implement precise interrupts.

3.3 Representation of a traditional static multi-operation issue processor

As a real example to start off with, consider a bare-bones static multi-operation issue with no dynamic behavior of any kind—typical of a traditional VLIW processor (although for the purposes of this example, all operation latencies are 1˜ for simplicity). For this example, we will use the following code sequence:

$$\begin{aligned}\mathcal{R}_2 &\leftarrow f(\mathcal{R}_1)\\ \mathcal{R}_3 &\leftarrow g(\mathcal{R}_2)\\ \mathcal{R}_2 &\leftarrow h(\mathcal{R}_3)\end{aligned}$$

There are two dependencies in this code (each operation is immediately dependent on its predecessor). However, since the compiler guarantees that there are no scheduling problems the processor can execute the code stream without regard to any potential dependencies. These operations generate results that correspond to the operation. The first operation thus generates the result f and so forth. Initial values correspond to the register number for simplicity.

Thus, we initially have

Cycle 0

ϕ^1
h : r($\mathcal{R}_3$)
g : r($\mathcal{R}_2$)
f : r($\mathcal{R}_1$)

ϕ^e		
src	*op*	*dest*
—	f	—
—	g	—
—	h	—

ϕ^2
h : w($\mathcal{R}_2$)
g : w($\mathcal{R}_3$)
f : w($\mathcal{R}_2$)

Register File	
$\mathcal{R}_1$=	1
$\mathcal{R}_2$=	2
$\mathcal{R}_3$=	3

After executing the first cycle we read $\mathcal{R}_1$, issue operation 1, and write $\mathcal{R}_2$ and this results in

Cycle 1

ϕ^1
h : r($\mathcal{R}_3$)
g : r($\mathcal{R}_2$)

ϕ^e		
src	*op*	*dest*
$\surd$	(f)	$\surd$
—	g	—
—	h	—

ϕ^2
h : w($\mathcal{R}_2$)
g : w($\mathcal{R}_3$)

Register File	
$\mathcal{R}_1$=	1
$\mathcal{R}_2$=	f
$\mathcal{R}_3$=	3

Similarly we execute the second cycle and get

Cycle 2

ϕ^1
h : r($\mathcal{R}_3$)

ϕ^e		
src	*op*	*dest*
$\surd$	(f)	$\surd$
$\surd$	(g)	$\surd$
—	h	—

ϕ^2
h : w($\mathcal{R}_2$)

Register File	
$\mathcal{R}_1$=	1
$\mathcal{R}_2$=	f
$\mathcal{R}_3$=	g

And finally we execute the third cycle and get

Cycle 3

ϕ^1

ϕ^e		
src	*op*	*dest*
$\surd$	(f)	$\surd$
$\surd$	(g)	$\surd$
$\surd$	(h)	$\surd$

ϕ^2

Register File	
$\mathcal{R}_1$=	1
$\mathcal{R}_2$=	h
$\mathcal{R}_3$=	f

The code operates in 3˜ as scheduled—not surprising since traditional static multi-operation issue architectures do not allow any unpredictability in the behavior of their operations.

3.4 Further examples of split-issue techniques

We now consider a configuration that relies on split-issue techniques to maintain correctness. As a first example of the abilities of split-issue to resolve problems with schedule difficulties, consider the same code sequence as before with the exception that each operation now has an architectural latency of 2˜ while the code is scheduled for the virtual latency of 1˜.

We have the same initial condition of the example in the previous section of

Cycle 0

ϕ^1
$h : \mathrm{r}(\mathcal{R}_3)$
$g : \mathrm{r}(\mathcal{R}_2)$
$f : \mathrm{r}(\mathcal{R}_1)$

	ϕ^e	
src	*op*	*dest*
—	f	—
—	g	—
—	h	—

ϕ^2
$h : \mathrm{w}(\mathcal{R}_2)$
$g : \mathrm{w}(\mathcal{R}_3)$
$f : \mathrm{w}(\mathcal{R}_2)$

Register File	
$\mathcal{R}_1=$	1
$\mathcal{R}_2=$	2
$\mathcal{R}_3=$	3

After the first cycle we've issued the first computation but it hasn't completed yet. We thus have the following situation where the first write must be held until it is ready. We thus have

Cycle 1

ϕ^1
$h : \mathrm{r}(\mathcal{R}_3)$
$g : \mathrm{r}(\mathcal{R}_2)$

	ϕ^e	
src	*op*	*dest*
$\surd$	(f)	—
—	g	—
—	h	—

ϕ^2
$h : \mathrm{w}(\mathcal{R}_2)$
$g : \mathrm{w}(\mathcal{R}_3)$

$f : \mathrm{w}(\mathcal{R}_2)$

Register File	
$\mathcal{R}_1=$	1
$\mathcal{R}_2=$	?
$\mathcal{R}_3=$	3

After the next cycle we retire the first computation but can't start the second since the read hasn't occurred yet. We get

Cycle 2

ϕ^1
$h : \mathrm{r}(\mathcal{R}_3)$

$g : \mathrm{r}(\mathcal{R}_2)$

	ϕ^e	
src	*op*	*dest*
$\surd$	(f)	$\surd$
—	g	—
—	h	—

ϕ^2
$h : \mathrm{w}(\mathcal{R}_2)$

$g : \mathrm{w}(\mathcal{R}_3)$

Register File	
$\mathcal{R}_1=$	1
$\mathcal{R}_2=$	f
$\mathcal{R}_3=$	?

Now we can start the second but have to deal with the deferred write and the new deferred read and write, thus

Cycle 3

ϕ^1

$h : \mathrm{r}(\mathcal{R}_3)$

	ϕ^e	
src	*op*	*dest*
$\surd$	(f)	$\surd$
$\surd$	(g)	—
—	h	—

ϕ^2

$h : \mathrm{w}(\mathcal{R}_2)$

$g : \mathrm{w}(\mathcal{R}_3)$

Register File	
$\mathcal{R}_1=$	1
$\mathcal{R}_2=$	?
$\mathcal{R}_3=$	?

With the end if the first cycle of the second operation we have the result posted but the third operation has not yet read its register so does not immediately reflect the result. Thus,

Cycle 4

ϕ^1

$h : \mathrm{r}(\mathcal{R}_3)$

	ϕ^e	
src	*op*	*dest*
$\surd$	(f)	$\surd$
$\surd$	(g)	$\surd$
—	h	

ϕ^2

$h : \mathrm{w}(\mathcal{R}_2)$

Register File	
$\mathcal{R}_1=$	1
$\mathcal{R}_2=$	?
$\mathcal{R}_3=$	g

The third operation completes its read and issues

Cycle 5

ϕ^1

	ϕ^e	
src	*op*	*dest*
$\surd$	(f)	$\surd$
$\surd$	(g)	$\surd$
$\surd$	(h)	—

ϕ^2

$h : \mathrm{w}(\mathcal{R}_2)$

Register File	
$\mathcal{R}_1=$	1
$\mathcal{R}_2=$	?
$\mathcal{R}_3=$	g

And after its second cycle the third operation completes and updates its result thusly

Cycle 6

ϕ^1

	ϕ^e	
src	*op*	*dest*
$\surd$	(f)	$\surd$
$\surd$	(g)	$\surd$
$\surd$	(h)	$\surd$

ϕ^2

Register File	
$\mathcal{R}_1=$	1
$\mathcal{R}_2=$	h
$\mathcal{R}_3=$	g

This has taken a total of 6 cycles—exactly what would have been scheduled by the compiler had it know that the latency for the operations were doubled. This is also the same amount of time that it would have taken a double-cycled static multi-operation issue processor although operations would not have been executed with the same timing.

The next example will demonstrate a more realistic situation where split-issue is useful—a mis-scheduled memory load operation that takes 5˜ to complete instead of the scheduled 2˜ . This is representative of a first level cache miss which is a hit in the second level cache. Other operations will be included and will serve to mitigate the cost of the load. All other operations are scheduled to complete in 1˜ . The code sequence to be considered is as follows:

$$\begin{array}{rcl} \mathcal{R}_3 & \leftarrow & ld(\mathcal{R}_1) \\ & \leftarrow & f(\mathcal{R}_2) \\ \mathcal{R}_5 & \leftarrow & use(\mathcal{R}_3) \\ \mathcal{R}_6 & \leftarrow & g(\mathcal{R}_1) \\ \mathcal{R}_7 & \leftarrow & h(\mathcal{R}_5) \\ \mathcal{R}_8 & \leftarrow & k(\mathcal{R}_6) \end{array}$$

This sequence takes 6˜ to complete as scheduled.

For this example we have the initial setup of

Cycle 0

ϕ^1
k : r($\mathcal{R}_6$)
h : r($\mathcal{R}_5$)
g : r($\mathcal{R}_1$)
use : r($\mathcal{R}_3$)
f : r($\mathcal{R}_2$)
ld : r($\mathcal{R}_1$)

	ϕ^e	
src	*op*	*dest*
—	*ld*	—
—	f	
—	*use*	—
—	g	—
—	h	—
—	k	—

ϕ^2
k : w($\mathcal{R}_8$)
h : w($\mathcal{R}_7$)
g : w($\mathcal{R}_6$)
use : w($\mathcal{R}_5$)
ld : w($\mathcal{R}_3$)

Register File	
$\mathcal{R}_1$=	1
$\mathcal{R}_2$=	2
$\mathcal{R}_3$=	3
$\mathcal{R}_4$=	4
$\mathcal{R}_5$=	5
$\mathcal{R}_6$=	6
$\mathcal{R}_7$=	7
$\mathcal{R}_8$=	8

After the first operation, the load operation, we have the following state:

Cycle 1

ϕ^1
k : r($\mathcal{R}_6$)
h : r($\mathcal{R}_5$)
g : r($\mathcal{R}_1$)
use : r($\mathcal{R}_3$)
f : r($\mathcal{R}_2$)

	ϕ^e	
src	*op*	*dest*
$\surd$	(*ld*)	—
	f	
—	*use*	—
—	g	—
—	h	—
—	k	—

ϕ^2
k : w($\mathcal{R}_8$)
h : w($\mathcal{R}_7$)
g : w($\mathcal{R}_6$)
use : w($\mathcal{R}_5$)
ld : w($\mathcal{R}_3$)

Register File	
$\mathcal{R}_1$=	1
$\mathcal{R}_2$=	2
$\mathcal{R}_3$=	3
$\mathcal{R}_4$=	4
$\mathcal{R}_5$=	5
$\mathcal{R}_6$=	6
$\mathcal{R}_7$=	7
$\mathcal{R}_8$=	8

So far, so good—the load was supposed to take 2˜ to begin with so we can easily deal with the next operation. Thus

Cycle 2

ϕ^1
k : r($\mathcal{R}_6$)
h : r($\mathcal{R}_5$)
g : r($\mathcal{R}_1$)
use : r($\mathcal{R}_3$)

	ϕ^e	
src	*op*	*dest*
$\surd$	(*ld*)	—
$\surd$	(f)	
—	*use*	—
—	g	—
—	h	—
—	k	—

ϕ^2
k : w($\mathcal{R}_8$)
h : w($\mathcal{R}_7$)
g : w($\mathcal{R}_6$)
use : w($\mathcal{R}_5$)
ld : w($\mathcal{R}_3$)

Register File	
$\mathcal{R}_1$=	1
$\mathcal{R}_2$=	2
$\mathcal{R}_3$=	?
$\mathcal{R}_4$=	4
$\mathcal{R}_5$=	5
$\mathcal{R}_6$=	6
$\mathcal{R}_7$=	7
$\mathcal{R}_8$=	8

Now we see the problem—the load operation has not completed but the next operation requires the result. Thus this operation is pended as follows and processing continues, giving

Cycle 3

ϕ^1
k : r($\mathcal{R}_6$)
h : r($\mathcal{R}_5$)
g : r($\mathcal{R}_1$)

use : r($\mathcal{R}_3$)

	ϕ^e	
src	*op*	*dest*
$\surd$	(ld)	—
$\surd$	(f)	
—	*use*	—
—	g	—
—	h	—
—	k	—

ϕ^2
k : w($\mathcal{R}_8$)
h : w($\mathcal{R}_7$)
g : w($\mathcal{R}_6$)

use : w($\mathcal{R}_5$)
ld : w($\mathcal{R}_3$)

Register File	
$\mathcal{R}_1$=	1
$\mathcal{R}_2$=	2
$\mathcal{R}_3$=	?
$\mathcal{R}_4$=	4
$\mathcal{R}_5$=	?
$\mathcal{R}_6$=	6
$\mathcal{R}_7$=	7
$\mathcal{R}_8$=	8

While the last operation was pended, the next can proceed since it is not dependent on the load but on the operation following the load. Thus we have this operation complete "out of order" with respect to the earlier operations. We then see

Cycle 4

ϕ^1
k : r($\mathcal{R}_6$)
h : r($\mathcal{R}_5$)

use : r($\mathcal{R}_3$)

	ϕ^e	
src	*op*	*dest*
$\surd$	(ld)	—
$\surd$	(f)	
—	(*use*)	—
$\surd$	(g)	$\surd$
—	h	—
—	k	—

ϕ^2
k : w($\mathcal{R}_8$)
h : w($\mathcal{R}_7$)

use : w($\mathcal{R}_5$)
ld : w($\mathcal{R}_3$)

Register File	
$\mathcal{R}_1$=	1
$\mathcal{R}_2$=	2
$\mathcal{R}_3$=	?
$\mathcal{R}_4$=	4
$\mathcal{R}_5$=	?
$\mathcal{R}_6$=	g
$\mathcal{R}_7$=	7
$\mathcal{R}_8$=	8

The next operation depends on the operation stalled by the load operation and we must pend it too. However, the load completes in this cycle so it and its dependent operation are updated giving

Cycle 5

ϕ^1
k : r($\mathcal{R}_6$)

h : r($\mathcal{R}_5$)

	ϕ^e	
src	*op*	*dest*
$\surd$	(ld)	$\surd$
$\surd$	(f)	
$\surd$	(*use*)	$\surd$
$\surd$	(g)	$\surd$
—	h	—
	k	—

ϕ^2
k : w($\mathcal{R}_8$)

h : w($\mathcal{R}_7$)
use : w($\mathcal{R}_5$)

Register File	
$\mathcal{R}_1$=	1
$\mathcal{R}_2$=	2
$\mathcal{R}_3$=	ld
$\mathcal{R}_4$=	4
$\mathcal{R}_5$=	?
$\mathcal{R}_6$=	g
$\mathcal{R}_7$=	?
$\mathcal{R}_8$=	8

This cycle we are (potentially) able to issue both ϕ^1 operations *use* and k since both are ready—this is shown here. Thus we now have one operation left to process:

Cycle 6

ϕ^1

$h : \mathrm{r}(\mathcal{R}_5)$

	ϕ^e	
src	*op*	*dest*
$\surd$	(ld)	$\surd$
$\surd$	(f)	
$\surd$	(use)	$\surd$
$\surd$	(g)	$\surd$
—	h	—
$\surd$	(k)	$\surd$

ϕ^2

$h : \mathrm{w}(\mathcal{R}_7)$

Register File	
$\mathcal{R}_1=$	1
$\mathcal{R}_2=$	2
$\mathcal{R}_3=$	*ld*
$\mathcal{R}_4=$	4
$\mathcal{R}_5=$	*use*
$\mathcal{R}_6=$	*g*
$\mathcal{R}_7=$	?
$\mathcal{R}_8=$	*k*

And after the final execution we have

Cycle 7

ϕ^1

	ϕ^e	
src	*op*	*dest*
$\surd$	(ld)	$\surd$
$\surd$	(f)	
$\surd$	(use)	$\surd$
$\surd$	(g)	$\surd$
$\surd$	(h)	$\surd$
$\surd$	(k)	$\surd$

ϕ^2

Register File	
$\mathcal{R}_1=$	1
$\mathcal{R}_2=$	2
$\mathcal{R}_3=$	*ld*
$\mathcal{R}_4=$	4
$\mathcal{R}_5=$	*use*
$\mathcal{R}_6=$	*g*
$\mathcal{R}_7=$	*h*
$\mathcal{R}_8=$	*k*

From this example it is clear that there was a 1˜ penalty from the increased latency of the *ld* operation due to a cache miss. Had the miss been scheduled for by the compiler there would not have been any penalty at all since other operations could have been scheduled in to the stall slots—however, this is not the typical situation in non-loop code and the use of dynamic scheduling will benefit these cases.

4 The NYFO VLIW model

The NYFO VLIW architectural model is based on the split-issue execution paradigm described in section 3 and allows the specification of a wide range of processor configurations through the use of different policies to control the transitions within the split-issue framework. At a high level—that of the virtual processor—the NYFO VLIW architectural model has the structure of a traditional static multi-operation issue machine. Instructions are fetched, decoded, issued, and executed precisely as scheduled. At a low level—that of the implementation processor—the NYFO VLIW architectural model has very different characteristics than the traditional static multi-operation issue. Latencies may differ from the architectural specification and resources may have contention for their use. This section describes the characteristics of the architectural model in detail and identifies the aspects of the model that provide the flexibility to model a wide range of architectures.

4.1 Stages in an NYFO VLIW model

The NYFO VLIW architectural model is organized as a sequence of four basic stages which are fetch, decode, invoke, and compute. These stages correspond to the major actions that occur within the NYFO VLIW architectural model and do not necessarily correspond to separate pipeline stages. Specific details on the ordering of events within any particular implementation are modeled by policies which provide for both event handling and time (cost) accounting.

The fetch stage is responsible for maintaining the code stream supply to the decode stage for processing. Unless otherwise directed, it continues to fetch contiguous memory along the current fetch path. Using branch prediction, other stages (primarily decode and compute) can adjust the fetch path to try and reduce the effective latency of branches. The fetch stage is not concerned with the boundaries and format of instructions—only the maintenance of a ready memory stream for the decode stage.

The decode stage assembles instructions from the memory stream provided by the fetch stage and delivers the assembled instructions to the invoke stage. Instruction assembly can be a simple or complex problem depending on the encoding used and whether or not the instruction stream is compressed to reduce the memory footprint. In the simple case, instructions are fixed size packets that are easily assembled. The problem with this simple scheme is that not all operation slots in an instruction are always filled and the presence of many unused operations in an instruction results in an inefficiently used instruction memory hierarchy. An alternative to the simple encoding is to use some form of compression—a tradeoff between memory utilization and decode complexity. This research may look into the characteristics of different encoding and compression schemes at a later point but these will not be considered further in this report. The delivered instructions are not just copies of uncompressed instructions from memory but are templates of the actions that are required by the architectural specification. This information is then processed by the invoke stage.

The invoke stage receives the assembled instruction template from the decode stage, adds in any dynamic information that is required for each contained operation, and delivers the specified actions to the compute stage for computation. This stage bears a strong resemblance to the instruction issue stage in a traditional processor and is effectively the instruction issue stage of the virtual processor. However, the notion of invoking instructions to generate and deliver the dynamic event instances reduces confusion with other points of issue within the implementation. Section 4.2 describes the actions of this stage in detail.

The compute stage continuously processes the events that have been delivered to it by the invoke stage. Specifics of when events are processed and how late events are handled are provided by the policies for a particular configuration. The invoke stage ensures that the events are scheduled for execution according to the implementation specification and the compute stage processes these events accordingly performing whatever actions are necessary to maintain the illusion of the virtual processor. Section 4.3 describes the actions of this stage in detail and introduces the aspects that are governed by specific policies.

4.2 Elaboration on the invoke stage

The invocation of an instruction results in two primary actions that are performed within the invoke stage. These actions are the collection and merging of relevant processor configuration information with the individual operations and the delivery of the operation events to the compute stage.

The view that the compiler has of an operation is that events occur at specific times based on the architectural specification. The compiler schedules operations into specific instructions based on the availability of results and on its view of the configuration of the processor at the scheduled cycle. Each operation is placed into a specific slot in the instruction which may be restricted in which operations may be placed there. These slots correspond to execution resources within the virtual processor. In the ideal processor, there are sufficient resources to satisfy all possible operation configurations without contention. In the NYFO VLIW architectural model there is a split-issue structure (ϕ^1, ϕ^e, and ϕ^2) that corresponds to each slot—there may or may not be an execution resource that corresponds to each slot depending on the details of the implementation.

The schedule that the compiler produces is based on the architectural specification and the compiler must keep track of any changes to the processor configuration that the scheduled code produces during execution. As the processor configuration changes during program execution, the code that the scheduler generates must take these changes into account—the same effective operation may be generated differently depending on where the compiler schedules it to take these changes into account. The reference point that the compiler uses to make these decisions is the scheduled invocation time. While not the only point that could be chosen, it appears to be the best for several reasons not the least of which is the simplicity that it provides. Because the processor configuration that affects the behavior of an operation is the processor configuration at the time of invocation, the relevant information must be carried with the operation throughout the remainder of the execution process so that operation events can correctly execute out-of-order within the machine framework at the time of invocation.

One example of a processor configuration change that affects the details of operation scheduling and execution is the use of a rotating register file. A rotating register file renames its registers in a systematic way—typically through a rotation base pointer which is added to any register reference to determine the actual register. Since the value for base pointer can change during the execution process, the relevant value for the base pointer must be maintained with an operation after the invocation process completes in order to reference the proper register at a later time in the presence of further rotations or changes to the base pointer.

All relevant configuration information is required before invocation can complete for a given virtual cycle. When this information is not available, virtual time must be stalled until it is available. Once the configuration information is collected, the operations must be inserted into the compute stage for evaluation. This is done in parallel for each slot and involves placing the individual events—acquisition, execution, and delivery—into the appropriate position in the structures—ϕ^1, ϕ^e, ϕ^2—so that the events occur at the appropriate time. In addition to ensuring that there are no data dependencies within the instruction, the compiler also ensures that there are no resource conflicts. This allows the insertion of these events into the ϕ^1 and ϕ^2 queues without checking for the presence of existing entries—if any conflicts exist then there is a scheduling problem and the results are indeterminate. Once all events for all operations are inserted into the appropriate places within the compute stage, the invoke stage is complete for the current virtual cycle. Example code fragments that show the invoke stage and its relation to the other stages is shown in appendix A.

4.3 Elaboration on the compute stage

The computation process performs a sequence of steps to emulate the virtual processor according to the architectural specification. These steps include the acquisition of source operand values, the computation of result values, and the delivery of result values to destination operands.

Computation is performed in four distinct steps that in some senses comprise an embedded processor within the compute stage that autonomously processes whatever events it has (which are inserted by the invoke stage each virtual cycle). These four steps process the data in the split-issue queues (described in section 3.1) that exist for each instruction slot and manage the allocation of resources when they are limited. The register binding and the acquisition of source operands is performed by the resolve step, the selection of ready operations and the allocation of execution resources to perform the required computation is performed by the issue step, the collection of computed results from the execution resources is performed by the complete step, and register binding and the delivery of result values to destination operands is performed by the post step.

Register binding is the process of mapping an architectural register (as specified in an access event) to a specific physical register on the processor. Not only does it establish this mapping but it also results in a connection between the register and the referencing event. In a simple processor that does not rename registers, binding is essentially an identity function. However, when registers are renamed, binding obtains a new physical register that replace the currently mapped physical register. The referencing event is then bound to this physical register. Previous bindings are unaffected since the physical register that they refer to cannot be reallocated if it is still in use. Once a register and an event are bound, that register cannot be reassigned until that event (and any other events bound to the same register) have completed. Binding registers must be performed in virtual time according to the architectural specification. When a register cannot be bound for any reason, virtual time must stall and cannot continue until all binding is completed for that virtual cycle. Binding is the only action performed during computation that occurs in virtual time—the rest of the actions occur as values and resources are available in real time.

Binding a register for acquisition is straightforward—there is always a register available that corresponds to the architectural register which can be referenced by the acquisition event (guaranteed by the nature of the posting step). Thus, while it is possible that resource limitations prevent the binding from completing, it is not possible that the appropriate register is not available for binding. Binding a register for delivery may be more complicated depending on the register allocation policy in use. A register may be unavailable for binding if either the register has outstanding acquisitions depending on it or there are no new registers that can be assigned in a renaming process. There may also be resource limitations which prevent the binding of an available register.

Once binding is completed, there is much similarity in the way that the ϕ^1 acquisition events, the ϕ^e execution events, and the ϕ^2 delivery events are handled. Ready events are selected from the queues, resources are distributed as available, and those events that can have been allocated resources are evaluated. For acquisition events, the destination temporary storage is updated with the value from the source register; for execution events, the allocated execution resource for computation is provided with the source operand values and specified function; and for the delivery events, the destination register is updated with the value from the source temporary storage. Which events are selected for evaluation and the mechanism for allocating resources are policy issues.

For the simplicity of the model, each step in the sequence is executed in turn. Thus resolve occurs first followed by issue, complete, and post. This provides the read-execute-write sequencing described in section 3.1 but does not take into account any of the timing issues that would need to be resolved to make a workable implementation. Implementation details are beyond the scope of this report and will be presented at a later point in this research.

4.4 Implementation policies

Detailed policy definitions is critical to the precise modeling of different architectural specifications; however, a detailed examination of these issues is beyond the scope of this report. This section briefly introduces the different aspects of the stages that are implemented through policy specifications and provides insight into the nature and scope of these policies.

The fetch stage performs the interface to the memory system and maintains the memory stream to the decode stage. The fetch policy includes such issues as inline and alternate path prefetch behavior, fetch width, and so forth.

The decode stage converts the instruction stream into expanded instruction packets and maintains a source of decoded instructions to the invoke stage. The decode policy includes such issues as instruction decompression, timing template generation, and so forth.

The invoke stage performs the invocation of the instructions and the placing of the individual events within the split-issue structures in the compute unit. The invocation policy includes such issues as configuration, number of virtual cycles processed in a single real cycle (also affected by the ability of the compute unit to bind registers at this rate). In the NYFO VLIW architectural model, each virtual cycle is processed in turn before real cycle actions are taken. This is possible to do concurrently just as in a dynamic multi-instruction issue processor but is much more complicated to model—the sequential processing of virtual cycles has the same effect (without the complexity) that concurrent processing would have. This can be seen in the code fragments in appendix A.

The compute stage performs the evaluation of the instruction and operates on the events that are provided in its split-issue structures by the invoke stage. The execution policy includes many issues that affect the behavior of the compute unit. Binding registers is a policy issue that may limit the ability of the invoke stage to process multiple invocations (in virtual cycles) within a single real cycle. It also affects the ability to reorder events within the split-issue structures.

Scheduling and allocating resources is another significant policy issue that has performance limitations. Selecting which events to process—whether ϕ^1, ϕ^e, or ϕ^2—may be limited in a number of ways. While the selection policy can be different for the different queues, they share similar considerations. These include bus and port limitations, value availability, execution resource limitations, and so forth. The individual slots within the compute unit may be treated independently or in groups (or some of both) depending on the implementation specification.

Events which have completed computation return their results to the split-issue structures. Completion has potential resource limitations since the evaluation of events out-of-order may cause the results to be available when they are not expected and when resources are not adequate to transfer all results. This results in some execution resources stalling (and any events within their pipelines as well) and delaying the completion of their computation. The completion policy ensures that these limitations are detected and accounted for.

5 Example implementation descriptions

There are many different configuration variables—how many, how fast, how connected, how arbitrated, *etc.*—that defining a simple description scheme is not possible without some restrictions in the scope of the problem. The approach taken in this research is to separate configuration into two separate aspects. One aspect is the quantizable information—how many, how fast, *etc.* Another aspect is the non-quantizable information—how connected, how arbitrated, *etc.* The quantizable information in the architectural specification and the implementation specification are similar. However, there is no non-quantizable information for the architectural specification since it defines the virtual processor. The virtual processor assumes that there are sufficient resources so that the instruction stream can be executed precisely as scheduled. This section will describe two example implementation specifications—one for a traditional static multi-operation issue processor, one for a traditional dynamic multi-instruction issue processor—and will present some simple modifications and their impact on the model. Each of the examples is based on similar hardware—a multiple execution unit processor with a simple non-rotating register file. However, the descriptions will be limited to a discussion of policies since these are the differentiating aspects of the processors.

5.1 Simple static multi-operation issue implementation

The static multi-operation issue processor that this section describes is a perfect match for the architectural specification of the processor with the exception of the memory system which uses data caches. There is no dynamic scheduling used in this example and delayed results stall virtual time. Register renaming is not used in this simple model.

This processor invokes a single instruction (each containing multiple operations) each virtual cycle and invocation is simple since the register file is simple. Register renaming is not necessary since virtual time stalls during posting if there are unavailable results and thus all registers appear to be available as scheduled. Resolution never generates a stall since values are always guaranteed to be ready at the appropriate virtual time. Issue always takes the next event since there is never contention for execution resources. Completion has no contention since (at most) only the scheduled results that are ready. Posting has all results with the possible exception of memory results available when scheduled which are the only source of virtual time stalls in this implementation.

Allowing outstanding incomplete events to exist with events from later virtual cycles only complicates things slightly. Binding now requires some mechanism for maintaining dependency information. Resolution and posting need arbitration to select which events should be processed. Other than a few policy changes, this new capability is a simple extension of the previous implementation with only a small increase in implementation complexity for a potentially significant increase in performance.

5.2 Simple **dynamic multi-instruction issue** implementation

The **dynamic multi-instruction issue** processor that this section describes is fairly traditional in that it maintains a window of fixed size that it attempts to issue multiple independent operations concurrently out of. Register renaming is used to eliminate false dependencies. Operations are evaluated out-of-order from within the window.

The instruction window is simulated through the manipulation of virtual time based on the current number of non-issued invoked operations within the **compute** stage. This processor invokes as many instructions (each containing a single operation) as necessary to maintain a constant window size (subject to instructions being available from the **fetch** and **decode** stages). Invocation keeps track of the number of non-issued operations and this figure must be updated as events are finished. Binding requires that register renaming be performed when required and since this may be limited, virtual time may be stalled. Resolution acquires values for as many events as possible (there are no dependencies since registers are renamed) but not all events may be processed due to resource limitations. Issue must select those events that can be matched with the available execution resources. Completion probably has no contention but there may be some limitations depending on the model. Posting is similar to resolution and may not be able to process all events. Virtual time stalls represent the actual stalls that would occur in a **dynamic multi-instruction issue** processor when the window is full. In this situation, the execution of operations (events in the **NYFO VLIW** architectural model) continues. As operations have all their events processed, more instructions are invoked to maintain the window size.

Allowing a maximum span of operations (representative of a non-compressing window) requires only a simple change to the policy that tracks the number of operations outstanding. Other changes to model different aspects of **dynamic multi-instruction issue** processors can be added using similar limitations—the structure supporting it is there already. This is indicative of one significant problem in actual **dynamic multi-instruction issue** implementations—there are simplifications that can be made; however, any hardware reductions tend to be minimal since most of the hardware is required for even a significantly limited implementation (although performance may suffer more in keeping with expectations).

6 Conclusion

This report has reviewed some problems with existing dynamic multi-instruction issue and static multi-operation issue processors and identified three key characteristics of these processors that are important to consider while examining hybrid processor designs. Based on these characteristics, we have proposed a new architectural model—the NYFO VLIW architectural model—that supports the study and analysis of these hybrid designs. Fundamental to this new model is the separation of the virtual processor that is visible to the user and the compiler and the implementation processor that is used to perform the computations. Since the implementation processor maintains the illusion of the implementation of the virtual processor, variations in configurations and behavior are allowed. Although it is advantageous to maintain a close correspondence between the virtual and implementation characteristics, there is no requirement to do so.

We are in the process of building software tools to assist in the study and analysis of these hybrid architectures. Using these tools we will develop an understanding of the performance consequences of different configuration and policy choices within the implementation processor. By better understanding these effects we will be able to focus in on those techniques that appear to be the most promising for achieving cost-effective high-performance hybrid processor designs.

7 Acknowledgments

I would like to thank the following individuals for their support and assistance during this work. Without their time, assistance, and support this research would not be possible.

- Michael J. Flynn, Stanford University
- B. Ramakrishna Rau, Hewlett-Packard
- Michael D. Schlansker, Hewlett-Packard
- Wen-mei Hwu, University of Illinois at Urbana-Champaign
- John C. Gyllenhaal, University of Illinois at Urbana-Champaign

A Illustrative code fragments

Examining some simple psuedo-code fragments may help in explaining the sequencing and relationship of these events. The first fragment, `cycle_nv_processor_t`, defines the top-level behavior of the model. When the model is initialized, `issueState` is set to the value `nv_is_invoke` and `virtualStall` is set to the value `false`. Each function in the virtual execution loop does one of two things: either it updates the issue state to the next state (if the current state completed processing for the current virtual cycle) and sets `virtualStall` to `false`; or it leaves the issue state at the current value and sets `virtualStall` to `true` indicating that the virtual cycle is incomplete and that processing must return to this point the next real cycle.

The parameter `concurrentIssues` specifies how many instructions are processed per real cycle. The state that the issue process is currently in is `issueState` and this controls which step is currently being processed. If everything goes according to the schedule, this will be updated from one state to another and will cycle through the appropriate number of times to complete the multi-instruction issue. When one state is incomplete, this will exit the loop and continue on with whatever real time processing can be completed this real cycle and then the virtual cycle will be continued where it left of the next real cycle.

```
void                    cycle_nv_processor_t(nv_processor_t *processor)
{
    int                 i;

    for (i = 0; i < processor->concurrentIssues; i++)
    {
        if (processor->issueState == nv_is_invoke)    invoke_nv_processor_t(processor);
        if (processor->virtualStall)                  break;

        if (processor->issueState == nv_is_bind)      bind_nv_processor_t(processor);
        if (processor->virtualStall)                  break;

        if (processor->issueState == nv_is_update)    update_nv_processor_t(processor);
        if (processor->virtualStall)                  break;

        processor->virtualCycles += 1;
        processor->issueState = nv_is_invoke;
    }

    compute_nv_processor_t(processor);
    decode_nv_processor_t(processor);
    fetch_nv_processor_t(processor);

    processor->realCycles += 1;
}
```

Invocation is performed in `invoke_nv_processor_t`. This fragment reflects the need to freeze the processor configuration for the events with the current instruction before the events are delivered to the compute unit for execution.

```
void                    invoke_nv_processor_t(nv_processor_t *processor)
{
    collect_nv_processor_t(processor);
    insert_processor_t(processor);
}
```

Computation is performed in `compute_nv_processor_t` and each step is executed in sequence—necessary to maintain the sequencing of read, execute, write that the model is based on.

```
void                     compute_nv_processor_t(nv_processor_t *processor)
{
    resolve_nv_processor_t(processor);     /* phase1 bind has already occured */
    issue_nv_processor_t(processor);
    complete_nv_processor_t(processor);
    post_nv_processor_t(processor);        /* phase2 bind has already occured */
}
```

References

[1] James T. Kuehn and Burton J. Smith. The Horizon supercomputing system: Architecture and software. In *Supercomputing '88*, pages 28–34, November 1988.

[2] B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towle. The Cydra 5 departmental supercomputer. *Computer*, 22(1):12–35, January 1989. A version of this artical appeared in The 22nd Annual Hawaii International Conference on System Sciences.

[3] B. Ramakrishna Rau. VLIW: Not your father's Oldsmobile. Invited talk, The 25th Annual International Symposium on Microarchitecture, December 1992.

[4] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The Multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[5] B. Ramakrishna Rau. Dynamically scheduled VLIW processors. In *The 26th Annual International Symposium on Microarchitecture*, pages 80–92, December 1993.