Levo: A Resource-Flow Computer

Abstract

A number of limit studies have concluded that there exists a large amount of Instruction Level Parallelism (ILP factors of 10 or more) in typical integer codes (e.g., SPEC). However, no machine has ever been constructed that realizes the available ILP. Most previous designs have proposed systems constructed of components that failed to scale linearly.

We are proposing a machine model that realizes much of the potential ILP present in codes. Our approach, which we call *Levo*, speculates across multiple data and control dependencies, executing many instructions, and then re-executing some of them when mis-speculation occurs. Levo utilizes a hybrid form of the Tomasulo algorithm which we call *resource flow*. Levo does not rely on any added compiler or software support to obtain ILP.

In this paper we present a number of the novel features of Levo, which include time-tagged execution, full run-time prediction, and resource flow computing. We have also included in Levo previously described architectural features that further increase ILP. Some of these features include disjoint eager execution and hardware-based unrolling and inlining. We will describe the Levo machine model and provide execution results for a single kernel taken from the SPEC95 compress benchmark. We are able to demonstrate instruction throughput of up to 16 instructions per cycle.

1 Introduction

Current and past processors execute using a data-flow or control-flow model of execution, or a combination of the two. Such hardware is unscalable since hardware cost grows superlinearly with either the ILP exhibited, the instruction window size or the number of processing elements. Very Long Instruction Window (VLIW) processing models, such as Intel's Itanium [1] or Transmeta's Crusoe [2], reduce this cost, at the expense of high compile-time, code incompatibility or reduced performance.

Since 1964 [3], a major focus of computer architecture has been to reduce dependencies or reduce their consequences. Although various limit studies [4, 5, 6] have shown the existence of large amounts of ILP (Instruction Level Parallelism) in typical general-purpose, branch-intensive codes, no physical machine or research machine model has realized high degrees of parallelism when realistic hardware assumptions are considered.

The highest ILP reported to date is that of Lipasti and Shen presented in [7]; their *superspeculative* engine exhibited on average greater than 7 Instructions Per Cycle (IPC), assuming instruction, memory and register dataflow. Enabling register dataflow provided the single most significant gain in ILP. Although their superspeculative engine strives to take advantage of both data and control speculation, it does not take advantage of *full speculation.* The best results obtained in their work considered a cache with an unlimited number of ports and a 128-entry reorder buffer (all appropriate assumptions in a limit study, but hardly scalable design points).

It is our contention that full speculation is needed to achieve both large amounts of ILP (and IPC) and scalable computing. Our definition of code execution that embraces full speculation using unlimited resources is:

Execute everything possible and then cleanup.

With limited hardware, the full speculation execution model becomes a *resource flow* model:

1. In this case, resources (processing elements) flow to the highest priority instructions every cycle, regardless of any instruction, data or control dependencies.

2. After execution, data loads, data stores and branch instructions broadcast their results to all instructions forward in time in the execution window.

3. A waiting instruction only re-executes if any of its inputs, data or predicates have matching addresses to the broadcast information, or if a datum or predicate has changed its value.

4. After a waiting execution receives an execution resource (i.e., it has priority for execution, and executes), its own data and predicate outputs are broadcast to the instructions forward in time in the execution window.

5. .... and the cycle repeats.

The basis for resource-flow computing is given in point 1 above; logically, the processing elements flow to the instructions with the highest priority, which are often the earliest instructions in the *execution window.* Dependencies are not explicitly maintained in any way, shape or form. They are implicitly preserved, after the fact, by the Tomasulo-inspired broadcast [8] of an instruction's data, address and *time tag.*

Levo utilizes time tags, as were used in the Metaflow computer [9] and as were proposed in the WarpEngine[10]. However, in Metaflow the main function of the time tag or stamp was to squash instructions after a branch misprediction. This class of execution model is the now a standard for managing data-flow on a single-path control speculation model (e.g., Intel P6) [11].

In both Levo and the WarpEngine, instruction order is preserved using timestamps. In both models, an instruction's source is the closest previous result having the same address as the source. The similarity ends here. WarpEngine timestamps are complex, may be generated by the compiler, require the broadcast of a *global virtual time* and require costly periodic rescaling. Any instruction in the WarpEngine execution window can have any timestamp value; 32-bits or more are required for the timestamp.

Levo's time tags are much simpler. Each instruction slot in the execution window has an associated time tag. The length of this tag grows with the log of the number instructions in the execution window. As instructions retire from the execution window and are committed, all remaining time tags have a value subtracted from them equal to the number of instructions committed. This subtraction necessitates the use of many down-counters; however, they are distributed and very small in length, so they are fast and cheap.

Similar to the Tomasulo algorithm, Levo has data buses used to forward instruction execution results to later instructions. Tomasulo called his bus the *Common Data Bus* (CDB). The blessing and curse of the CDB was that it is driven by, and snooped by, all instructions in the execution window. The hardware cost relationship is quadratic: as the size of the window grows, the address matching hardware cost grows as the square of the window size. This is, however, not necessary.

Franklin and Sohi [12] demonstrated that register lifetimes are short, typically equal to one or two basic blocks (the average length of a basic block ranges from 6-15 instructions on a RISC-based ISA). This arises because the compiler always tries to make good use of the available registers and reuses them as soon as possible.

Levo takes advantage of this key observation. Its *forwarding buses* are fixed, independent of the size of the execution window, and are set with a *forwarding span* length necessary to capture average register lifetimes (one or two basic blocks). In our present implementation of Levo, this has been set to 32 instructions. Further, since it is likely that multiple instructions within a span will be executed in any given cycle, there are multiple forwarding buses per forwarding span. Figure 1 shows a scaled down version of Levo (4 ASs per sharing group), indicating how the forwarding buses are interconnected between columns. A *Forwarding Register* (FR) is used to buffer messages from the previous column. We are currently determining the proper design and depth of these registers. Levo provides similar interconnections in the opposite direction (i.e., backwarding buses).

If the Levo execution window size is *N* instructions, there will be *N/32* groups of forwarding buses. This design choice works well since computations tend to be generally localized. Of course, we must allow for the case that the result of the first instruction is needed by the last instruction in the execution window. This will incur a multicycle forwarding operation. Because we limit the number of units attached to any bus, there is a one-cycle penalty for every span traversed during forwarding. This design tradeoff embodies the principle of *make the common case fast*; the closer a sourcing instruction is to the result-generating instruction, the less the forwarding penalty will be.

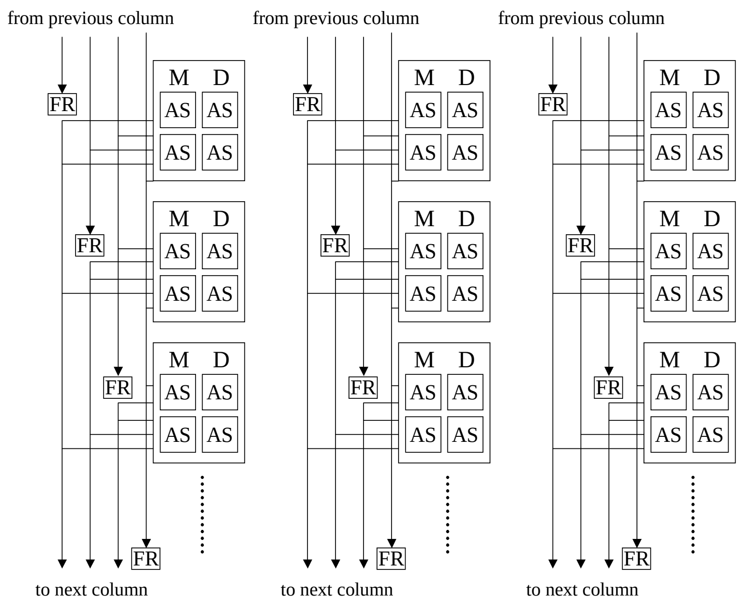

Figure 1: Forwarding buses used to communicate register values, predicate values, and memory values to later columns of execution. A Forwarding Register (FR) is used to buffer values between columns. M and D stand for *mainline* and *disjoint eager execution*, respectively.

The Levo microarchitecture presented herein is a realization of resource-flow computing. It is our conjecture that Levo is both scalable and realizes IPC's above 10 on general-purpose and branch-intensive code. The scalability of Levo is generally orthogonal to the execution model; it is likely that any ILP-oriented architectural model could use a subset of the Levo scalable microarchitectural features.

The remainder of this paper is organized as follows. Section 2 reviews related work on ILP models and Levo-specific microarchitectural features. Section 3 presents the Levo architecture in detail. Section 4 summarizes the contributions of this work and discusses future work.

2 Related Work

The design of Levo is motivated by two observations:

1. many ILP studies have concluded that there is substantial parallelism present in even the most control-oriented code (e.g., SPECint), and

2. predicated execution presents an attractive model for achieving scalable dataflow if predicates are left to be determined at runtime.

Next we review past work on architectural models and features which pursue high ILP as their objective.

2.1 ILP Architectural Models

Rau and Fisher described three forms of IPL [13]:

1. dependence architectures (i.e., dataflow),

2. independence architecture (i.e., VLIW), and

3. sequential architecture (i.e., superscalar).

2.1.1 Dependence Architectures

In a dependence architecture, an instruction *fires* or executes only when all of its inputs have been received. The instruction's result helps to fire other instructions. All dependences are explicitly specified, with each instruction providing a list of *successor* instructions. Instructions execute as soon as their operands are available (but not before). This guarantees that no work will be wasted, but also suffers from stalls waiting for dependent instructions to resolve. A number of attempts have been made to exploit an explicit data flow model of execution [14, 15].

2.2 Independence Architectures

The Multiflow machine [16] and the Cydra5 [17] are examples of independence architectures. These VLIW machines provided a complete specification of where each instruction is to be executed, and accomplishes this by architecting this into the instruction set architecture. More recent examples of commercial VLIW processors include the Intel Itanium [1] and the Transmeta Crusoe [2]. As was pointed out in a recent paper on Itanium [18], the reliance on the compiler to expose ILP at compile time can be a daunting task. VLIW execution is static. Current approaches to using predication use *visible* and *explicit* predicates; the predicates are controlled by the compiler and use storage explicitly present in the computer's instruction set architecture (similar to regular data registers or main memory); they are explicit since there is at least one 1-bit predicate hardware register associated with each instruction.

The Itanium architecture is a good example of this is level of predication. Itanium has 64 visible-explicit predicate registers [1]. Predicates cannot be effectively used when the processor executes traditional IA-32 (x86) machine code. Therefore, existing software cannot take advantage of Itanium without modification. Other microprocessors have similar constraints to x86 processors; predicates are not currently provided in their instruction sets, so they cannot take advantage of predication techniques (the one exception being Crusoe, though at a performance penalty due to code morphing).

Levo provides for *Hidden Explicit Prediction*, where predication is performed on any binary at runtime. A major advantage of HEP is its applicability to all existing computer designs, including both those with and without predicates in their instruction set architectures. For example, popular ISA's such as the Intel IA-32, which contain little or no predication, can have full predication added without affecting the underlying instruction set.

2.2.1 Sequential Architectures

An early example of an aggressive sequential architecture was Tomasulo's reservation station and common data bus scheme as described in the IBM 360/91 [8]. Although little ILP was exhibited by the scheme in an absolute sense, many years later it reappeared in Patt's HPS model[19], and was then in turn incorporated into the Intel P6 microarchitecture [11]. The former only allowed one instance of an instruction to be active at a time, that is its execution was effectively limited to one active iteration of a loop at a time, greatly limiting its performance.

2.3 Levo Microarchitectural Features

While our present target for Levo follows an aggressive sequential architecture paradigm most closely, a number of the concepts embodied in Levo can be applied to VLIW architectures as well. Next we will review related work which is used in the design of Levo.

2.3.1 Control and Data Speculation

Levo utilizes a variation of branch speculation called *Disjoint Eager Execution (DEE)* [20]. DEE is a form of *multipath* execution; code is executed down both paths from a branch under certain circumstances. Only the most likely code paths are allocated execution resources (processing elements, etc.) The code execution is unbalanced; code on the predicted or Main-Line (ML) path is given preferential priority for execution resources over code on the not-predicted or DEE paths. ILP of the order of 10 instructions per cycle was shown to be possible.

The prior proposed machine realization of DEE [20] required many large and cumbersome data dependency and control dependency bit matrices. Data and control issues were treated separately. Approaches to reducing the size of the matrices were partially devised but never proven. In Levo's implementation, we have specifically addressed the scalability of DEE.

2.3.2 Value Prediction

Lipasti and Sazeides have demonstrated the value of *data speculation* [21, 22]. In this scenario, input values for some instructions are predicted and the instructions allowed to execute speculatively. As with control speculation, there is a penalty for data value misprediction. Lipasti has shown that data value speculation can be used effectively to pre-execute instructions. Levo employs a last value predictor, though it is likely that future work will include more aggressive designs such as Calder's last N-value predictor [23] and Lipasti's *silent stores* predictor [24].

2.3.3 Loop Unrolling

Gonzalez et al. have shown that loop unrolling can be implemented in the microarchitecture, and that a history table can be used to unroll loops in the instruction window during instruction issue [25]. Levo utilizes this approach, and adds to this the ability to inline small procedures. We are also suggesting that software pipelining can be employed in the instruction window, since Levo effectively performs software pipelining in hardware through resource flow.

2.3.4 Time Tags

Time tags or the concept of *timestamping* have been used in two prior machines. Neither machine has been built. The first machine was the Metaflow processor[9]. Metaflow used time tags for a branch misprediction mechanism. When a branch is mispredicted, all instructions with tags later in time than the branch's are squashed. This makes it easy to determine anywhere in the machine if an instruction needs to be squashed. This idea is only slightly related to our use of time tags; our method is much more general and is applied to data as well. Further, our mechanism is used to actually get the right data to an instruction, something not attempted in the Metaflow architecture.

The second machine is the Warp Engine [26], which was based on the *Time Warp* concept first presented in [27]. The Warp Engine uses *timestamps* in a similar fashion to Metaflow, but goes beyond Metaflow by executing some code or events speculatively, then rolling back the state when necessary using *anti-messages*. Some code can be executed without regard to data dependencies (as is done in Levo), but data dependencies must still be computed and maintained in order to determine the proper rollback state. Levo uses a very simple algorithm; time tag reuse is straightforward

The Warp Engine used timestamps to order speculative memory loads and stores correctly. Their hardware algorithm required the use of many relational comparisons (less than, greater than). Used in their *time-space* cache, their hardware is more costly and complex than a sizable fully-associative cache. The Levo algorithm uses its time tags to similarly order operations, but first, it does so without relational comparisons, and second, the method is applied uniformly to all instructions including assignment operations (adds, etc.), memory accesses and branch or predicate operations.

More recently, Wood et al. [28] describe a *timestamp snooping* mechanism used to provide a total logical order of memory address operations. While this work is not directly applicable to Levo, it demonstrates that timetags can expose parallelism in a stream of operations, only worrying about ordering after the fact.

2.4 Competing Machine Model Philosophies

The design most similar to Levo's overall machine philosophy is the *Multiscalar* project [29]. Multiscalar processors divide a program into a set of tasks (i.e., basic blocks or groups of basic blocks), and assign them to a set of parallel processing units. Register results are broadcast to the appropriate location, as specified by compiler-generated masks. Masking registers use a bit to indicate if a register value generated in a prior task is needed by a later task. The bit is cleared once the value is transmitted.

Multiscalar differs from Levo in a number of ways. In Multiscalar the register value production and consumption among code regions (their "tasks" versus our "forwarding spans") must be determined by the compiler and transmitted to the processor with the code for execution. Hence, there is a code compatibility problem with Multiscalar; programs must be re-compiled before they can be used on the machine. However, this is usually not possible on commercial code. Further, the Multiscalar scheme is fully static (i.e., it cannot adapt based on the current contents of the instruction or execution windows, as can Levo).

Another competing machine model is Rotenberg's *Trace Cache* [30]. This model proposes to predict entire sequences of branches and caches these sequences for ready access. It exhibits substantially improved performance over simple single-path speculation alone, however the ILP achieved is still much lower than what is possible. Recently an approximation to Levo's control dependency design was incorporated into a Trace Cache mechanism [31].

The *superspeculative* processor[7] uses a Trace cache and Speculation of various types throughout its data path. Using realistic assumptions, Lipasti and Shen obtained an average IPC of about 7.

Levo can coexist with VLIW/EPIC style architectures[16, 18, 32], easing the scheduling task of the compiler. Traditionally the argument has been that this is not an issue, since compilation is usually done once when there are no time constraints. However, reducing the scheduling complexity in the compiler can reap two major benefits:

1. there is less of a chance of introducing errors (not all optimizations are safe), and

2. run-time (i.e., dynamic) compilation should be aided substantially.

3 Levo: Resource Flow Computing

Figure 2 provides a high-level view of the Levo machine model. Our present configuration of Levo has 4 *Sharing Groups* per column, with a total of 32 mainline and 32 DEE ASs per column. The length of a row is variable. While not shown here, columns are interconnected using a torus topology. Next we will describe individual components of our model.

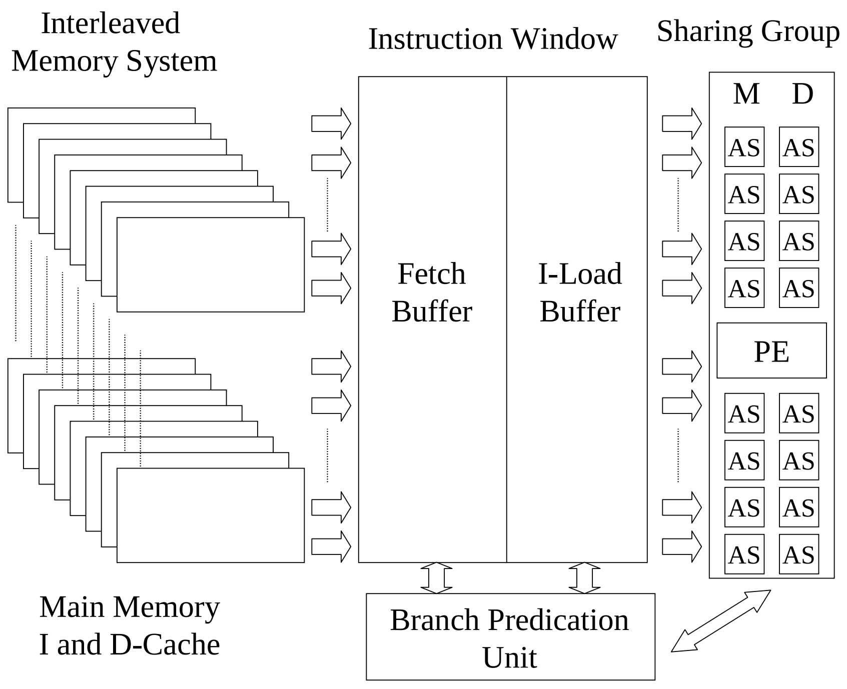

Figure 2: The Levo machine organization.

3.1 Instruction Window and Instruction Fetch

The Levo instruction stream is produced by exploiting techniques developed by Uht [20], namely *Disjoint Eager Execution* (DEE). The concept of DEE is to utilize the program control graph, and label each edge (each edge represents a basic block) with a probability value signifying the chances of traversing that path during execution. The graph is used to guide instruction fetch and instruction issue. Instruction fetch and issue will follow the *Main Line* (ML) path until the probability of pursuing this path is less the probability of following an alternative path (a DEE path). This switch will occur when we have speculated down the ML path to a point where the probability of continuing down this path is lower than the probability of following a DEE path. DEE is presented in detail in [20], and so we will not discuss the instruction fetch/issue logic further in this paper.

Some other techniques that Levo exploits are runtime loop unrolling and procedure inlining. These are accomplished in hardware. Levo will dynamically detect the occurrence of a loop (i.e., a conditional branch with a negative target displacement, or a conditional branch with a positive target displacement and whose target instruction is an unconditional jump with a negative target displacement). Levo caches loop instances and unrolls loops in the instruction window based on past loop execution history.

Tubella and Gonzalez provide statistics on dynamic loop execution for SPEC-95 benchmarks [25]. Of particular interest to us is the average number of instructions in a loop for SPECint (loops in SPECFP tend to be much longer, so they are generally poor candidates for hardware-based unrolling). This number varied from 40 instructions (m88ksim) to 336 (ijpeg), with 4 out of 8 programs having less than 100 instructions executed in a loop (on average). This suggests that hardware-based unrolling could be employed aggressively.

3.2 Levo Active Stations

An Active Station (AS) is one of the main components of the Levo machine model. It replaces the concept of reservation stations, which are used in current modern processors, by extending their functionality. After an AS is loaded with an instruction, it will make sure that the instruction is executed with the latest input value of the register, memory or predicate. The speculative nature of the instruction execution in an AS is one of the sources for achieving higher ILP.

Each instruction in the execution window has associated with it a dynamically changing time tag. The time tag is formed as the concatenation of the column address of the instruction with the row address of the instruction. This composite tag is just the position of the instruction in the execution window. In the following discussion, for the sake of simplicity we assume that the time tags are fixed with respect to the physical AS. In reality, the columns of ASs can be renamed, i.e., any physical column can effectively be the “leftmost” column.

Figure 3 shows the interface between an AS and the rest of the system. Eight buses are used to broadcast the results of computations to other Active Stations, as well as to other components in

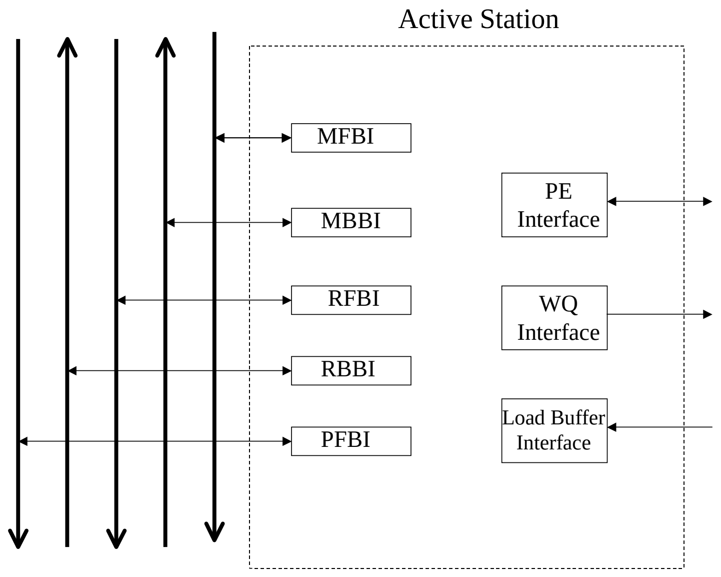

Figure 3: The Levo Active Station. Each Active Station holds an instruction, multiple Active Stations make up a Sharing Group and each Sharing Group holds both a Main Line and a Disjoint Eager Execution path.

the system. The following list describes the task of each bus:

- Register Forwarding Bus (RFB): Sends the newly computed value of a register to the other Active Stations with a higher time tag.
- Register Backwarding Bus (RBB): Requests for a register value are sent through this bus. Either the Active Station with the most recent value, or a *register filter unit* will provide the data on the Register Forwarding Bus.
- Memory Forwarding Bus (MFB): Sends the new computed value of a memory location to the other Active Stations with higher time tag values.
- Memory Backwarding Bus (MBB): Requests for a value of a memory locations are sent through this bus. Other Active Stations or the Write Queue will provide the latest value.

- Predicate Forwarding Bus (PFB): the value of each predicated address, computed at branch locations, is forwarded on this bus. An Active Station executes its loaded instruction based on the value of its corresponding predicate value.

- Write Queue Interface: This bus is used to send the final committed memory values to the write queue. The write queue will then collapse different values for a memory location and select the latest value to be written into the first level of memory hierarchy.

- Processing Element Interface: A set of Active Stations are usually grouped together to create what is named as a Sharing Group. The Active Stations in a Sharing Group would share the same resources such as ALU or Floating-point unit to execute their corresponding instruction.

- Load Buffer Interface- The unit that dispatches instructions from the I-Load Buffer to the ASs.

The concept and implementation of an Active Station are central to the operation of the resource flow computer. The Active Station is based on the classic Tomasulo *reservation station*[8], but has significantly more functionality. Like a reservation station, the main function is to *snoop* or look at one or more buses carrying the results of the computation from the processing elements, *snarfing* or copying the information from a bus into the station's own storage. A reservation station snarfs the data when the register address on a bus is equal to the contents of one of the station's source address registers. In a reservation station, the corresponding instruction is fired (sent to a processing element) when all of its sources have been snarfed.

Each AS snoops all the interface buses in each clock to see if there is any new value for its source register memory or predicates. Once such a value is observed, it would replace the old value and if it is necessary, the instruction would be re-executed. The new register, memory or predicate values are then sent over to the other Active Stations through the forwarding buses.

Given the above information, we can now explain the operation of an AS. Suppose we have a column of idle Active Stations. Once a column worth of instructions are fetched from memory, decoded, and assigned the corresponding predicate addresses, they are stored in the instruction load buffer. The Active Stations are then loaded with instructions from the load buffer and are assigned a time tag value. The initial values for memory and registers are loaded and predicates generated to control whether an instruction should be speculatively executed. Note that since value prediction is allowed, instructions could actually be speculatively executed. The new computed values for the register or memory are then forwarded on the forwarding buses and are snooped and potentially snarfed by other Active Stations. The branches in the Active Stations are responsible for evaluating the predicate and *canceling predicate* values. These values are then sent over the predicate forwarding bus to the other ASs. Branch prediction can be used to set the initial value for the predicates.

Active Stations and reservation stations differ in the following respects:

1. Time tags and ISA register addresses are used instead of the arbitrary renaming addresses in the Tomasulo algorithm. The Tomasulo algorithm does not explicitly represent time in its reservation stations; correct execution order is maintained by having computations chained to follow the data flow of the code in the execution window. The resource flow computer's usage of time tags allows it to dynamically change both the ordering of instructions and the time when instructions get new data.

2. There are four conditions for snarfing data in an Active Station:

   (a) the broadcast register address must equal the source address;

   (b) the broadcast register value must be different from the current value of the source;

   (c) the broadcast time tag must be less than the time tag of the source correct Active Station;

   (d) and the broadcast time tag must be greater than or equal to the time tag of the last datum snarfed for the source.

   The latter two conditions ensure that only the register value produced closest to the Active Station, but not after it, is used by the source.

Sharing group	I-buffer position	Execution cycle	Instruction
0	65	3	addiu sp,sp,-80
0	66	(4),5	sd gp,0(sp)
0	67	**5**,7	addiu at,at,-26388
0	68	**6**,8	addu gp,t9,at
0	69	(9),10	lw v1,-32508(gp)
0	70	3	li a2,8
0	72	(10),11	lw a5,-32368(gp)
0	71	11	bltz a0,(far away)
1	73	**(3)**,(4),5	sd s0,24(sp)
1	74	3	lui s0,0x6
1	75	**5**,12	andi a7,a5,0x7
1	76	**(6),(7),8**,(9),10	lw a6,-32740(gp)
1	77	(8),9	sd s6,40(sp)
1	78	10	ori s0,s0,0x51e1
1	79	**11**,13	sllv at,a0,a7
1	80	(14),15	lw a1,-3204(gp)

Table 1: Example Levo code execution taken from the compress benchmark from SPEC95. Column 1 designates the Sharing Group involved, column 2 shows the instruction location in the instruction fetch buffer, column 3 shows the cycle in which execution resources are consumed in the processing element, and column 4 shows the instruction with its operands. Parentheses represent a memory address generation, and bold numbers indicate results that are later squashed.

3. The Active Station incorporates a small amount of logic to make predicate calculations.

4. A branch in the instruction stream takes up one Active Station in the execution window. Branches to targets outside of the execution window are handled in conjunction with the tracking buffer.

5. The predication mechanism works similarly to the snooping and snarfing mechanism for data communication. Therefore there is a unified approach to the handling of control flow and data flow.

3.3 Levo Execution

Levo may be realized with any RISC or CISC instruction set and does not require any change to existing code, which may avoid classic code compatibility issues. Levo also uses substantially less hardware and simpler hardware than prior methods.

Table 1 shows how a portion of the SPEC95 compress benchmark as run on Levo. We are showing code generated for the MIPS R4000 ISA, though Levo can be used for executing any ISA. We are modeling a moderately aggressive (two cycle) instruction fetch and data store/fetch. Address calculations vie for shared resources with ALU operations. Every load or store is a two cycle operation (address generation, store or fetch using this address). We are modeling here a fairly simple Processing Element, where address generation shares the ALU datapath.

In this example we are looking at two 8-instruction Sharing Groups, where each Sharing Group has shared resources, and where execution results are broadcast to other Sharing Groups located in the span in a single cycle. Execution is shown from cycle 3 since we are looking at column 3 in the execution window.

Some things to notice in Table 1:

- I-67 in Sharing Group 0 (addiu instruction) attempts to execute in cycle 5, but this result is squashed due to a dependency that occurred in a prior column (which we have not shown),
- I-68 in Sharing Group 0 (addu) attempts to execute in cycle 6, but again this result is squashed due to a dependency that occurred in cycle 7 from I-67 modifying *at*,
- I-73 in Sharing Group 1 (sd) attempts to perform address generation in cycle 3, but this result is squashed due to a dependency that occurred in cycle 3 from I-65 modifying *sp*,
- I-75 in Sharing Group 1 (andi) attempts to execute in cycle 5, but this result is squashed due to a dependency that occurred in cycle 11 from I-72 modifying *a5*,
- I-76 in Sharing Group 1 (lw) attempts to perform address generation in cycle 6, but this result is squashed due to a dependency that occurred in cycle 6 in I-68 modifying *gp*, then attempts to use the speculative version of *gp* generated by I-68 i cycle 6, but this update is squashed,

and the actual value of *gp* finally arrives in cycle 8, though we had to squash both the address generation and memory access for I-76 in cycles 7 and 8, and

- I-79 in Sharing Group 1 (lw) attempts to execute in cycle 11, but this result is squashed due to a dependency that occurred in 12 for I-75 modifying *a7*.

While the above two sequences of 16 instructions retire in a total of 12 cycles, in our current implementation of Levo we have 30 additional Sharing Groups running in parallel to these two (32 Sharing Groups in total). We have run a kernel of this benchmark through a Levo simulation and obtained an IPC of greater than 16 instructions per cycle.

When every instruction in the leftmost or earliest column has finished executing, the column is *retired* by effectively shifting the entire execution window contents to the left by one column. This changes the time tags of every instruction in the execution window, effectively decrementing the column address part of every instruction's time tag by one. The results from the retired column are sent to both the register file copies and the memory system, as appropriate.

3.3.1 Scalability

A key design point of Levo is that it is the first high-ILP machine model that is scalable. By scalable we mean that the hardware cost (amount of hardware) of the machine grows linearly with the number of Processing Elements. Machines with dependency matrices grow at least as quickly as the square of the number of PE's. The hardware cost of existing machines also typically grows with the square of the number of PE's.

The hardware cost of our resource flow machine grows no faster than linearly because there is no dependency storage, generation or checking hardware, and because the size of the forwarding buses is fixed, that is, the forwarding span normally stays the same regardless of the number of PE's or Active Stations. Since the number of buses grows as a constant fraction of the number of Active Stations, the hardware cost of the buses also grows linearly with the number of PE's.

4 Conclusions

In this paper we have described a new machine model called Levo which realizes much of the potential ILP present in codes. Our approach speculates across multiple data and control dependencies, executing many instructions, and then re-executing some of them when mis-speculation occurs, while utilizing Hidden Explicit Prediction to facilitate resource flow.

We have provided an overall model description of Levo and included a single datapoint showing the inherent benefits of Levo (potentially 16 IPC). We are presently completing a cycle-accurate machine simulator called *LevoSim*. We are modeling the design of this simulator after the popular SimpleScalar architectural toolkit. LevoSim will provide cycle-accurate simulation results of complete applications, including system calls and libraries. We are close to running complete programs (versus kernels as we did with compress) to validate the ideas presented in this paper. We will report results for a subset of the SPEC-2000 benchmark suite in the final version of this paper.

References

[1] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir, "Introducing the ia-64 architecture," in *IEEE Micro*, pp. 12–23, September 2000.

[2] B. Glass, "Crusoe: Transmeta comes out of the closet," in *http://www.linuxplanet.com/ linuxplanet/reports/1441/1/*, 2000.

[3] J. Thornton, "Parallel operation in control data 6600," in *Proceedings of the AFIPS Fall Joint Computer Conference*, pp. 33–40, 1964.

[4] E. Riseman and C. Foster, "The inhibition of potential parallelism by conditional jumps," *IEEE Transactions on Computers*, vol. C21, pp. 1405–1411, December 1972.

[5] N. Jouppi and D. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proceedings of the International Conference on Architectural Suport for Programming Languages and Operating Systems*, pp. 272–282, April 1989.

[6] M. Lam and R. Wilson, "Limits of control flow on parallelism," in *Proceedings of the International Symposium on Computer Architecture*, pp. 46–57, May 1992.

[7] M. Lipasti and J. Shen, "Superspeculative microarchitecture for beyond ad 2000," in *IEEE Computer Magazine*, pp. 59–67, September 1997.

[8] R. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, January 1967.

[9] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman, “The metaflow architecture,” in *IEEE Micro*, June 1991.

[10] J. Cleary, R. Littin, D. McWha, and M. Pearson, “Scaling the reorder buffer to 10,000 instructions,” in *IEEE TCCA News*, pp. 16–20, June 2000.

[11] D. Papworth, “Tuning the pentium pro microarchitecture,” in *IEEE Micro*, pp. 8–15, April 1996.

[12] M. Franklin and G.S. Sohi, “The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism,” in *Proceedings of the 19th International Syposium on Computer Architecture*, pp. 58–67, ACM, May 1992.

[13] B. Rau and J. Fisher, “Instruction-level parallel processing: History, overview and perspective,” *International Journal of Supercomputing*, vol. 7, pp. 9–50, October 1996.

[14] Arvind and K. Gostelow, “The u-interpreter,” *IEEE Computer*, vol. 15, pp. 12–49, February 1982.

[15] J. Gurd, C. Kirkham, and I. Watson, “The manchester prototype dataflow computer,” *Communications of the ACM*, vol. 28, pp. 34–52, January 1985.

[16] R. Colwell, R. Nix, J. O’Donnell, D. Papworth, and P. Rodman, “A vliw architecture for a trace scheduling compiler,” *IEEE Transactions on Computers*, vol. C-37, pp. 967–979, August 1988.

[17] G. Beck, D. Yen, and T. Anderson, “The cydra 5 minisupercomputer: Architecture and implementation,” *Journal of Supercomputing*, vol. 7, pp. 143–180, 1993.

[18] R. Zahir, J. Ross, D. Morris, and D. Hess, “Os and compiler considerations in the design of the ia-64 architecture,” in *Proceedings of the International Conference on Architectural Suport for Programming Languages and Operating Systems*, pp. 212–221, November 2000.

[19] Y. Patt, W.-M. Hwu, and M. Shebanow, “Hps, a new microarchitecture: Rationale and introduction,” in *IEEE Symposium on Microarchitecture*, pp. 103–108, December 1985.

[20] A.K. Uht and V. Sindagi, “Disjoint Eager Execution: An Optimal Form of Speculative Execution,” in *Proceedings of the 28th International Symposium on Microarchitecture, MICRO-28*, pp. 313–325, ACM-IEEE, November/December 1995.

[21] M. Lipasti, C. Wilkerson, and J. Shen, “Value locality and load value predication,” in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138–147, September 1996.

[22] Y. Sazeides and J. Smith, “The predictability of data values,” in *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 248–258, December 1997.

[23] B. Calder, P. Feller, and A. Eustace, “Value profiling,” in *Proceedings of the 30th IEEE Symposium on Microarchitecture*, December 1997.

[24] K. Lepak and M. Lipasti, “On the value locality of store instructions,” in *Proceedings of the International Symposium on Computer Architecture*, pp. 182–191, June 2000.

[25] J. Tubella and A. Gonzalez, “Control speculation in multithreaded processors through dynamic loop detection,” in *Proceedings of the 4th Symposium on High Performance Computer Archtitecture*, pp. 14–23, January 1998.

[26] J. Cleary, M. Pearson, and H. Kinawi, "The architecture of an optimistic cpu: The warp engine," in *Proceedings of HICSS'95*, pp. 163–172, 1995.

[27] D. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, pp. 404–425, July 1985.

[28] M. Martin, D. Sorin, A. Ailamaki, A. Alameldeen, R. Dickson, C. Mauer, K. Moore, M. Plakal, M. Hill, and D. Wood, "Timestamp snooping: An approach for extending smps," in *Proceedings of the International Conference on Architectural Suport for Programming Languages and Operating Systems*, pp. 25–36, November 2000.

[29] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar processors," in *Proceedings of the International Symposium on Computer Architecture*, pp. 414–425, June 1995.

[30] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace processors," in *IEEE Symposium on Microarchitecture*, pp. 138–148, December 1997.

[31] E. Rotenberg and J. Smith, "Control independence in trace processors," in *IEEE Symposium on Microarchitecture*, pp. 4–15, December 1999.

[32] B. Rau, D. Yen, W. Yen, and R. A. Towle, "The cydra 5 departmental supercomputer: Design and philosophies, decisions and tradeoffs," *IEEE Computer Magazine*, vol. 22, pp. 12–34, Jan 1989.