University of Rhode Island
Dept. of Electrical and Computer Engineering
Kelley Hall
4 East Alumni Ave.
Kingston, RI 02881-0805, USA

Technical Report No. 092001-001 September 18, 2001

IPC in the 10's via Resource Flow Computing with Levo

Augustus K. Uht*, David Morano°, Alireza Khalafi°, Marcos de Alba°,
Thomas Wenisch*, Maryam Ashouei°, David Kaeli°
*University of Rhode Island °Northeastern University

Abstract[1]

High ILP (Instruction Level Parallelism) exists in typical integer codes. Our goal is to create a machine called Levo that will realize this potential ILP in the face of real hardware limitations. Levo is modeled and evaluated at three different levels of abstraction. FastLevo models the potential of Levo via high-level trace-driven simulation embodying key hardware constraints. LevoSim is a detailed cycle-accurate execution-driven simulator; it gives us precise performance data on the Levo architecture. Lastly, HDLevo is a synthesizable VHDL model of the key elements of Levo; it gives us detailed Levo hardware cost estimates. Levo has many novel architectural features, including the use of time tags with instructions to implicitly enforce data and control dependencies. Levo provides a registerless data path, in that there is no central register file or bottleneck. A resource-flow model of computation is used in which instructions execute whenever resources are available, regardless of true data or control dependencies. Levo obtains this high IPC with scalability in our hardware implementation. FastLevo shows IPC's in the 10's over a wide range of Levo configurations on a subset of the SPECint2000 benchmarks allowing for relaxed memory and control dependencies, though real machine hardware and data dependencies. With HDLevo we obtain a hardware cost for the Levo core (no PE's, memory or I-fetch) of less than 59 million transistors.

1 Introduction

The premise of our work is to produce a machine microarchitecture that realizes much of the ILP (Instruction Level Parallelism) existent in general purpose programs, i.e., SPECint. We should be able to obtain this ILP without requiring program recompilation so as to make our work as generally applicable as possible, though similar IPC should be obtained on non-legacy codes run on emerging ISA's, e.g., the VLIW and JIT (Just-In-Time) compilation worlds. Other goals

[1] This work was partially supported by the National Science Foundation through grants: MIP-9708183, DUE-9751215, EIA-9729839; by the URI Office of the Provost; by an equipment grant from the Champlin Foundations; by donations from Mentor Graphics Corporation and Xilinx Corporation; and by the Spanish Ministry of Education. Patents applied for. A version of this work has been submitted for publication.

include realizable cost and the ability of the machine's cost to scale linearly with processing capabilities, e.g., ALU's.

1.1 The Classic ILP Problems

It has been known for decades[9, 13, 20] that there exists a significant amount of ILP in programs. Current high-end superscalar processors do take advantage of the ILP to have many instructions in flight at a given time, but the net resulting IPC (Instructions Per Cycle) is always low (in the low single digits).

There are several contributing factors to these disappointing results: many processors do not take advantage of the most sophisticated ILP techniques, e.g., only using simple *single-path* branch speculation[21] to reduce the effects of unpredictable control flow; until recently, chip transistor budgets were not great enough to allow sophisticated ILP techniques to be realized; even with more transistors available, the existing hardware ILP extraction methods do not scale well either with increasing numbers of processing elements or an increasing number of in-flight instructions. Further, it is possible that the cycle time could be adversely affected with the more complex hardware; this concern has increased as processor clock rates have increased to close to 2 GHz (Intel Pentium 4) and beyond. Note that a reduced cycle time can be tolerated, marketing concerns aside, as long as the net performance or IPSecond increases. Power consumption is largely beyond the scope of this paper.

1.2 Motivation and Contributions

Lam and Wilson[9] conducted an extensive set of revealing simulations on some of the standard SPEC89 benchmarks assuming different types of branch effect (control flow) reduction techniques. Their most advanced model employed Single-Path branch speculation (same as simple branch prediction) with reduced Control Dependencies (instructions after a forward branch's target are independent of the branch) and Multiple Flows of control (multiple program counters); it was called: SP-CD-MF. For this model, a harmonic mean speedup of a factor of 40 was achieved on the integer benchmarks; unlimited execution resources were assumed. For an Oracle (i.e., a branch predictor that obtains 100% accuracy), a speedup of 158 was obtained. (For a limit study on scientific benchmarks, see: [8].)

Lam and Wilson demonstrated that large ILP exists in integer codes, but concluded that it was unlikely to be realized, particularly with high IPC, because of the machine limitations extant at the time. In particular, no commercial machine realized MF, and few realized CD, although such models had been created in research machines[18, 19]. We view the Lam and Wilson results as a challenge to produce a microarchitecture that will realize high IPC from the available high ILP. Our machine model goes beyond SP-CD-MF in both the data speculation and control speculation dimensions. Thus, we should be able to exploit ILP between 40 and 158, and possibly even beyond, since data speculation was not included in the Lam and Wilson study.

There are very stringent architectural and physical design constraints on any computer seeking to realize IPC in the 10's. Of course, 10's of Processing Elements (PE) or functional units are needed. This requires an enormous bandwidth to the register file or rename buffers, not to mention the complexity requirements placed on the classic reorder buffer. Secondly, both control and data dependency determination and/or enforcement can be either very complex (e.g., the dependency and domain matrices of Uht and Sindagi[20]), or have limited performance (e.g., the standard Tomasulo algorithm[12, 16]). Thirdly, these and other characteristics of classical techniques have scalability issues; both the Tomasulo and the Uht and Sindagi hardware grow in

size as the square of the number of PE's or the size of the instruction window, or both. If we ever intend to realize IPC in the 10's, we need hardware that will scale linearly. Lastly, as the time to cross a chip in cycles increases, we need designs that either utilize local communication or can tolerate multi-cycle latencies in the core of the hardware.

Our new Levo machine model addresses these concerns through the use of *registerless* design, local computation, and instruction *time tags*. First, in Levo there is no central register file, no central renaming buffers and no reorder buffer, although instructions are committed in order as necessary. What Levo does have is locally consistent register files distributed uniformly throughout the Levo execution window and among the PE's. The files' contents are likely to be globally inconsistent, but locally usable. Secondly, local computation is achievable because register values have short lifetimes[3], that is, a register value computed for one instruction is likely to be used (and overwritten) within a few later instructions. In Levo, PE's broadcast their results directly only to a small subset of the instructions in the Execution Window. Thirdly, by assigning small time tags to each instruction in the execution window, it is possible to enforce a time order among dependent instructions, as well as to use the correct value when an instruction has flow dependencies with two or more prior instructions. Also, since in Levo a time tag is assigned based on the instruction's position in the execution window, and not on an absolute overall dynamic instruction order, the time tags may be made very small, and they scale well.

As an example of Levo-style time tags, and of the overall structure of the Levo execution window core, consider Figure 1. In the Figure, each square corresponds to an instruction. In order to make the hardware simpler and facilitate locality, the window of static instructions is folded into an *n*-by-*m* matrix both logically and physically.

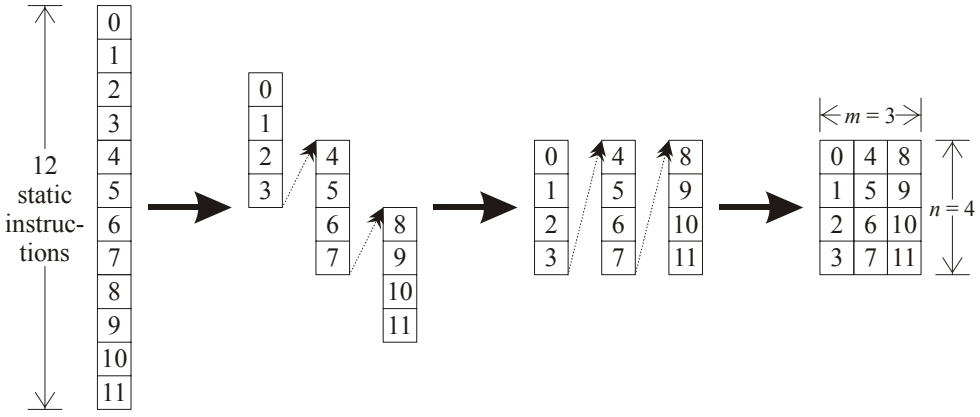

Figure 1 Example of mapping of static instructions to the Execution Window. Typically $n \geq 32$ and $m \geq 8$. The number in each box is the time tag value of the corresponding instruction.

1.3 Resource Flow Computing

Levo attempts to exploit as much speculation as possible; this is achieved via a *resource flow* computing model. With resource flow, PE's are assigned to the highest priority instructions that have not been executed, *regardless* of whether their inputs or operands are known to be correct (data flow constraints), and *regardless* of the necessity of execution of the instructions (control flow constraints). The rest of the execution time is spent applying programmatic data flow and control flow constraints to the instructions in Levo, so as to end with a programmatically-correct execution of the code. Levo executes standard *control flow* based programs using methods that

go beyond the standard control flow model, the *data flow* model[1] and the Superspeculative model[10].

The remainder of this paper is organized as follows. Section 2 reviews prior work related to Levo. In Section 3, we provide a detailed description of Levo's operation and microarchitectural features, focusing on their novel aspects. In Section 4 we both discuss our simulation methodology for quantifying the benefits of Levo, and present simulation results. We conclude in Section 5.

2 Related Work

2.1 Studies in Instruction Level Parallelism

Riseman and Foster[13], Lam and Wilson[9], and Uht and Sindagi[20] found much ILP in general purpose code. The latter study presented Disjoint Eager Execution (DEE) as a multipath way of potentially realizing IPC's in the 10's with constrained resources. (In the near future, Levo will include DEE, realizing higher degrees of IPC.) None of these studies made use of *data speculation*[11]. This may further improve the amount of ILP extractable[14], although no study has quantified how much of the ILP obtained via data speculation is also obtained by branch effect reduction methods.

2.2 Other Approaches

Probably the most successful high-IPC machine to date is Lipasti and Shen's *Superspeculative* architecture[10], achieving an IPC of about seven with realistic hardware assumptions. The Ultrascalar[4, 5] machine achieves "asymptotic" scalability, but only realizes a small amount of IPC, due to its conservative execution model. The Warp Engine[2, 6] uses time tags, like Levo, for a large amount of speculation; however their realization of time tags is cumbersome, utilizing floating point numbers and machine wide parameter updating. Pure data flow machines[1] were once promising, but did not achieve their goals. Limited data flow, such as the Tomasulo algorithm[16] used in current superscalar microarchitectures is more successful, but is still not aggressive enough: predication is not handled, SP-CD-MF is not handled and one or two broadcast buses for the ALU results is quite limiting for high IPC.

3 Levo Operation and Microarchitecture

A high-level diagram of the Levo machine is shown in Figure 2. Levo consists of three major components: the Memory Window, the Instruction Window and the Execution Window. For the purposes of this paper, the Memory Window is assumed to be a high-bandwidth interleaved memory system, e.g., 4-way low-ordered interleaved, with caches assigned on a per-bank basis, though a time-tagged memory is also under design. The basic operation is as follows. In the Instruction Window, the Instruction Fetch unit uses branch and other predictors, as well as special *Veiled Explicit Predication (VEP)* hardware, to buffer a large number of instructions from the memory system in the Instruction Load Buffer. The Execution Window is n rows by m columns of instructions; for our base system we use 32 rows and 8 columns. The Instruction Load Buffer typically loads all of the last logical column of the Execution Window at a time, e.g., 32 instructions. This is easier than might be thought, since the Execution Window is a static

instruction window, that is, the instructions nominally appear in the window in the same order they exist in memory, which is independent of the actual control flow of the executing program.

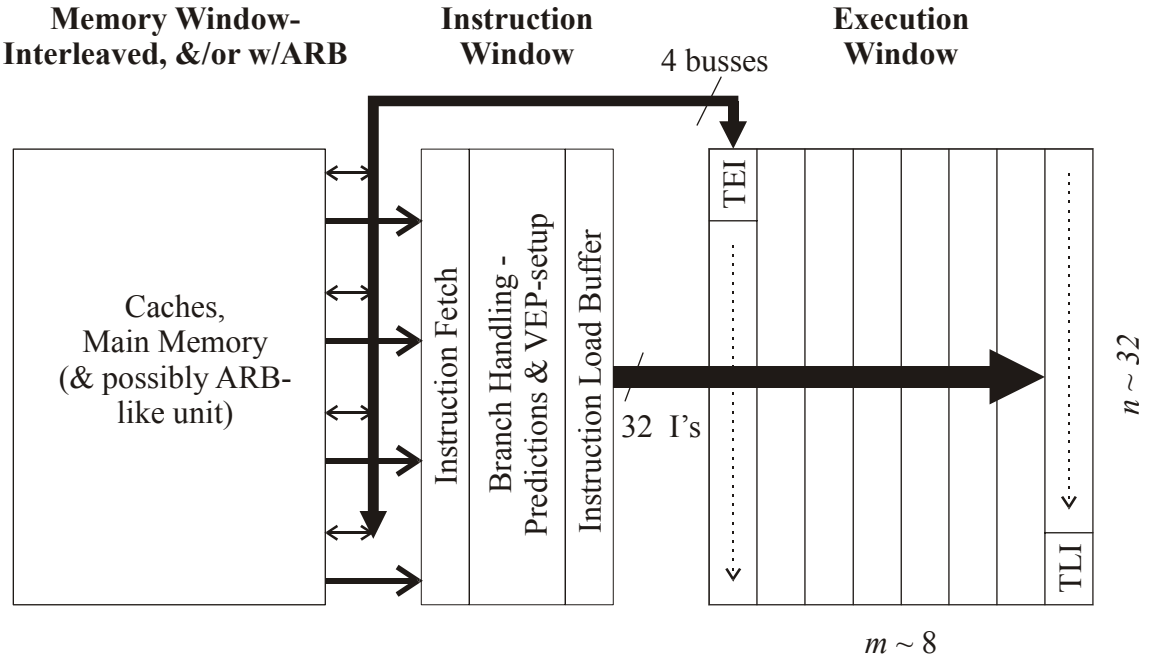

Figure 2 Levo block diagram. In order to sustain a high IPC, many instructions must be fetched and loaded into the Execution Window simultaneously. (TEI – Temporally Earliest Instruction; TLI – Temporally Latest Instruction)

As the left-most column (temporally earliest) finishes execution of all of its instructions, results are logically committed to storage. As the left-most column commits, the columns logically (but not physically) shift left, and the right-most column is filled with new instructions from the Instruction Load Buffer.

3.1 Execution Window

Most of the key novel elements of Levo appear in the Execution Window (see Figure 3): *time tags(TT)*, *active stations (AS)*, *sharing groups(SG)*, *register filter/forwarding units (RFU)* and *veiled explicit predication (VEP)*. Time tags enforce the nominal sequential order of the instructions to be executed, as well as enforcing data dependencies.

Levo's active stations are more intelligent versions of Tomasulo's classic reservation stations[16]. There is one instruction per active station; unlike a reservation station, an active station is not initialized with renaming or other dynamic information. Active stations are able to snoop and snarf data from buses with the help of the time tags, as well as providing predication (branch) support and redundant execution elimination when a new operand has the same value as the old operand.

Sharing groups associate some number of AS's with processing resources, e.g., one or more Processing Elements (PE), able to execute any instruction in the target ISA, including Floating Point operations; current and future transistor densities support this. PE's can only execute instructions from their sharing group, simplifying interconnections.

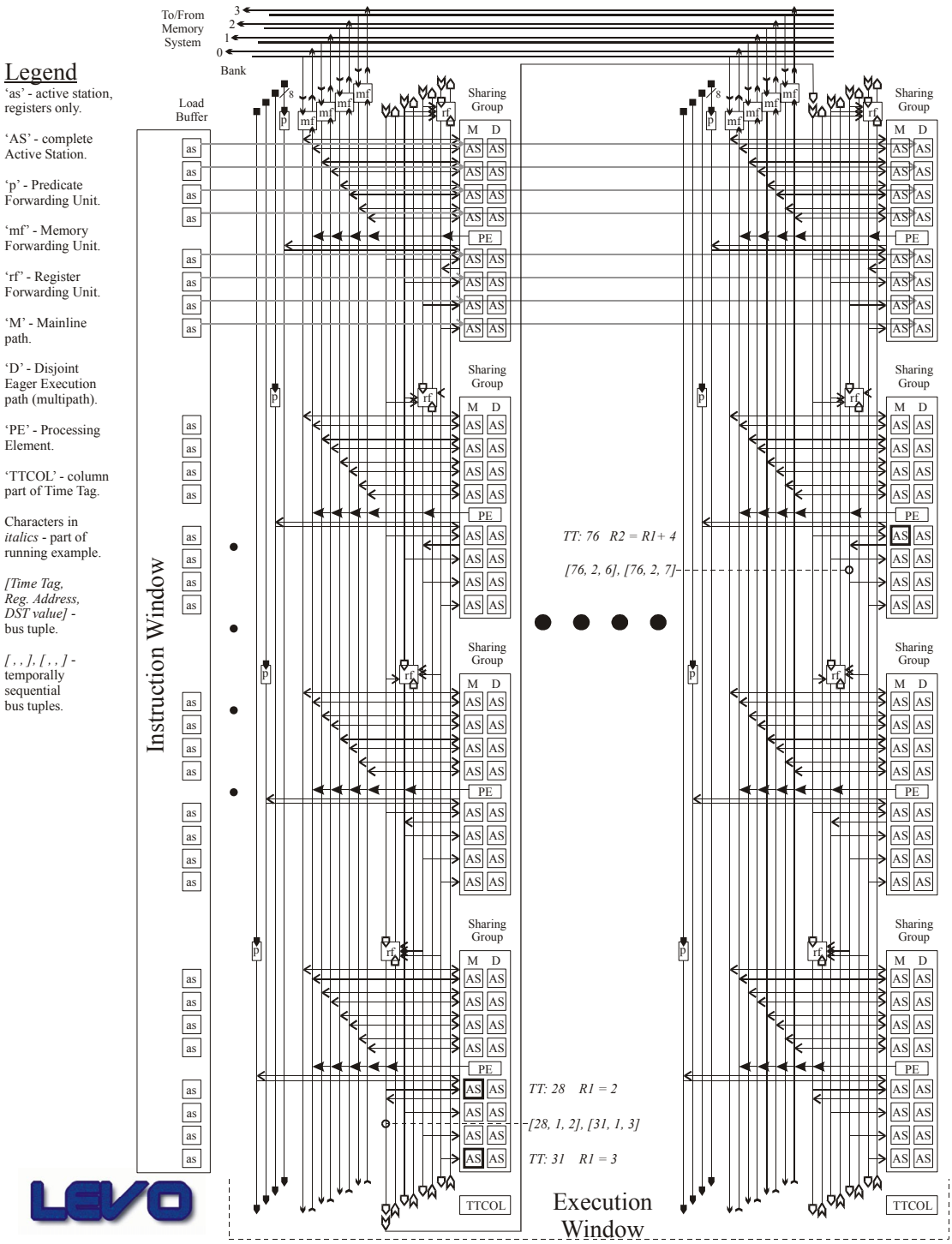

Figure 3 Levo Execution Window details. Two of possibly very many (*m*) columns shown. Number of rows (*n*) set to 32 and other dimensions set to specific values only for sake of illustration. A running example of operand broadcasting and snarfing is shown with *italics:* see the Legend. The example continues in Figure 4 and Section 3.1.6. The DEE AS's are not included in the simulation studies presented herein, but are left for future work. The point is, we know how to realize DEE in Levo.

RFU's separate the buses into scalable segments; an RFU takes results coming from a temporally earlier AS, and broadcasts them as necessary on temporally later buses, one or more cycles later.

VEP realizes full predication (executing branches only via predicates and not using the Program Counter) within the execution window without the intervention of the compiler or assembly language programmer. Without loss of ISA generality, we assume a MIPS-like ISA consisting of register operations, branches and memory operations.

3.1.1 Time Tags (TT)

The key operational feature of Levo is to allow instruction execution whenever possible. This implies the need to temporally sequentialize data flow dependent instructions, or rather ensure that the end result is the same as program ordered execution.

The hardware must have two abilities in order to realize a data flow execution of instructions without *a priori* or machine-wide dependency calculations. The first, commonly-known, ability is to compare register addresses between instruction operands and result data on one or more common busses. In Levo, this matching is done by comparators within the Active Stations.

Secondly, *time tags* enforce a temporal order on operand snarfing, and hence instruction execution. Only the closest result value (on the bus) before an instruction is used as the final value of the instruction's operand (having the same register address as the bus result value). Each time tag corresponds to exactly one instruction, and hence exactly one Active Station.

Time tags are logically divided into two parts: a column tag concatenated with a row tag. The row time tag of an instruction never changes once the instruction is loaded into the Execution Window. However, the column time tags are all decremented by one whenever the left-most or earliest column (with column time tag 0) commits and the Execution Window logically shifts left. That is, the Execution Window is logically a circularly shifting set of columns; however no shifting ever physically takes place, only the columns are renamed by decrementing their column time tags. The required range of time tag values is roughly equal to the number of instructions in the Execution Window (256 in our base configuration, implying an 8-bit time tag). One other benefit of the column renaming is the reduced power consumption resulting from not actually shifting the Execution Window.

3.1.2 Register Filter Units (RFU) and Spanning Busses

So far we have described a base machine with 256 AS's all connected together with some small number of spanning busses. In effect, so far there is little difference between a spanning bus and Tomasulo's Common Data Bus. This microarchitecture may reduce the number of cycles needed to execute a program via resource flow, but having the busses go everywhere will increase the cycle time unacceptably. Further, the machine is no where near being scalable.

Our solutions to these problems come from Franklin and Sohi's[3] observation that register lifetimes are typically quite short, 32 instructions at the high end. Based on this important observation in particular, we partition the busses into smaller segments, limiting the number of AS's/sharing groups any spanning bus is connected to; this has been set to 32 AS's, or four sharing groups, in our base configuration.

Each spanning bus starts at a sharing group to further simplify the hardware. For now, we assume only one bus originates at each sharing group, though this is not essential. Each spanning bus is one column's worth or four sharing groups long; each bus may be in multiple columns of the Execution Window (i.e., span column boundaries). With the above assumptions, any typical

sharing group sees four busses, including the one it originates. Each sharing group's AS results are sent only to the bus originating at that sharing group.

At this point in the design, we have made the common case fast: a generated value at a sharing group is most likely consumed before it reaches the end of a bus. However, logically Levo must be able to handle intermediate and extreme cases, e.g., where the value at AS 0 is used by AS 255 (the last AS).

Therefore, we connect each terminating bus to its adjacent originating bus with a *Register Filter/Forwarding Unit (RFU)*. An RFU takes a register value from the preceding bus segment, stores the value in the RFU's local register file, then broadcasts it on the following bus, competing with later sharing groups for the bus. Thus, there is a one or more cycle delay for register values going from one bus segment to the next.

Over the entire execution window, the total possible delay of a register value is great (proportional to the number of columns divided by the spanning distance). However, the likelihood that the value is used at the end of the window is small. Further, it is likely that the corresponding register address will be reused for a different value by the end of the last column. Lastly, as columns commit, the initially later columns may pass the register values flowing the other way.

Therefore, the spanning distance is held constant as the number of AS's or sharing groups or PE's increases or decreases. This gives us scalability.

3.1.3 Backwarding Spanning Buses

In certain cases it is necessary for AS's to request an operand from an earlier RFU. For example, an instruction that is just loaded into the Execution Window and has R1 as an operand must request R1's value from a previous RFU, because there is no guarantee that R1 will be broadcast (it may never be a destination in the current Execution Window).

In order to allow such requests, *backwarding busses* are added from the sharing groups to the RFU's. These busses daisy chain the RFU's in the backwards direction, mirroring the RFU forwarding busses. The first RFU that gets a request on a backwarding bus that has a valid value for R1, say, will satisfy the request by broadcasting R1 forwards in the usual way. In this case the backwarding request is terminated at the R1-sourcing RFU.

3.1.4 Active Stations (AS)

Active stations comprise the distributed intelligence and control logic necessary for Levo to correctly route data and predicates to instructions, without any prior setup or data dependency initialization necessary. The AS's distributed nature and their local communications help to ensure a small cycle time.

A basic block diagram of an Active Station is shown in Figure 4. Each of the register spanning busses is similar to a Tomasulo Common Data Bus, except that in Levo the time tag of the register data is also broadcast on the bus. Each AS Source (SRC1 and SRC2) snoops all of the register spanning busses to see if a relevant datum is being broadcast. If a datum is relevant to one of the sources, the source snarfs the datum and its time tag off of the bus. Unlike Tomasulo, the AS is speculative; in an AS, any one source receiving a new datum will cause the AS's instruction to fire, that is, both sources are sent to the PE with the instruction opcode for execution. The result is sent to the Destination block (DST), which in turn broadcasts the result, result address and AS time tag on a unique spanning bus. And then the process repeats.

The detailed logic of the AS's SRC and DST blocks is shown in Figure 5 and Figure 6, respectively. The caption of Figure 5 describes the detailed conditions necessary for the SRC block to both snarf data and fire the parent AS's instruction. The two novel parts of this operation are the time tag comparisons and the SRC data change determination. These ensure that only the closest previous datum is snarfed, and that the instruction only fires if the input datum has changed value.

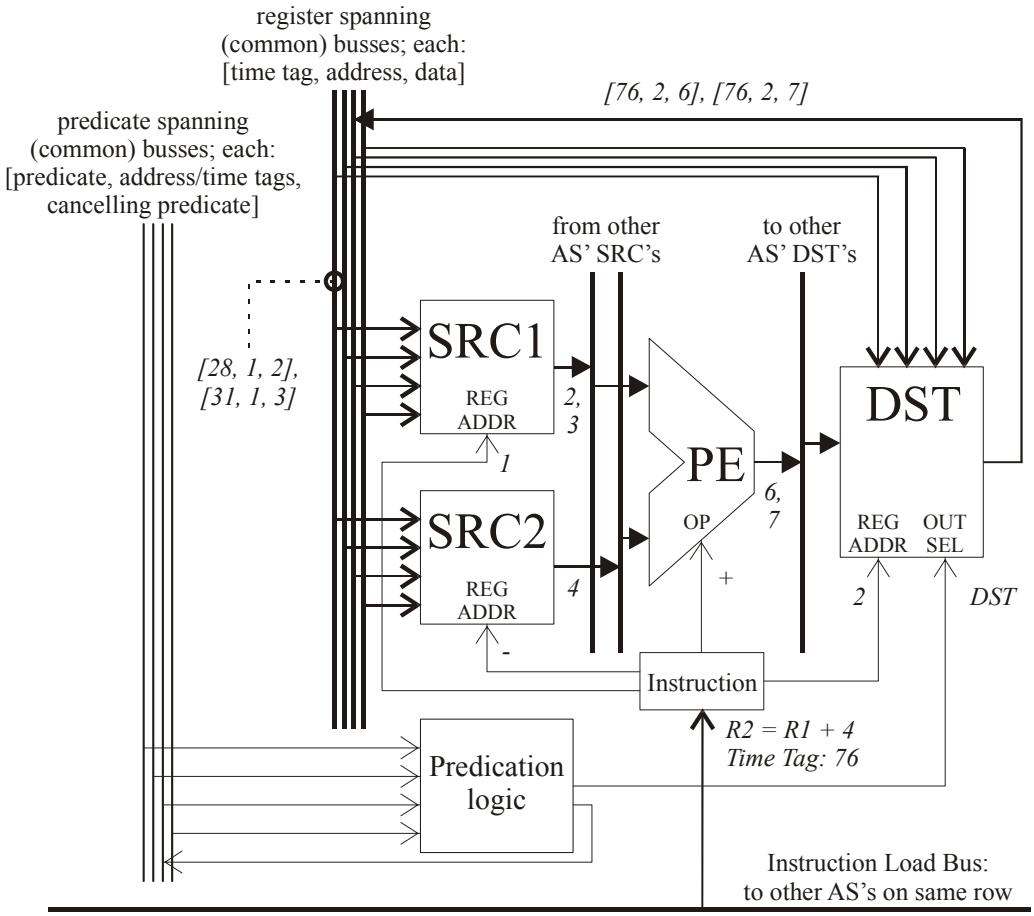

Figure 4 Active Station (AS) Functional Block Diagram, including common PE (Processing Element). There are typically 4 to 16 AS's per PE; the PE may be pipelined. Four spanning busses are shown for both register data and predication data; these are typical dimensions and the two may not be the same. The spanning busses logically go to the other AS's. All register instructions are of the form: DST = SRC1 op SRC2 . Running example information is in *italics* (see Figure 3 and Section 3.1.6).

The logic of the DST block is shown in Figure 6. Normally, when the instruction's predicate is true, that is the instruction is NOT branched around, and its result is needed, the output of the common PE is loaded into the DST register and then broadcast on a particular spanning bus. If the predicate is false, the result of the instruction is not needed, but it is necessary to ensure that later instructions have the correct value of the instruction's ISA result register. The relay register holds this value, having snarfed any previous writes to the same register address. Therefore, the value of the relay register is broadcast instead if the instruction's predicate is false.

3.1.5 Sharing Groups (SG)

In typical processors it is common for all of the instructions in the window to be able to access any of the processing resources, although, of course, not at the same time. This creates wire length, loading and prioritization logic complexity problems. There have been partial solutions to this problem employed by advanced processors such as the Alpha 21264; the processing resources are replicated and divided among a couple of sections of the machine.

In Levo several AS's, e.g., 4-16, share the same PE. The PE and AS's together are called a *sharing group*. Normally the AS's in a sharing group are adjacent members of the same Execution Window column.

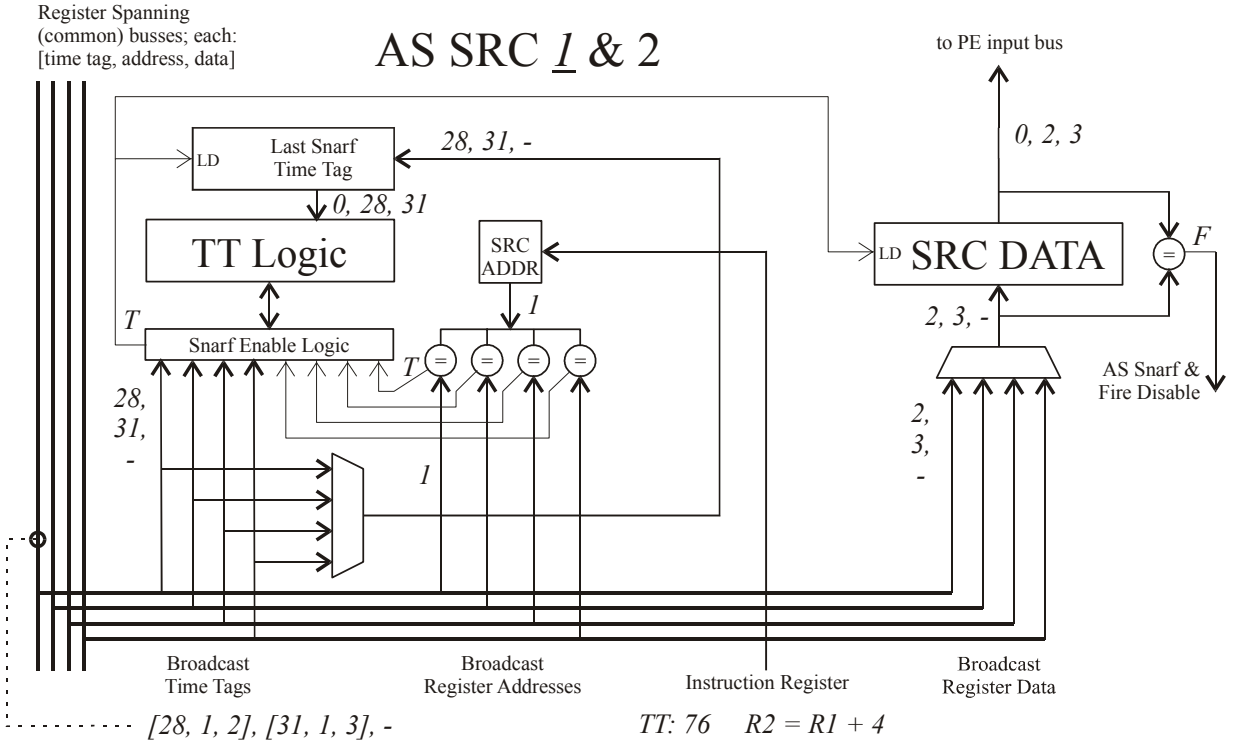

Figure 5 AS Source Operand Logic. Broadcast data is snarfed and the instruction is fired when its register address is the same as the SRC ADDR, and the broadcast time tag is less than this AS's time tag (implicit), and the broadcast time tag is greater than or equal to the time tag of the previously (Last) snarfed data, and the broadcast data differs from the current SRC DATA. Four spanning busses assumed (typical). Running example information in *italics* (see Figure 3 and Figure 4).

3.1.6 Levo Example – Register Operations

We show an example of Levo's operation primarily with respect to register operations in parts of Figure 3 through Figure 6. The specific text of the figures corresponding to the example is in *italics*. The example illustrates snooping, snarfing, and uses of the "last snarfed Time Tag" to enforce correct data flow.

Basically, we show the results of the execution of two simple instructions in the first column, having the same result register, on a later instruction in the second column, having the latter result register R1 as an operand. The instruction with time tag 28 executes first, broadcasting its

result value on one of the register spanning busses. The instruction with time tag 76 appropriately snarfs this value, its time tag meeting the standard constraints; this instruction then fires immediately, broadcasting its own result on a later spanning bus. When the instruction with time tag 31 executes later, its result value is also snarfed by instruction 76 since 31's time tag is greater than that of the previously snarfed value, and instruction 76 fires again. Thus, correct true data dependencies are realized.

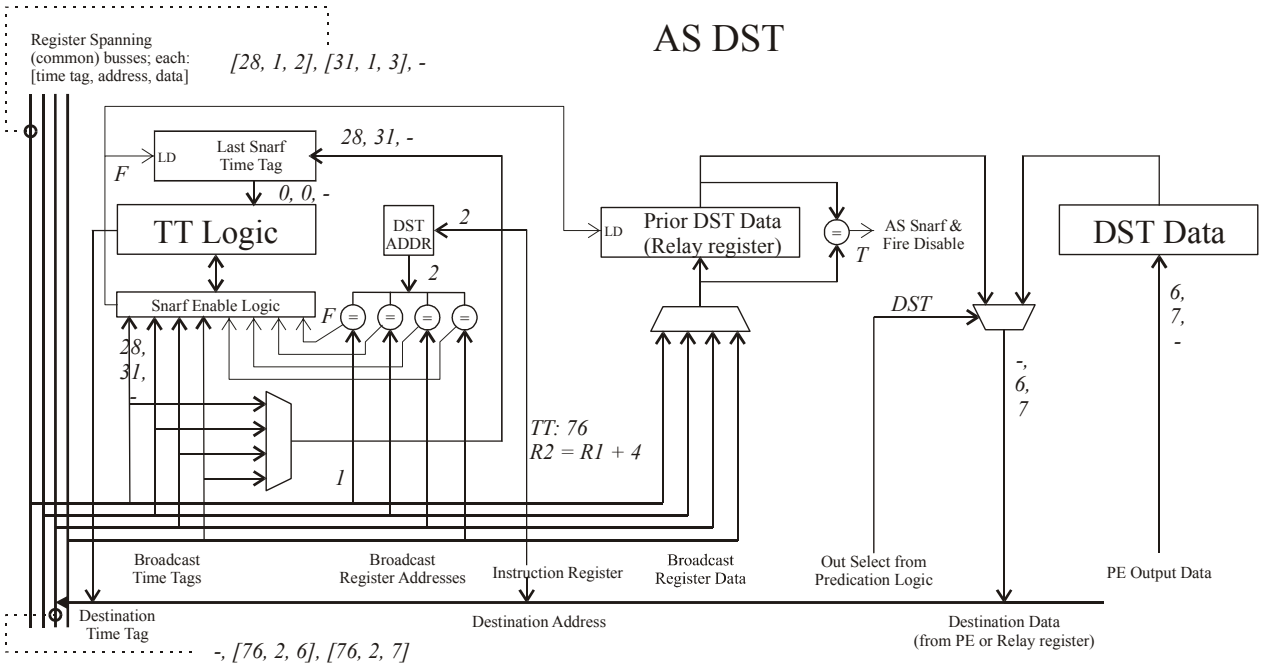

Figure 6 Active Station Destination (Result) Logic. If the instruction's predicate is true, the instruction's computed result is needed, and the result will come from the PE and be broadcast on a spanning bus with its address and time tag. If the predicate is false, the instruction's result is the closest value of the register data prior to this instruction, held in the relay register. This mechanism ensures that if the instruction is branched around, its own PE-produced data will not be used for later instructions. Running example information is in *italics* (see Figure 3 - Figure 5).

Note that if 31 had executed and broadcast before instruction 28, instruction 76 would have snarfed 31's result and fired. Later, when instruction 28 executed and broadcast its result, it would be ignored by instruction 76 (and it would not fire) since instruction 28's time tag is NOT greater than that held in 76's SRC1 "last snarf time tag" register (31).

3.1.7 Veiled Explicit Predication (VEP)

Our goal is to realize full *explicit predication* (with predicate bits) within the Execution Window transparent or *veiled* to the compiler, assembly language programmer, and ISA. We call our predication technique *Veiled Explicit Predication* (VEP). Within the Execution Window all branch executions are handled via predication, with the exception of branches with targets

outside of the Window. A simple example of VEP is given in Figure 7, though VEP can handle any mapping of branches and targets within the Execution Window.

The key to VEP is the consideration of branch *domains*[19]. A forward branch domain consists of the static instructions from the branch to its target, exclusive of the target and the branch itself. Backward branches also have domains, but Levo treats them specially, so we will not discuss them further here.

Every branch generates both a predicate p having the usual definition, and a *canceling predicate cp*, which is novel. Given that the branch's condition (true or false, taken or not taken, 1 or 0) is bc, then: $p = \overline{bc}$ and $cp = bc$.

Full predication is then achieved by ANDing in the branch predicate to the predicates of the instructions in the branch's domain, and ORing in the branch canceling predicate to instructions after the domain. Examining the logical implications of this statement, instructions within the domain have their predicates: $p_I = p = \overline{bc}$, which is clearly correct, while instructions after the domain have predicates: $p_I = p + cp = \overline{bc} + bc = 1$; that is, these instructions will execute regardless of the branch condition and are control independent[19, 21] of the branch, as expected.

address	Instruction	p_{in}	cp_{in}	p_{out}	cp_{out}	p_I
1	d = e + f	p_0	*0*	–	–	p_0
10	if *bc* goto 40	p_0	*0*	$p_{10} = \overline{bc}\, p_0$	$cp_{10} = bc\, p_0$	*1*
20	x = y + z	p_{10}	*0*	–	–	$\overline{bc}\, p_0$
30	u = v + w	p_{10}	*0*	–	–	$\overline{bc}\, p_0$
40	r = s + t	p_{10}	cp_{10}	$p_{40} = p_{10} + cp_{10}$ $= \overline{bc}\, p_0 + bc\, p_0 = p_0$	–	$= p_{out} = p_0$

p_{in} is effective input predicate for Instruction; its address is kept with Instruction.

cp_{in} is effective input cancelling predicate for Instruction; its address is kept with Instruction.

p_{out} is effective output predicate for Instruction. Is only set by branches and their targets. In general, for branches with branch condition *bc*, $p_{out} = \overline{bc}\, p_{in}$; for branch targets, $p_{out} = p_{in} + cp_{in}$. Address = Instruction's Time Tag.

cp_{out} is effective output cancelling predicate; only set by branches. In general, $cp_{out} = bc\, p_{in}$. Address = Instruction's Time Tag.

p_I is effective predicate for corresponding (same line) Instruction. For non-branch and non-branch targets, $p_I = p_{in}$. For branches, $p_I = 1$. (All branches are completely independent and may execute at any time.) For branch targets, $p_I = p_{out}$.

Figure 7 Example of VEP, once instructions are loaded into the Execution Window.

Two observations: with full predication every branch is directly data and control independent of all other branches (this is demonstrated theoretically in [19]). Next, although by construction

the predicates/canceling predicates are serially chained together, excessive execution time does not normally occur due to the typically limited propagation distance of predicate changes through the predication logic.

3.1.8 Data Memory Reference Handling

Data memory references are handled similarly to both register and predicate references. There are memory spanning buses, both forward and backward, and Memory Filtering/Forwarding Units (MFU) separating the partitioned spanning buses. Each MFU holds a small cache which holds values received from prior AS's on the forwarding bus. Unlike the RFU's, the addresses here are virtual memory addresses, and they are interleaved over a small number of spanning buses, typically corresponding to the Memory Window interleaved buses. This way, an AS knows exactly which memory spanning bus to use by looking at the least significant bits of the reference's address. TLB translation takes place in the Memory Window proper.

When an AS executes a Load instruction, it first computes the effective address of the reference, and then sends a request on a backwarding bus to get the latest value from an earlier MFU. Similarly, Stores broadcast their values forward to later instructions and MFU's.

In the case of a Store being disabled by a predicate evaluating false, instead of the relay register mechanism, a "nullify" message is sent forward with the address to squash dependent instructions and force them to request new values using the backwarding buses.

The modified data values in the MFU's closest to the Memory Window (at the top of column 0 of the Execution Window) are those that are actually committed to memory.

3.2 Instruction Window

The purposes of the Instruction Window are to keep the Execution Window filled with executable instructions as much as possible and as long as possible, while initializing or predicting register, memory data, predicate and canceling predicate values. These ends are realized by employing several novel and standard predictors, and other techniques. Details of the Instruction Window are beyond the scope of this paper and will be published later.

4 Simulation Methodology and Results

4.1 Levels of Abstraction

The FastLevo trace-driven simulator is the coarsest level of simulation used. FastLevo does not actually compute data values, though it does enforce data dependencies and faithfully models various hardware limitations such as sharing group size and bus bandwidth. The objective of FastLevo is to give a first-order estimate of the performance to be expected with Levo, without the use of more detailed execution-driven simulation.

The second level of abstraction we use in our work is the LevoSim execution-driven simulator, which consists of about 53K lines of C code that mimic the detailed functionality of Levo, down to the bus transaction level. All possible sources of added latency are included. Presently LevoSim executes unmodified MIPS R3000/R4000 ISA (RISC) code, although it runs on Sun SPARC machines. In order to verify functionality, instruction commitment and register and memory write traces from both LevoSim and a real MIPS processor are compared for correctness. All model dimensions are parameterized, so that the complete Levo machine design space can be examined.

A third level of modeling abstraction used in our work is done with a synthesizable VHDL model (HDLevo[22]) of key elements of Levo. We use this model to further check Levo's functionality, determine realistic hardware cost estimates, and start us on the path to physical prototype construction. The hardware models are written in VHDL text and synthesized for large Xilinx FPGAs.

4.2 Simulation Results

We now present preliminary results for five of the SPECint2000[15] benchmarks and Dhrystone, shown in Table 1; it also includes relevant characteristics of the benchmarks. The latter were all compiled on a MIPS processor with –O2 level optimization.

Table 1. Benchmark programs and their characteristics.

Benchmark (all SPECint2000, except *)	Type or Function of Benchmark	Input File (all std. except Bzip2-reduced)	Dynamic Instructions Simulated	Simulated Well Past Initialization[2]?
Bzip2	Compression	input.program	1,577,493	yes
Go	Game	5stone21.in	1,081,504	yes
Twolf	Circuit designer	ref	1,085,233	yes
Gcc	Compiler	integrate.I	1,129,776	no: 80% is init.
Vortex	Database	bendian1.raw	2,101,239	no: 69% is init.
*Dhrystone	Std. Synthetic	standard	65,819	yes: to completion

In Figure 8 we show the performance of the benchmarks as a function of the total hardware cost of the Execution Window, in terms of AS's. In order to simplify the simulations, perfect branch prediction was assumed; this is near what we would expect with DEE and predication. However, no data speculation was assumed, a conservative assumption. An RFU delay of 1 cycle was assumed. (While no memory latency was assumed for these results, other FastLevo simulations have demonstrated a high tolerance for memory delays in Levo.) These results are therefore a rough indication of what might be expected in a real Levo machine. Note the large gains made by Levo. For 512 AS's and more, all of the benchmarks showed an IPC of greater than 10. With 1024 AS's, a Harmonic Mean IPC of 15.3 is achieved.

We also ran the detailed LevoSim simulator on a small 4SG single column configuration with 1 AS per SG, running bzip2 for about 20000 instructions. It produced a CPI of about 7, matching the more general FastLevo results for the same configuration. Although this data point is for resources much less than the first FastLevo results shown above, it does help to validate the FastLevo results. Further, the LevoSim result demonstrates Levo's functionality.

4.3 VHDL Modeling and Logic Synthesis Results

The current version of HDLevo realizes and correctly simulates AS's (for registers and predication), RFU's, and register and predicate spanning busses. (The focus is on modeling the key novel parts of Levo in the Execution Window.) The model assumes a multi-column Execution Window.

[2] Although two benchmarks were simulated primarily in their initialization phase, the other benchmarks all exhibit high IPC as well, so initialization is not an issue here.

The synthesis of HDLevo produced the gate-equivalent counts for the Levo sections shown in Table 2. We find that a 32 by 8 AS Levo Execution Window, with DEE, but not counting PE's, is likely to use about 14.5 million gate-equivalents, including flip-flops, or 58.6 million transistors (using Xilinx's 2-input NAND gate-equivalent[23]). Even doubling this for better performance and adding in the cache's, PE's and Instruction Window's costs, this is certainly realizable with current custom VLSI technology: McKinley, the follow-on to Intel's Itanium, uses 240 million transistors[7]. We should also be able to build a Levo prototype with about 32 million-gate equivalent FPGA's in the Execution Window proper; most of these have already been donated by Xilinx. (PE's will be realized by force-feeding instructions to MIPS CPU chips.)

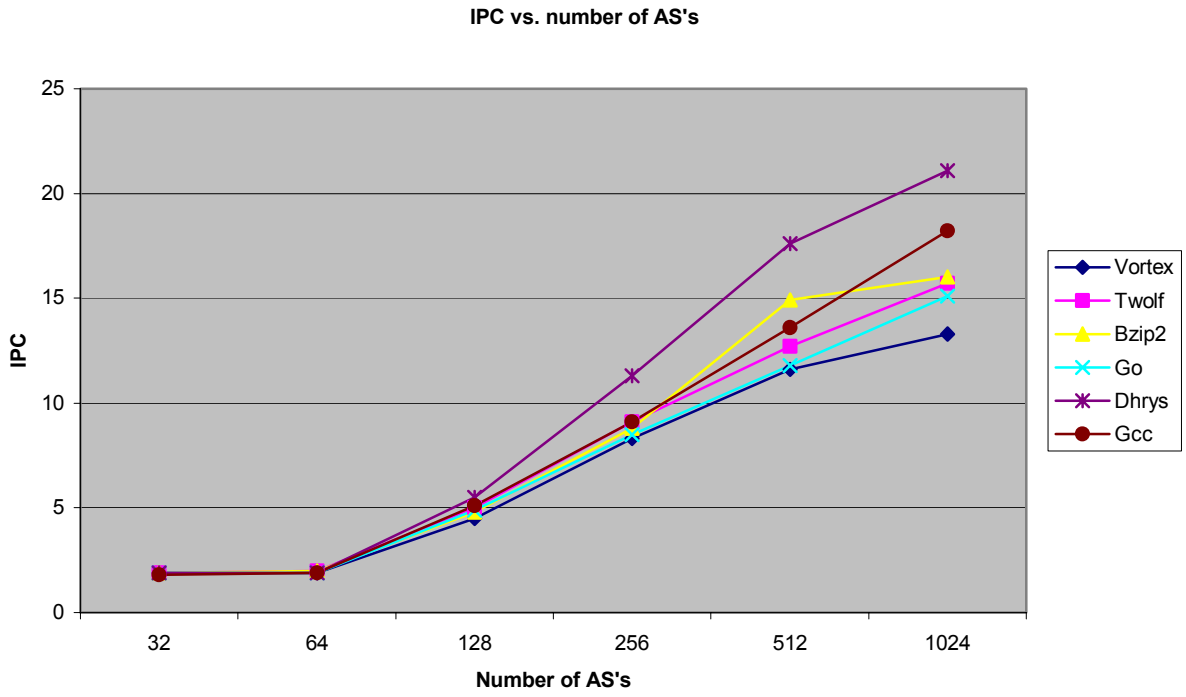

Figure 8 IPC vs. Total number of AS's in the simulated Levo. From FastLevo. 256 AS's corresponds to the 32 by 8 design point.

Table 2 Modeled hardware cost of Levo Execution Window, both in 2-input NAND gate-equivalents and transistors. 32 by 8 Execution Window assumed, with DEE. Eight AS's per SG, one PE per SG.

Levo Section	Gate-equivalents	Transistors
Per AS	21,621	86,484
All AS's (512)	11,069,952	44,279,808
Average RFU (4/column)	135,247	540,988
PFU (estimated) (4/column)	5,000	20,000
MFU (estimated) (4/column)	37,604	150,414
Total Execution Window (no PE's)	**14,475,880**	**58,614,922**

4.4 Discussion

We have demonstrated with the FastLevo constrained limit study that Levo will likely obtain high IPC. The IPC should be further improved with the use of a number of optimizations including rampant speculation[14, 17, 20].

5 Summary and Conclusions

IPC's in the 10's are achievable with scalable hardware.

Resource flow is a reasonable execution model. Registerless architecture and time tags work well.

Both the LevoSim and HDLevo results indicate that Levo works. Further, the logic synthesis results of HDLevo indicate that Levo is buildable, even with current technology.

Future work includes the addition of data speculation, DEE and predication to the simulation models, further exploration of the vast Levo design space, microarchitectural enhancements to Levo, and finally the detailed design and construction of a Levo Prototype. We should also be able to find out if there is overlap or synergy between ILP from branch effect reduction techniques and that from data effect reduction techniques.

References

[1] T. Agerwala and Arvind, "Data Flow Systems - Special Issue," *IEEE COMPUTER*, vol. 15, no. 2, 1982.

[2] J. G. Cleary, M. W. Pearson, and H. Kinawi, "The Architecture of an Optimistic CPU: The Warp Engine," in *Proceedings of the Hawaii International Conference on Systems Science (HICSS)*, vol. 1: University of Hawaii, January 1995, pp. 163-172.

[3] M. Franklin and G. S. Sohi, "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors," in *Proceedings of the Twenty-Fifth International Symposium on Microarchitecture (MICRO-25)*: IEEE and ACM, December 1992, pp. 236-245.

[4] D. S. Henry, B. C. Kuszmaul, G. H. Loh, and R. Sami, "Circuits for Wide-Window Superscalar Processors," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*. Vancouver, BC, Canada: IEEE and ACM, June 10-14, 2000, pp. 236-247.

[5] D. S. Henry, B. C. Kuszmaul, and V. Viswanath, "The Ultrascalar Processor: An Asymptotically Scalable Superscalar Microarchitecture," in *HIPC '98*, December 1998, URL: http://ee.yale.edu/papers/HIPC98-abstract.ps.gz.

[6] D. Jefferson, "Virtual Time," *Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404-425, July 1985.

[7] K. Krewell, "Intel 1Q01 Earnings Plummet," *Cahners Microprocessor*, vol. 15, no. 5, May 2001.

[8] M. Kumar, "Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications," *IEEE Transactions on Computers*, vol. 37, no. 9, pp. 1088-1098, September 1988.

[9] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*. Gold Coast, Australia: IEEE and ACM, May 1992, pp. 46-57.

[10] M. H. Lipasti and J. P. Shen, "Superspeculative Microarchitecture for Beyond AD 2000," *IEEE COMPUTER*, vol. 30, no. 9, pp. 59-66, September 1997.

[11] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," in *Proceedings of the Seventh Annual International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*. Boston, MA: IEEE and ACM, October 1996, pp. 138-147.

[12] Y. Patt, W. Hwu, and M. Shebanow, "HPS, a New Microarchitecture: Rationale and Introduction," in *Proceedings of the Eighteenth Annual Workshop on Microprogramming (MICRO-18)*: IEEE and ACM, December 1985, pp. 103-108.

[13] E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Transactions on Computers*, vol. C-21, no. 12, pp. 1405-1411, December 1972.

[14] Y. Sazeides, S. Vassiliadis, and J. E. Smith, "The Performance Potential of Data Dependence Speculation & Collapsing," in *Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29)*: IEEE and ACM, December 1996, pp. 238-247.

[15] SPEC, "Description of the SPECint2000 Benchmarks," URL: http://www.spec.org/osg/cpu2000/CINT2000/, created 1999, accessed: June 5, 2001.

[16] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25-33, January 1967.

[17] J. Tubella and A. Gonzalez, "Control Speculation in Multithreaded Processors through Dynamic Loop Detection," in *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA-4)*. Las Vegas: IEEE, February 1998.

[18] A. K. Uht, "An Efficient Hardware Algorithm to Extract Concurrency From General-Purpose Code," in *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, January 1986, pp. 41-50.

[19] A. K. Uht, "A Theory of Reduced and Minimal Procedural Dependencies," *IEEE Transactions on Computers*, vol. 40, no. 6, pp. 681-692, June 1991.

[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)*. Ann Arbor, MI, November/December 1995, pp. 313-325.

[21] A. K. Uht, V. Sindagi, and S. Somanathan, "Branch Effect Reduction Techniques," *IEEE COMPUTER*, vol. 30, no. 5, pp. 71-81, May 1997.

[22] T. Wenisch and A. K. Uht, "HDLevo - VHDL Modeling of Levo Processor Components," Department of Electrical and Computer Engineering, University of Rhode Island, Kingston, RI, Technical Report 072001-100, July 20, 2001.

[23] Xilinx Staff, "Gate Count Capacity Metrics for FPGAs," Xilinx Corp., San Jose, CA, Application Note XAPP 059 (V. 1.1), February 1, 1997, URL: http://www.xilinx.com/xapp/xapp059.pdf, accessed: June, 2001.