Levo - A Scalable Billion Transistor CPU With IPC in the 10's

Augustus K. Uht*, David Morano**, Alireza Khalafi** and David Kaeli**

*University of Rhode Island and **Northeastern University

Abstract

The Levo high ILP processor is presented, described and evaluated. Levo employs instruction time-tags *and* active stations *to ensure correct operation in a rampantly speculative and out-of-order* resource flow *execution model. The Tomasulo-algorithm-like broadcast buses are segmented; their length is relatively constant, that is, does not increase with machine size. Hence Levo is scalable. Known High-ILP techniques such as Disjoint Eager Execution and Minimal Control Dependencies are implemented in novel ways. Data value prediction is also employed. Examples of basic Levo operation are given. A chip floorplan for a particular version of Levo is presented, demonstrating feasibility and little cycle-time impact. The Levo model is also extensively simulated, showing both its likely near-term performance as well as its performance with resources available in the future. IPC in the 10's are shown.*

1 Introduction

We present Levo, a General-Purpose (GP) processor exhibiting IPC (Instructions Per Cycle) in the 10's with realistic hardware constraints and little detriment to cycle time. The basic operation model is *resource flow* execution: instructions execute as soon as their operands (speculative or otherwise) are free and a Processing Element (PE) is free. Thus, the resources, PEs, flow to the ready instructions.

Although embedded processors make up the vast majority of processors in the world, GP processors are still greatly of interest. As well as being important for desktop applications, GP Microprocessor technologies drive much of the advances in embedded systems.

GP processor performance is improved by improving either IPC or clock frequency, for a given instruction set. We seek to improve performance via high IPC with realizable hardware, scalable both as more ILP extraction techniques become available and as more transistors become available. We have a great opportunity: in a few years, billion transistor chips will be available.

Levo is not 100% efficient. In computer microarchitecture efficiency is in the eye of the beholder, e.g., consider a 32-bit bit-serial adder vs. a 32-bit parallel carry-lookahead adder. The former is much more efficient, but the costlier latter (or something better) is always used, since

This work was supported in part by the U.S. National Science Foundation under Grants Nos. MIP-9708183 and EIA-9729839, the University of Rhode Island Office of the Provost, the Xilinx Corp., the Mentor Graphics Corp. and the Spain Education Ministry. <<<???did I get that right, Dave K.?

the logic is affordable; this has always been the trend of microarchitecture. While power and energy consumption are also issues, these are not our research focus, and are not covered in depth herein.

In this paper we describe Levo and its operation. We provide detailed simulation results characterizing Levo over a large range of its possible geometries, as well as present evidence of Levo's large future potential performance, demonstrating IPC's in the 10's. The paper is organized as follows. In Section 2 we review major problems impeding high IPC realization. Section 3 provides the Levo logical description, and discusses Levo's solutions to the high IPC problems. Other issues are addressed in Section 4. Section 5 describes the physical operation of Levo and presents a possible Levo single chip floorplan. Section six explains our experimental methodology, while Section seven presents our simulation results, with discussions. We conclude in Section eight.

2 High IPC Problems

There are three major impediments to high IPC: high and/or unscalable hardware cost; degradation of (increase in) cycle time, negating IPC performance gains; and lack of high IPC extraction methods. Prior work has shown that there is much ILP (Instruction Level Parallelism) in typical GP code [9]. Large instruction windows are necessary to realize the ILP [1, 16]; the large windows greatly exacerbate the first two high-IPC impediments.

2.1 High Cost

In typical microprocessors, such as the Pentium P6 microarchitecture [13] and the Alpha EV8 processor [15], a large Reorder Buffer is needed to maintain the logical correctness of the executing program in the face of its out-of-order (OOO) execution. The cost of Reorder Buffers and other dependency checking/maintaining types of structures [18, 20] is large and does not scale linearly with the number of entries; the typical cost is $O(k^2)$ for dependency checking/enforcement, etc., where k is the size of the reorder buffer and/or instruction window.

2.2 Unscalable Microarchitecture

As chip feature sizes shrink the buses interconnecting the microarchitecture become electrically long (high RC time). These long buses, as well as the unscalable hardware mentioned in Section 2.1 lead to longer cycle times and hence reduced overall performance. Centralized resources such as architectural register files exacerbate the problem, as well as being awkward to use. They result in long bus delays and a prohibitively high number of ports [15]. The latter can increase the size of the register file substantially, further slowing the system.

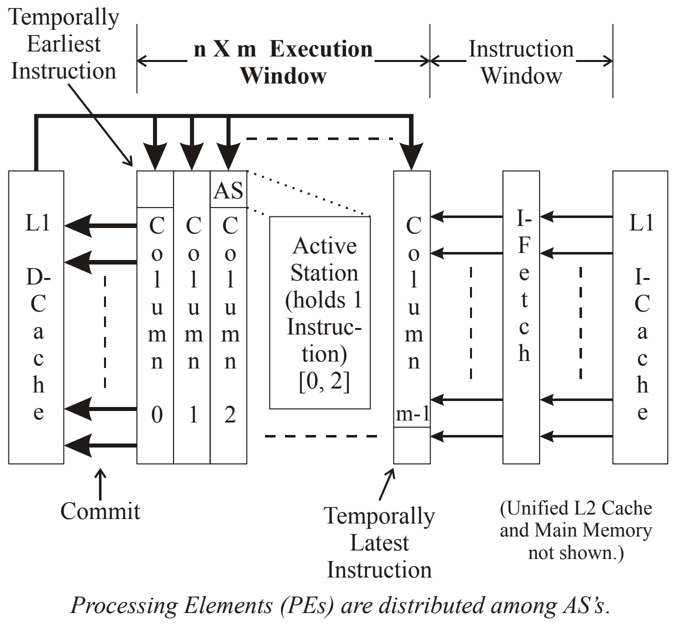
2.3 Low IPC

The high ILP promised over the years has not translated into high IPC or overall performance in a realistic processor. Even in those machines that did relatively well [2, 10], the overall performance is low. Part of the problem is that high-yield ILP methods and combinations of methods have not been attempted with realistic hardware. Combinations of methods can be particularly effective [22]; in the latter study both Disjoint Eager Execution (DEE) [21, 22] and Minimal Control Dependencies (MCD) [4, 18, 19] were found to be needed for very high ILP, assuming constrained resources. [6] demonstrated that data value speculation can be used with control speculation for large gains.

3 Levo High IPC Solutions and Description

Levo realizes high IPC with distributed and scalable hardware. A high-level logical block diagram of Levo is shown in Figure 1. The major novel part of Levo is the *n* X *m* Execution Window (E-window), holding *nm* Active Stations (AS). An Active Station is a more intelligent form of Tomasulo's reservation station [17]. Each AS holds one instruction. Small numbers of row adjacent AS's form Sharing Groups (SG). Processing Elements (PE) are assigned to each sharing group, typically one PE per SG. Each AS in the Execution Window has a corresponding *Time Tag (TT)* indicating its instruction's nominal temporal execution order. TT's are formed by the concatenation of the E-window column number and row number.

Levo presents a vanilla CPU picture to the user: any Instruction Set can be used, and no compiler modifications or assistance are necessary; these allow the execution of legacy code. Adding compiler optimizations is a subject for future work.



Temporally Earliest Instruction
n X m Execution Window
Instruction Window
L1 D-Cache
Column 0
Column 1
Column 2
AS
Active Station (holds 1 Instruction) [0, 2]
Column m-1
I-Fetch
L1 I-Cache
Commit
Temporally Latest Instruction
(Unified L2 Cache and Main Memory not shown.)
Processing Elements (PEs) are distributed among AS's.

Figure 1. Levo high-level logical block diagram. The major novel aspects of Levo are in the Execution Window.

3.1 Time Tags with Active Stations → Low Cost

Levo uses time-tagged active stations to realize speculative data-flow execution of code. No explicit renaming registers or reorder buffer are used. Active stations and this version of time-tagging are both novel.

The basic operation of time-tagged instructions is shown in Figure 2. Both classic renaming and time-tagging assume the broadcast of instruction result information on a bus, snooped by all reservation/active stations. (a) shows the basic code sequence considered and its outcome. Instruction 9 needs to use the closest previous value of R4 as its input. (b) shows the execution of the code assuming the use of renaming registers. Instruction 9 has been modified at instruction load time to source only the result of Instruction 5. Instruction 9 snarfs the result value of Instruction 5 when I9's operand register address equals the register address (4b) broadcast on the bus; Instruction 9 then executes.

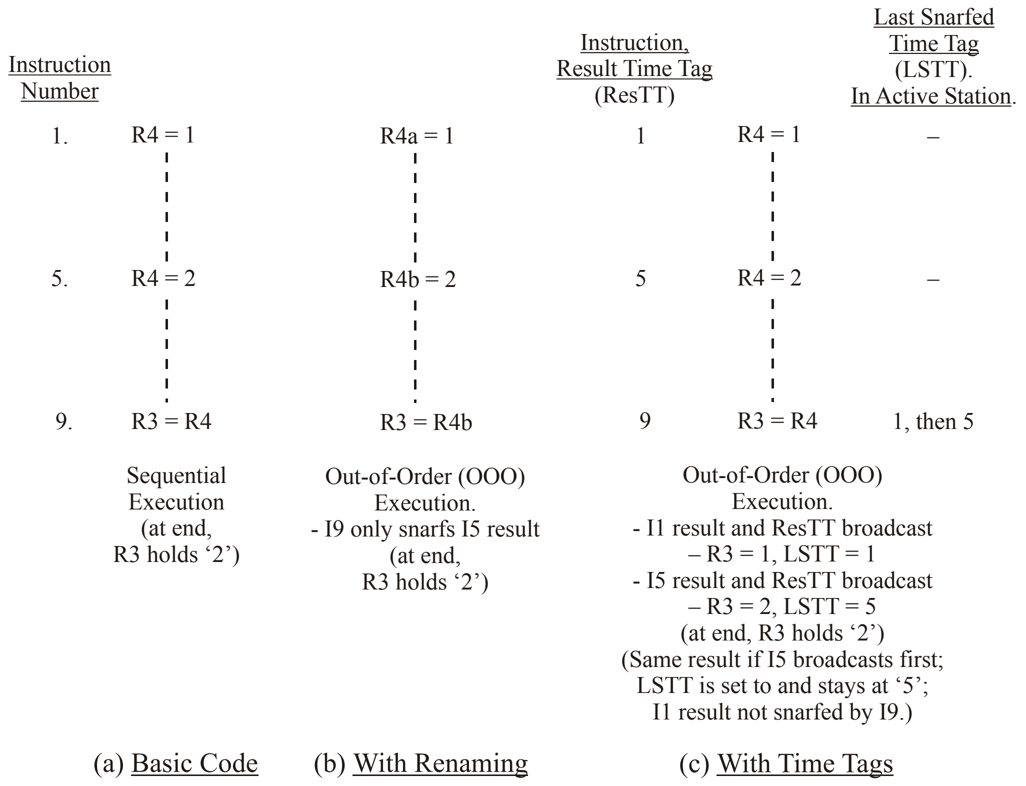

Instruction Number
Instruction, Result Time Tag (ResTT)
Last Snarfed Time Tag (LSTT). In Active Station.
1. R4 = 1
5. R4 = 2
9. R3 = R4
R4a = 1
R4b = 2
R3 = R4b
1 R4 = 1 –
5 R4 = 2 –
9 R3 = R4 1, then 5
Sequential Execution (at end, R3 holds '2')
Out-of-Order (OOO) Execution. - I9 only snarfs I5 result (at end, R3 holds '2')
Out-of-Order (OOO) Execution. - I1 result and ResTT broadcast – R3 = 1, LSTT = 1 - I5 result and ResTT broadcast – R3 = 2, LSTT = 5 (at end, R3 holds '2') (Same result if I5 broadcasts first; LSTT is set to and stays at '5'; I1 result not snarfed by I9.)
(a) Basic Code
(b) With Renaming
(c) With Time Tags

Figure 2. Time-tagged execution of code sample, with comparisons to other methods.

In (c), with time-tagging, no renaming is performed. Instead, each station now has a Last Snarfed Time Tag (LSTT) register and the basic operation is changed. When an instruction executes, it additionally broadcasts its time tag (in the example, this is just taken to be the instruction number). Snooping stations now also compare the broadcast result or instruction time tag (ResTT) with that held in the LSTT. If either the LSTT <= ResTT or the LSTT has not been loaded yet, and the register addresses match, then the result value is snarfed, the snarfing instruction is executed, and LSTT is updated by loading it with the value of ResTT. This ensures that only the closest previous version of an operand is used by an instruction at the end of execution. As is shown in the Figure, Instruction 9 ends with the correct value of R4 snarfed.

Thus, time-tagging has the large advantages of not requiring O(k^2) hardware for register renaming; time-tagging's cost grows linearly, O(k), with the number of instructions held in the execution window, and is thus scalable; and its execution algorithm is simple. This all combines in time-tagging having a low cost relative to large traditional superscalar processors' methods.

Time-tagging is a very general concept. In fact, Levo also uses it for memory instructions and operands as well as control-flow instructions (branches) and predicates. The only difference with memory operands is that the memory address is used instead of a register address for address matching purposes. Levo's hardware-generated predicates (described later) use the time tag as the predicate register address.

Figure 3 shows the detailed components of one operand of an Active Station. As shown there are three other necessary conditions for instruction execution: first, the operand must have changed value (this feature has been used by other researchers, e.g., [10]); secondly, and this is also a condition for snarfing, the broadcast result must be a member of the same path (predicted or not-predicted) as the station's instruction, in the case of multipath or DEE execution; and lastly, the operand must be from an instruction prior to the AS (ASTT > ResTT).

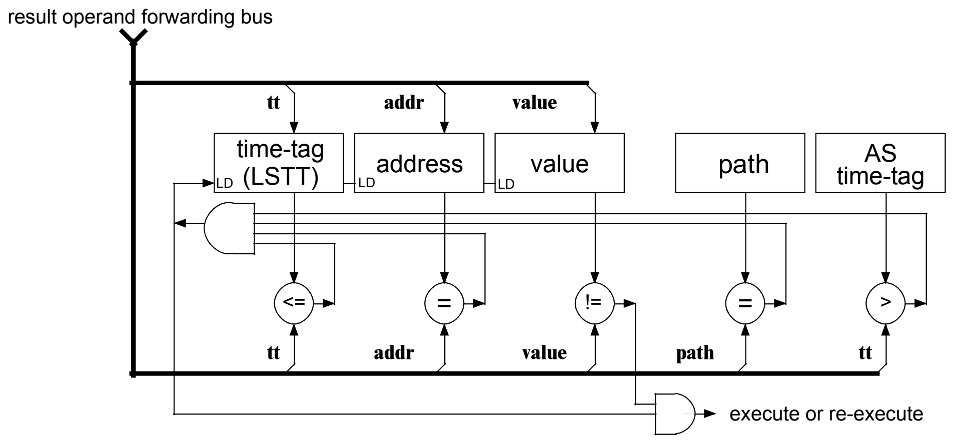


Figure 3. Levo Active Station (AS), showing comparison operations necessary for operand snarfing and instruction execution.

Time tags are used in [14] and other processors solely to squash instruction results occurring after a mispredicted branch. Time tags as used in Levo were originally proposed for simulation in the Virtual Time (Time Warp) model [8], and later in the Warp Engine [3] processor. The latter's implementation was relied on the use of floating point numbers for the time tags. In Levo, only the instructions' positions in the E-window are used for time tags, and hence are just small binary integers. Other high-ILP machines with different approaches include [7, 12].

3.2 Segmented Result Buses → Scalable Microarchitecture

In Levo segmented or *spanning* buses are used to propagate active station results to other, later, active stations. This is splitting Tomasulo's Common Data Bus. This is allowable without a large performance hit because an instruction's result is likely to be used soon after it has been created [5, 16]. Adjacent segments are connected via *Register Forwarding Units* (RFU) which introduce a small, say one cycle, delay from segment to segment; see Figure 4. The idea is that the later a result is used, the more likely it is to be used later in time, and the delays introduced by the RFU's will be transparent. Since the length of the segments need not change with the size of the machine, the spanning buses help realize scalability of the Levo microarchitecture.

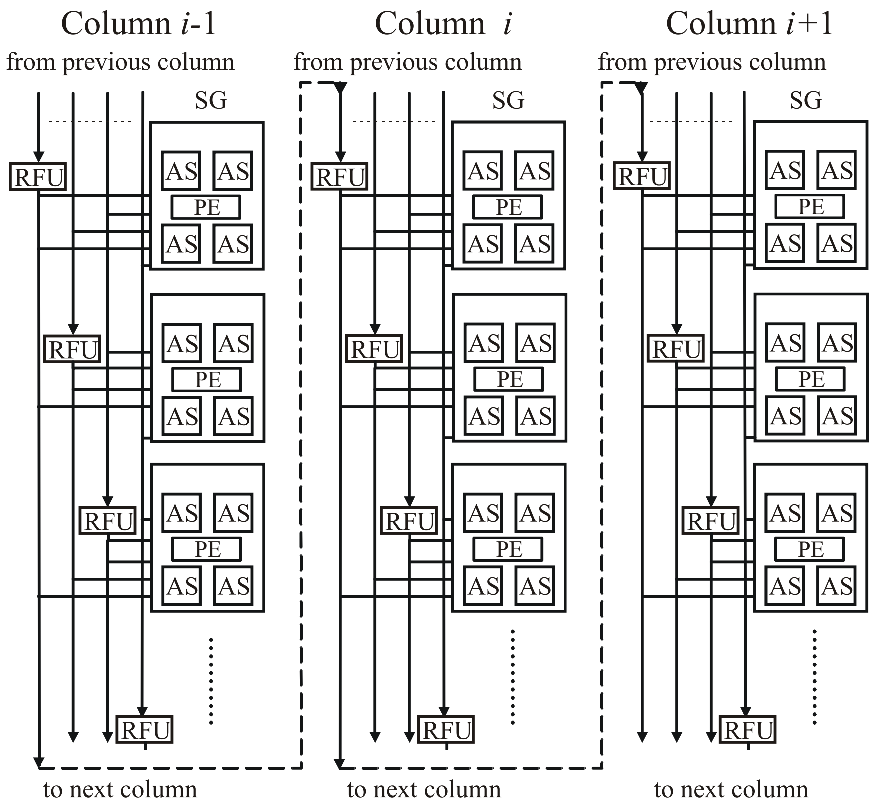


Figure 4. Segmented buses in generic Levo Execution Window. Note that the bus's length does not change as columns are added to the machine.

In Figure 4, note that nominally there is one RFU per sharing group and one spanning bus per RFU. Each RFU can maintain a unique copy of an architectural register. There are also Memory Forwarding Units (MFU), Predicate Forwarding Units (PFU), and spanning buses for each for the corresponding data (not shown in the figure). Since the number of ports to/from RFUs, MFUs and PFUs are constant with respect to the size of the machine, this also helps ensure scalability.

Another novel feature is the elimination of a centralized register file by using RFUs. To see this, assume in the figure that column i-1 is column 0. In operation, the instructions in column 0 stay there until they have all executed, at which time the entire column can be committed, and the remaining columns of instructions logically shifted left. Since the register values have already been broadcast to later RFUs in later columns it is unnecessary to save their state in a separate register file. This is also true for the predicate state. The memory values, however, must be written to the L1 D-cache.

Sometimes instructions must request operands from earlier in the E-window. This is done via *backwarding buses* (not shown), following the same paths as the forwarding buses, just going in the opposite direction.

3.3 ILP Enhancement Methods → High IPC

3.3.1 Hardware Predication

Full *hardware-based predication* is a new implementation of Minimal Control Dependencies. Briefly, MCD allows the execution of all branches concurrently, as well as allowing the execution of instructions after a branch's domain [19] independently of the branch. In prior work hardware-based methods required $O(k^2)$ hardware to realize MCD, k being the number of instructions in the E-window. In the new method the cost is $O(k)$.

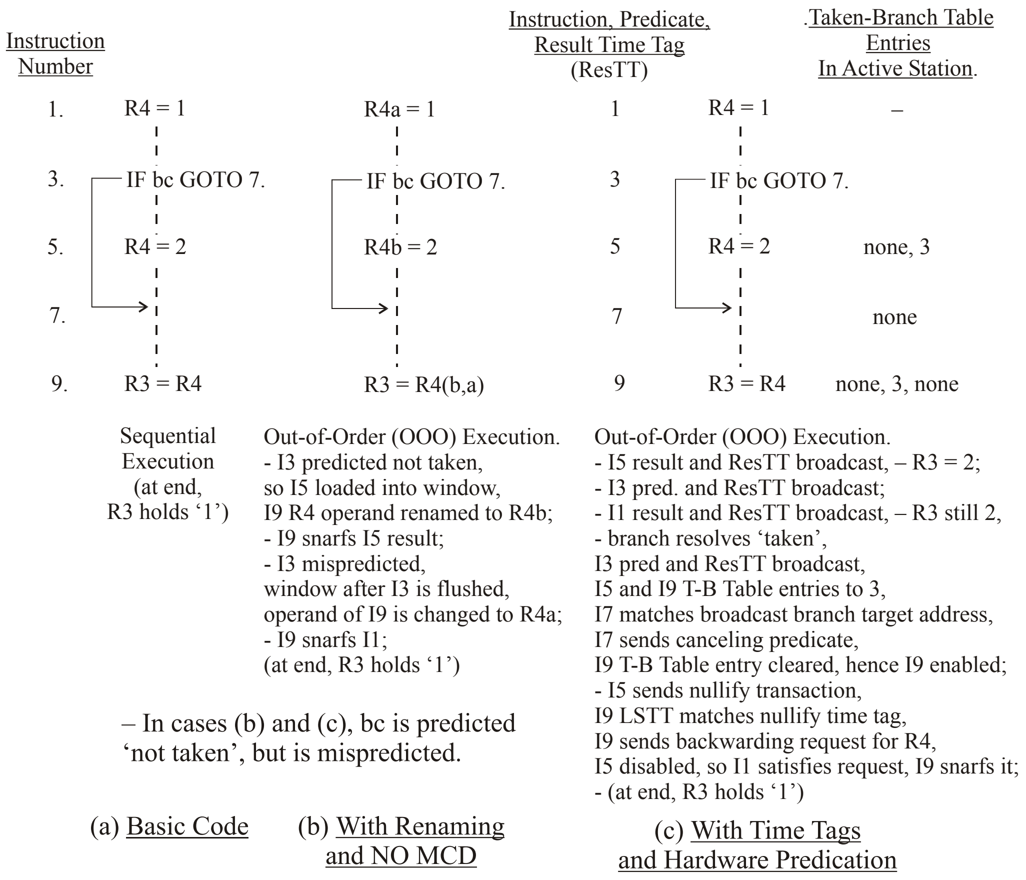
In this method predicates are assigned to all branches, predicted and evaluated solely with hardware, allowing the use of legacy code. Each active station has a predicate output associated with it, only valid if the instruction in the active station is a forward branch. Each active station also has a register for a possible branch target address, as well as a register holding the program address of the active station's instruction. Lastly, each active station has a *taken branch table*. Each entry of the table consists of a valid bit and a branch time tag; a branch's predicate is implicitly true (taken branch) if the branch has an entry in the table.

A simple example of hardware predication is shown in Figure 5, based on the example of Figure 2. The method works as follows. When a branch in the execution window executes, it broadcasts its target address (on a separate spanning bus), predicate value and time tag. Non-branch active stations following the branch, whose instruction addresses do not match the target address, snarf the predicate and its time tag. If the value of the predicate is true, the branch is taken and the time tag is entered in the stations' taken branch tables, with the corresponding valid bits asserted. As long as there is one or more entry in a table, the snarfing instruction is disabled and effectively branched around. If the table is empty, and the snarfed predicate is false, the branch is not taken and the instruction executes and broadcasts its result normally.

If there is a match between the broadcast target address and a following station's instruction address, then the instruction is just after the end of the branch's domain [19] and should thus be unaffected by the branch's execution. This station still snarfs the predicate and its time tag and then rebroadcasts them with the predicate changed to a *canceling predicate*. Later stations with a predicate address in their taken branch table matching the canceling predicate address have the table entry invalidated; thus, the corresponding branch no longer affects the operation of the station, the desired effect.

If a not taken branch is mispredicted, and thus becomes taken, the instructions within the branch's domain must be disabled; this happens as described above. But also, the now-disabled instructions within the domain that have already executed and broadcast their results must *nullify* these results and cause dependent instructions to re-execute. In order to do this, executed-now-disabled instructions broadcast a nullify transaction, containing the nullify indication, the instruction's time tag and the register address. Any later instruction with a matching operand register address and LSTT equal to the broadcast time tag (dependent instruction) sets itself to the unexecuted state, invalidates its LSTT, and sends a backwards request for the nullified operand. A prior instruction with a valid result, or an RFU, satisfies the request and execution resumes normally.

There are other nuances to the correct operation of hardware predication, including overflow of the taken-branch table. Space precludes their description here; please see [11] for more information.



Sequential Execution (at end, R3 holds ‘1’)

Out-of-Order (OOO) Execution.
- I3 predicted not taken, so I5 loaded into window, I9 R4 operand renamed to R4b;
- I9 snarfs I5 result;
- I3 mispredicted, window after I3 is flushed, operand of I9 is changed to R4a;
- I9 snarfs I1;
(at end, R3 holds ‘1’)

Out-of-Order (OOO) Execution.
- I5 result and ResTT broadcast, – R3 = 2;
- I3 pred. and ResTT broadcast;
- I1 result and ResTT broadcast, – R3 still 2,
- branch resolves ‘taken’, I3 pred and ResTT broadcast, I5 and I9 T-B Table entries to 3, I7 matches broadcast branch target address, I7 sends canceling predicate, I9 T-B Table entry cleared, hence I9 enabled;
- I5 sends nullify transaction, I9 LSTT matches nullify time tag, I9 sends backwarding request for R4, I5 disabled, so I1 satisfies request, I9 snarfs it;
- (at end, R3 holds ‘1’)

– In cases (b) and (c), bc is predicted ‘not taken’, but is mispredicted.

Figure 5. Example of Hardware Predication. Compared to both sequential execution and traditional superscalar (non-MCD) execution.

The cost of hardware predication is low, since most of the extra state storage only takes a few bits. More buses are needed, but not many more than already exist. In fact, it may be possible to share buses with other functions.

3.3.2 Disjoint Eager Execution (DEE)

DEE is described in [21, 22]. Briefly, it is an optimal form of speculative execution. DEE requires that only the most likely instructions, to be executed over all possible unexecuted instructions, be given priority for execution resources. In Levo likelihoods are not expressly calculated. Therefore this can be considered a form of multipath execution in which there is the predicted or *mainline* path (M) as well as typically several much shorter not-predicted or *disjoint* paths (D) spawning from the mainline path at some conditional branches.

DEE is realized in Levo by including AS’s solely dedicated to D-path execution in Sharing Groups; see Figure 6. The current model has as many D-path AS’s as M-path AS’s. In effect this means that each Levo execution window column is actually composed of two columns, one for part of the M-path and one for (part of) a D-path. The two columns share the execution, bus and other resources. Mainline AS’s always have priority for the resources. The cost impact of realizing DEE is relatively low: less than 10% greater cost (see Section 5) for a large performance improvement, typically ???%.

D-paths and M-paths execute concurrently, to greatly reduce branch misprediction penalties. Operationally, conditional branches are assigned a D-path (D'd) after they enter the E-window and as soon as a D-path is free. D-paths assume the D'd branch to have executed in the opposite sense as in the M-path. Other than that, the contents, predictions, etc. of instructions in corresponding M-path AS's and D-path AS's are the same. If the D'd branch is correctly predicted, the corresponding D-path state is thrown away and the D-path is reassigned to the next unresolved branch. If the D'd branch is mispredicted, the M-path is relabeled to be free D-paths, and its state is thrown away; the D-path is relabeled to be the M-path, and execution resumes. All told, it only takes a cycle to switch paths, and this can be overlapped with instructions' execution.

Considering the predication example of Figure 5 in the context of D-paths in Figure 6, assume the code is in M-path Column 0. Further assume that D-path 5 Column 0 is spawned from the branch at I3; AS's in the D-path above I3 hold the same state as the M-path above I3. Since I3 is predicted not taken, the M-path has the state with the result of I9 being R3=2. Simultaneously, the D-path version of I3 is set to taken, and the result of I9 is R3=1. Since the branch is mispredicted, once the branch resolves the M-path state is thrown away, the D-path becomes the new M-path Column 0, and R3 is at its final correct value of 1, all in one cycle.

3.3.3 Data Value Prediction

Levo realizes a type of data value prediction fashioned after the forms evaluated in [6]. There is one data value predictor per row of the execution window; they reside in the instruction window, and are used to predict data values just before a new column is loaded into the execution window. The parameters and other description of the predictor are left until Section ???, since it is evaluated with Levo for the first time in this paper. Cost is not great ??? Estimate, in transistors?

4 Other Issues and Levo Solutions

4.1 Instruction Window and I-Fetch

The basic instruction fetch mechanism is to obtain entire column(s) of instructions at once from the I-cache for loading into the E-window, once columns in the E-window commit and there is room for more instructions. The key here is that the instructions are fetched in the static or memory order; basically, the loading assumption is that they are not taken; they may still be predicted taken for execution purposes. This keeps the fetch mechanism simple and allows for high I-fetch bandwidth. It also keeps branch domains with their branches, so that MCD and DEE can be fully exploited and window flushes can be reduced.

This fetch mechanism is actually slightly modified in order to handle cases of large branch domains. Now instructions are fetched and loaded in the static order unless a branch is encountered that is predicted taken and whose target is far away (large domain), i.e., the distance is greater than some fraction of the execution window size.

4.2 Large Memory Latencies – Modified Memory System

The deep E-window in Levo results in a large tolerance to main memory latency. In prior work ??? we have shown that Main Memory latencies of 800 cycles or more can be tolerated without significant performance degradation. These latencies are typical of what is expected in the next few years. However, Levo is less tolerant of L1 Data Cache (D-cache) latencies, and thus requires a high-speed lower level of the memory system. (The L2 cache is also latency tolerant.)

Stores must be handled specially with respect to the L1 D-cache. The D-cache cannot be modified until the store is committed, that is, until its corresponding active station is in column 0 and the column has completely executed.

Store data is also held earlier in the E-window. Each MFU is chained to the next column's MFU, as with the RFUs. However, an MFU's internal structure is different. There is an L0 cache and a Previous Column Buffer (PCB). The PCB has *n* entries and holds the latest stores from the previous column. Each L0 entry also has an LSTT register associated with it, used in the same way as the LSTTs are in the active stations. The PCB memory contents are sent to the L1 D-cache when the prior column (0) has committed; they are then used to update the L1 D-cache state.

Load requests are handled with memory backwarding buses. Loads can be satisfied from either earlier active stations, earlier MFUs (via either a PCB or an L0 cache), the L1 D-cache, or higher up in the hierarchy. If the L1 cache satisfies the request, it does so by transmitting the data as a special store. The original load's L0 cache snarfs the store (as may any other L0 cache), sees that it is a special store, and then broadcasts it on its column bus (forced broadcast) for snarfing by the requesting load.

As will be seen in Section 5, the physical realization of the Levo memory system employs multiple copies of the L1 D-cache to keep the access time to the cache low (1 cycle) and to keep cache access bandwidth high. The multiple cache copies hold the same data, within a few cycles, with all of them replacing the same lines at the same time. One might think there is a problem with maintaining cache coherency, in that a nullified store transaction might arrive after the stale data has been saved in the cache, but it is solved by the construction and operation of the PCBs. The worst case is that a store nullify occurs from column 0 right after it has committed; but this cannot happen, since by definition column 0 cannot commit until all of its operations have finished. Regardless of the physical or electrical distance of column 0 from a particular cache copy, it is guaranteed that the nullify will arrive before the commit signal, and thus coherency is maintained. ???Dave M.: if this doesn't satisfy you, please explain why.???

<<<QUESTION to Dave M.: was Smith high latency paper published? Where? Does anyone have a reference for it???>>>

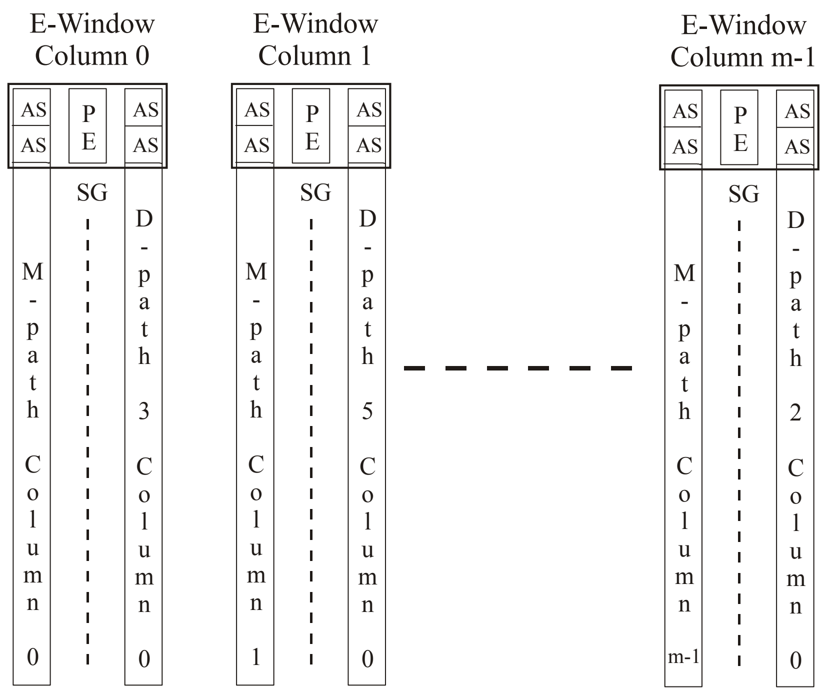


Figure 6. Arrangement of Mainline (M) path columns and DEE (D) path columns. Note that D-paths need not occupy the same execution window column as their corresponding M-path column. D-paths can also be multi-column.

5 Physical Considerations: Column Renaming and Chip Floorplan

As previously stated, logically the columns of the E-window shift left as the instructions in column 0 finish executing and column *m*-1 is loaded with new instructions. Physically shifting such an enormous amount of state is cumbersome.

Levo solves this by renaming the columns. Each physical column has one or more registers associated with it that hold its logical column number. When a logical left shift occurs, the logical column numbers of all of the columns are decremented. Recall that time tags throughout the machine are formed from the concatenation of the logical column number and the fixed row number of the corresponding active station; therefore, as left shifts occur the time tags are automatically corrected and regenerated.

The column renaming greatly simplifies the wiring and reduces the necessary power consumption associated with a physical shift.

A Levo chip floorplan for a specific geometry was constructed to demonstrate the realizability of Levo on a single chip within the next few years; the goal was not area optimization or exactness per se. The Compaq/Intel EV8 chip floorplan and dimensions [15] were used to size similar Levo structures, as well as to ensure that the critical path is not substantially increased by the Levo microarchitecture.

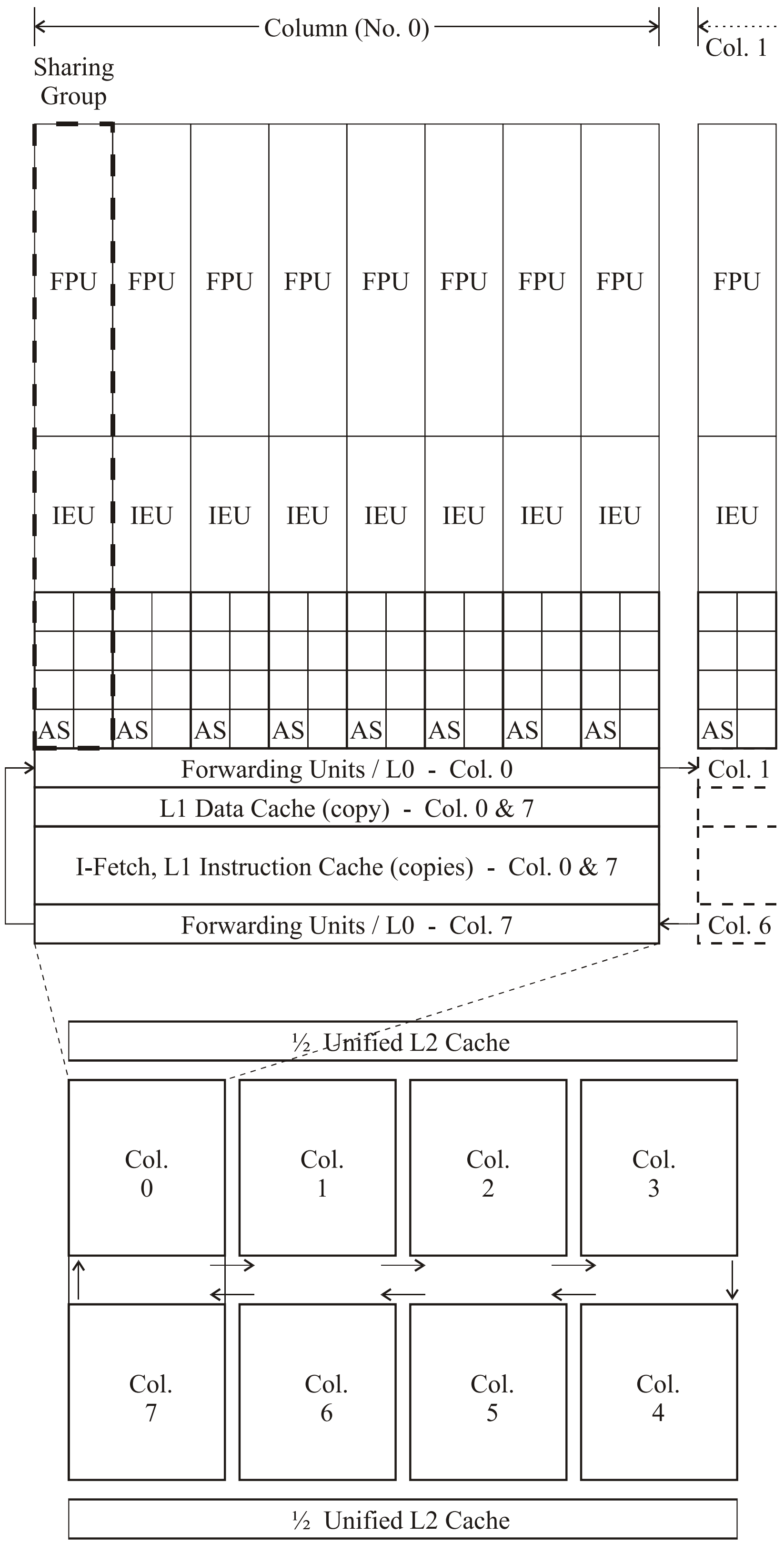

Column (No. 0)
Col. 1
Sharing Group
FPU
IEU
AS
Forwarding Units / L0 - Col. 0
L1 Data Cache (copy) - Col. 0 & 7
I-Fetch, L1 Instruction Cache (copies) - Col. 0 & 7
Forwarding Units / L0 - Col. 7
Col. 1
Col. 6
½ Unified L2 Cache
Col. 0
Col. 1
Col. 2
Col. 3
Col. 7
Col. 6
Col. 5
Col. 4
½ Unified L2 Cache

Figure 7. Possible Levo chip floorplan for a given geometry (but scalable upwards).

The geometry is: 8-4-8-8, that is, 8 sharing groups per column, 4 M-path and 4 D-path active stations per sharing group, 8 M-path columns and 8 D-path columns (8 E-window columns, total); see Figure 7. One FPU (Floating Point Unit) and one IEU (Integer Execution Unit) are assumed per sharing group. 64-bit data paths and machine architecture are also assumed.

In the floorplan the columns are physically oriented end-to-end, to keep the critical path length low. Every active station within a column is accessible from every other active station in the same column within one clock cycle. The delay from one forwarding unit to the next is one cycle or less. Assuming a target clock frequency of 10 GHz, possible within a few years, the realized clock frequency should be about 87% of this, that is, there should be a loss in performance from a reduced clock frequency of about 13%. This is offset much more by the IPC gain of Levo for the geometry considered: ???.

The Levo chip as described above is estimated to use about 500 million transistors; this is from both actual VHDL synthesis of key components [23] as well as rough estimates. The other 500 million, of a billion transistor chip, is free for other purposes, including I/O, the main memory interface, or more E-window columns.

Also note that Levo is easily scalable. In order to increase the machine size only pairs of columns need to be added to either end and inserted in the physical loop.

References

Technical Reports by this paper's authors are available from: http://www.ele.uri.edu/~uht

[1] T. M. Austin and G. S. Sohi, "Dynamic Dependency Analysis of Ordinary Programs," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*. Gold Coast, Australia: IEEE and ACM, May 1992, pp. 342-351.

[2] T. F. Chen, "Supporting Highly Speculative Execution via Adaptive Branch Trees," in *Proceedings of the 4th Annual International Symposium on High Performance Computer Architecture*: IEEE, January 1998, pp. 185-194.

[3] 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.

[4] J. Ferrante, K. Ottenstein, and J. Warren, "The Program Dependence Graph and its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319-349, July 1987.

[5] 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.

[6] J. Gonzalez and A. Gonzalez, "Limits on Instruction-Level Parallelism with Data Speculation," Department Architectura de Computadores, Universitat Polytechnica Catalan, Barcelona, Spain, Technical Report UPC-DAC-1997-34, 1997.

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

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

[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] D. Morano, "Execution-Time Instruction Predication," Dept. of Electrical and Computer Engineering, University of Rhode Island, Kingston, RI 02881, Technical 032002-0100, March 2002.
[12] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler, "A Design Space Evaluation of Grid Processor Architectures," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*. Austin, Texas USA: ACM, December 2001.
[13] D. B. Papworth, "Tuning the Pentium Pro Microarchitecture," *IEEE MICRO*, vol. 16, no. 2, pp. 8-15, April 1996.
[14] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman, "The Metaflow Architecture," *IEEE MICRO*, vol. 11, no. 3, June 1991.
[15] R. P. Preston, R. W. Badeau, D. W. Bailey, S. L. Bell, L. L. Biro, W. J. Bowhill, D. E. Dever, S. Felix, R. Gammack, V. Germini, M. K. Gowan, P. Gronowski, D. B. Jackson, S. Mehta, S. V. Morton, J. D. Pickholtz, M. H. Reilly, and M. J. Smith, "Design of an 8-wide Superscalar RISC Microprocessor with Simultaneous Multithreading," in *Proceedings of the International Solid State Circuits Conference*, January 2002. Slides from talk at conference also referenced.
[16] G. S. Sohi, S. Breach, and T. N. Vijaykumar, "Multiscalar Processors," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*: IEEE and ACM, June 1995.
[17] 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.
[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. Also in the tutorial ``Instruction-Level Parallel Processors", Torng, H.C., and Vassiliadis, S., Eds., IEEE Computer Society Press, 1995, pages 171-182.
[20] A. K. Uht, "Concurrency Extraction via Hardware Methods Executing the Static Instruction Stream," *IEEE Transactions on Computers*, vol. 41, no. 7, pp. 826-841, July 1992.
[21] A. K. Uht, "Disjoint Eager Execution: What It Is / What It Is Not," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 1, March 2002. URL: http://www.ele.uri.edu/~uht/papers/SIGArchCAN2002prepr.pdf.
[22] 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. URL: ftp://ele.uri.edu/pub/uht/micro95.ps.
[23] 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.