Levo - A Scalable Billion Transistor CPU With High IPC

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

*University of Rhode Island and **Northeastern University

Abstract

The Levo high ILP processor is 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 constant, that is, does not increase with machine size. This helps to make Levo scalable. Known High-ILP techniques such as Disjoint Eager Execution and Minimal Control Dependencies are implemented in novel ways. Examples of basic Levo operation are given. A chip floorplan of Levo is presented, demonstrating feasibility and little cycle-time impact. Levo is simulated, characterizing its basic geometry and its performance.*

1 Introduction

Levo is a General-Purpose (GP) processor exhibiting large IPC (Instructions Per Cycle) 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 acquired and a Processing Element (PE) is free.

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

While Levo does use many transistors, the trend has always been to use hardware less efficiently as chip transistor densities increased, e.g., all common GP microprocessor families. Power and energy consumption are also issues, but 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

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., and the Mentor Graphics Corp. David Kaeli is supported by the Ministry of Education, Culture and Sports of Spain. Patents applied for.

Levo and presents a possible Levo single chip floorplan. Section 6 gives our experimental methodology, while Section 7 presents our simulation results, with discussions. We conclude in Section 8.

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 [16]; the large windows greatly exacerbate the first two high-IPC impediments. A system is scalable if its cost grows linearly or less with an increase in Processing Elements or other key elements.

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 when it is executed out-of-order (OOO). The cost of Reorder Buffers and other dependency checking/maintaining types of structures [18] is large and does not scale with the number of entries; the typical cost is $O(k^2)$ for dependency checking/enforcement, etc., where k is the number of entries in the reorder buffer and/or instruction window.

2.2 Unscalable Microarchitecture

As chip feature sizes shrink buses 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. 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, such as [10], the overall performance is not great. 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 [19]; in the latter study both Disjoint Eager Execution (DEE) and Minimal Control Dependencies (MCD) [2, 18] were found to be needed for very high ILP.

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 E-window has a corresponding *Time Tag (TT)* indicating its instruction's nominal temporal execution order. TT's are formed by the concatenation of the AS's E-window column number and row number.

In Levo any instruction set can be used, and no compiler modifications are necessary; these allow the execution of legacy code. Adding Levo-specific compiler optimizations is a subject for future work.

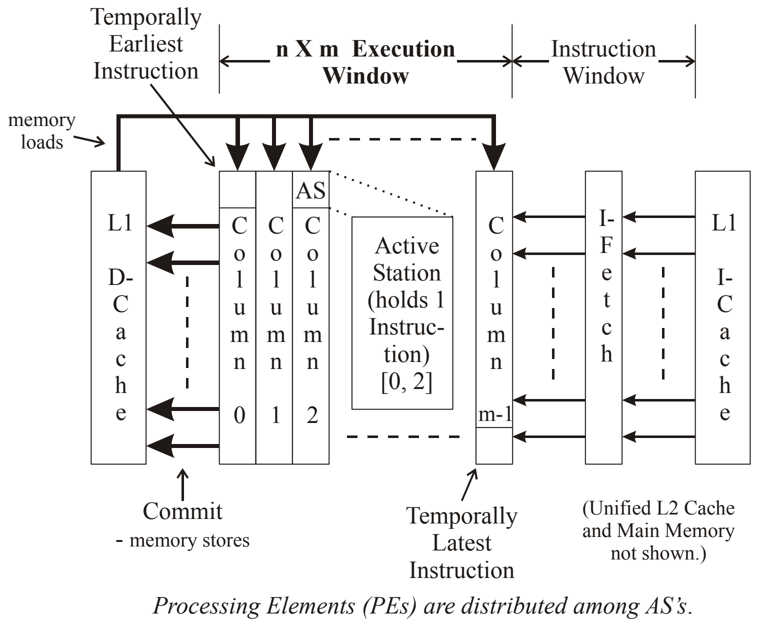

Temporally Earliest Instruction
n X m Execution Window
Instruction Window
memory loads
AS
L1 D-Cache
Column 0
Column 1
Column 2
Active Station (holds 1 Instruction) [0, 2]
Column m-1
I-Fetch
L1 I-Cache
Commit - memory stores
Temporally Latest Instruction
(Unified L2 Cache and Main Memory not shown.)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

Figure 1. Levo high-level logical block diagram. The Execution Window is the key element.

3.1 Time Tags with Active Stations → Low Cost

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

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 program code sequence considered and its outcome. Instruction 9 (I9) 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. I9 has been modified at instruction load time to source only the result of I5. I9 snarfs the result value of I5 when I9's operand register address equals the register address (4b) broadcast on the bus; I9 then executes.

In (c), with time-tagging, no renaming is performed. Instead, each station now has a Last Snarfed Time Tag (LSTT) register. 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 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.

Thus, time-tagging has the large advantages of having only linear growth, O(k), with the number of instructions held in the execution window, and is thus scalable; also, its execution algorithm is simple. Therefore time-tagging has a low cost relative to large traditional superscalar processors' methods (O(k^2) hardware).

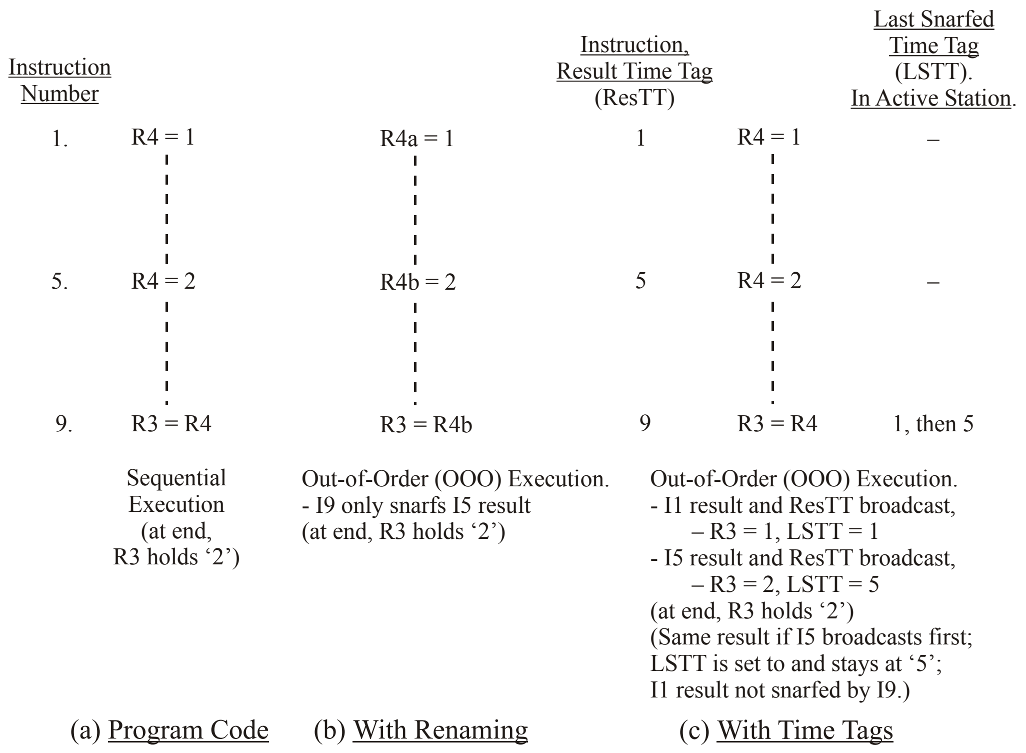

Instruction Number
Instruction, Result Time Tag (ResTT)
Last Snarfed Time Tag (LSTT). In Active Station.
1.
R4 = 1
R4a = 1
1
R4 = 1
–
5.
R4 = 2
R4b = 2
5
R4 = 2
–
9.
R3 = R4
R3 = R4b
9
R3 = R4
1, then 5
Sequential Execution (at end, R3 holds '2')
Out-of-Order (OOO) Execution.
- I9 only snarfs I5 result
(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) Program Code
(b) With Renaming
(c) With Time Tags

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

Time-tagging is a very general concept. Levo also uses it for memory instructions and operands, as well as branches and predicates. With memory operands the memory address is used instead of a register address for address matching purposes, while the hardware-generated predicates (described in Section 3.3.1) use the time tag as the predicate register address.

Figure 3 shows the detailed components of one operand of an Active Station. There are three other necessary conditions for operand snarfing and instruction execution or re-execution: first, the operand must have changed value (this feature has been used before, e.g., [10]); secondly, 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).

Time tags are used in [14] and common processors solely to squash instruction results occurring after a mispredicted branch. Time tags as used in Levo were originally proposed for simulation in the Time Warp model [6], and later in the Warp Engine [1] processor. The latter's implementation 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 [5, 12].

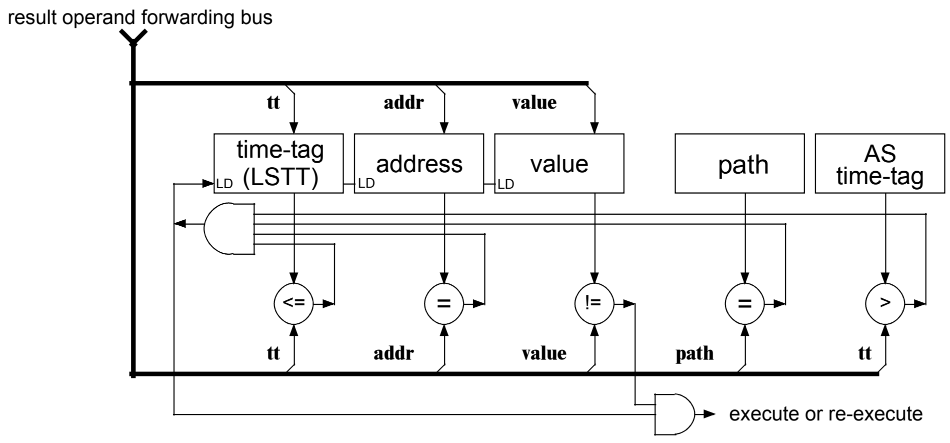


Figure 3. Levo Active Station (AS), showing comparison operations necessary for operand snarfing and instruction execution.

3.2 Segmented Result Buses → Scalable Microarchitecture

In Levo, segmented or *spanning* buses are used to propagate active station results to 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 [3, 16]. Adjacent segments are connected via *Register Forwarding Units* (RFU), which introduce a small delay, usually one cycle, from segment to segment; see Figure 4. The idea is that the later in the E-window 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. Spanning buses in tall columns may be split into multiple segments. Since the length of segments need not change with the size of the machine, the spanning buses help make Levo scalable.

RFU's hold a version of the Instruction Set Architecture (ISA) register state. Since RFU's are distributed throughout the E-window, their contents are globally inconsistent but locally accurate. Compared to what a typical register file for a Levo-sized machine would need, RFU's need a very small number of ports.

In Figure 4, note that nominally there is one RFU per sharing group and one spanning bus per RFU. There are also Memory Forwarding Units (MFU), Predicate Forwarding Units (PFU), and spanning buses (not shown) for each of the corresponding data. 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.

Other novel features are the elimination of a centralized register file, and the simplification of state commitment, both by using RFUs. To see this, assume in the figure that column *i*-1 is column 0. 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. By the time an RFU's state reaches column 0, it contains the equivalent of what would normally be thought of as the ISA register state. Since the register values have already been broadcast to 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; this is covered in Section 4.2, with the rest of the memory system.

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.

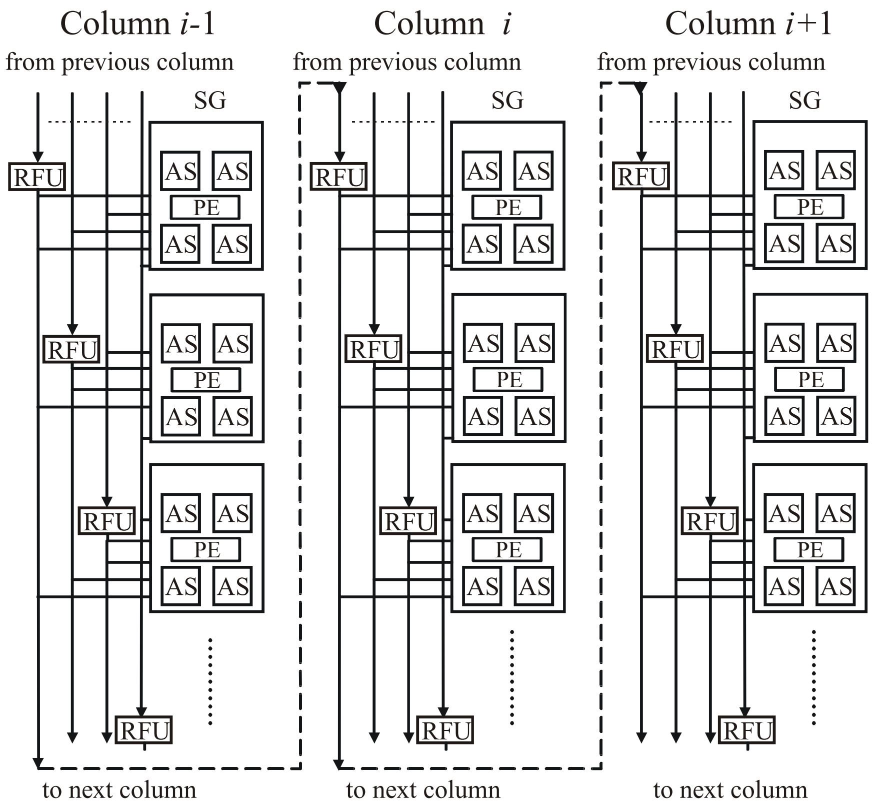


Figure 4. Spanning buses in generic Levo E-window. Note that the bus's length does not change as columns are added to the machine. Each SG drives its spanning bus and RFU through one bus, and snoops the output of the RFU through another connection. Each RFU also snoops the other buses at its level in the same column (not shown), to maintain RFU consistency for its SG.

3.3 ILP Enhancement Methods → High IPC

3.3.1 Hardware Predication

Full *hardware-based predication* is a new implementation of Minimal Control Dependencies. MCD allows the execution of all branches concurrently, as well as allowing the execution of instructions after a branch's domain [18] independently of the branch. Formerly hardware-based methods required O(k^2) hardware to realize MCD, k being the number of instructions in the E-window. The new method's 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 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) 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 (I3) in the execution window executes, it broadcasts its target address (7) (on a separate spanning bus), predicate value and time tag (3). Non-branch active stations following the branch, whose instruction addresses do not match the target address (I5, I9), snarf the predicate and its time tag. The branch is initially not taken, so no entries are made in the following instructions' taken-branch tables. Since the tables are empty, and the snarfed predicate is false, the instructions execute and broadcast their results normally. After the misprediction the branch is taken and the time tag (3) is entered in the stations' (I5, I9) taken branch tables, with the corresponding valid bits asserted. As long as there are one or more entries in a table, the snarfing instructions (I5, I9) are disabled and effectively branched around.

If there is a match between the broadcast target address (7) and a following station's instruction address (I7), then the instruction is just after the end of the branch's domain [18] (I4-I6) and should thus be unaffected by the branch's execution. This station still snarfs the predicate (taken) and its time tag (3) and then rebroadcasts them with the predicate changed to a *canceling predicate*. Later stations (I9) with a predicate address in their taken branch table matching the canceling predicate address (3) invalidate the table entry (3 to 'none'); thus, the corresponding branch no longer affects the operation of the stations (I7-I9), the desired effect.

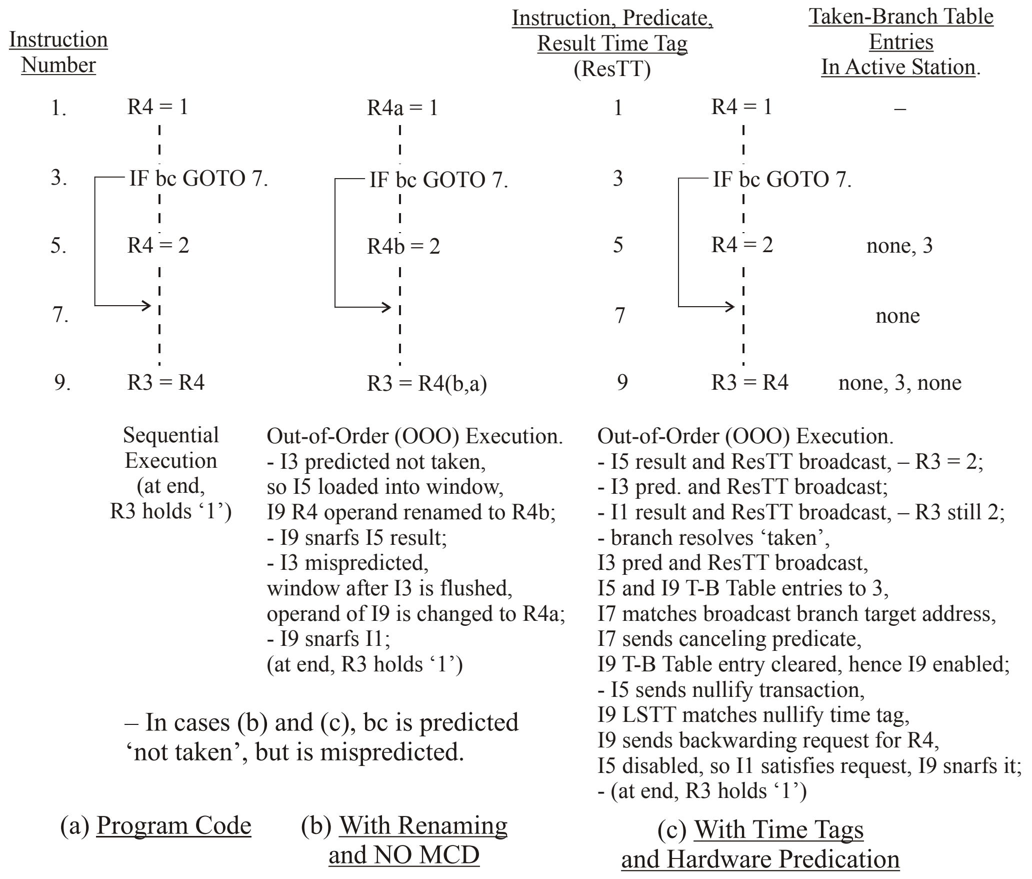
When a branch is mispredicted not taken the instructions within the branch's domain must be disabled; this happens as described above. But also, the now-disabled instructions within the domain that have already executed and broadcast their results must *nullify* these results and cause dependent instructions to re-execute. In order to do this, executed-now-disabled instructions (I5) broadcast a nullify transaction, containing the instruction's time tag (5) and the register address (R4). Any later instruction (I9) with a matching operand register address (R4) and LSTT equal to the broadcast time tag (5) (dependent instruction) sets itself to the unexecuted state, invalidates its LSTT, and sends a backwards request for the nullified operand (R4). A prior instruction with a valid result (I1), or an RFU, satisfies the request, and execution (of I9 et al) resumes normally.

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

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, and they are mainly narrow.

3.3.2 Disjoint Eager Execution (DEE)

DEE is described in [19]. It is an optimal form of speculative execution with respect to constrained resources. 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; instead the "static tree" heuristic of [19] is used. 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.



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.

DEE is realized in Levo by including AS's solely dedicated to D-path execution in the 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 50%.

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 take the D'd branch to have executed in the opposite sense as in the M-path. Other than that, the 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 corresponding M-path column(s) is relabeled to be a free D-path, and its state is thrown away; the original D-path is relabeled to be the M-path, and execution resumes. The M-path state in the columns logically after the original D-path's last column is kept, but is recomputed as necessary by broadcasting any modified state values from the Forwarding Units in the original D-path. 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 column 0 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.

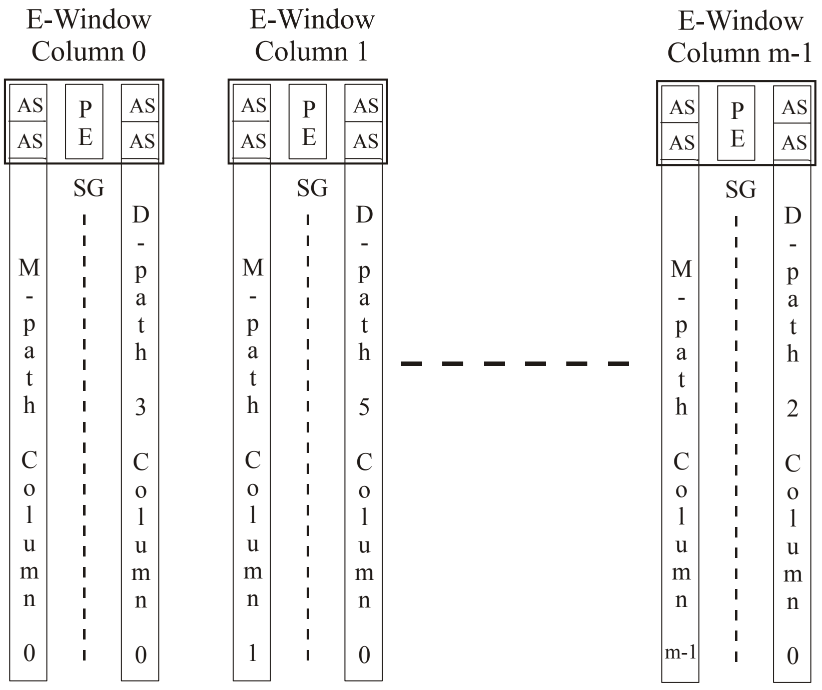


Figure 6. Arrangement of Mainline (M) path columns and DEE (D) path columns. Note that D-paths need not occupy the same execution window column as their corresponding M-path column. D-paths can also be multi-column.

4 Other Issues and Levo Solutions

4.1 Instruction Window and I-Fetch

The I-Fetch hardware obtains entire column(s) of instructions at once from the I-cache for loading into the E-window, once column(s) in the E-window commit and there is room for more instructions. The key here is that instructions are normally fetched in the static or memory order, keeping branches not taken for loading purposes, unless the branch is predicted taken and has a large domain (greater than some fraction of the E-window). In that case the fetch becomes dynamic, resuming from the target of the branch. Initial predicate values for the E-window are predicted separately. This all keeps the fetch mechanism simple and allows for high I-fetch bandwidth. It also tends to keep branch domains with their branches, so that MCD and DEE can be fully exploited.

Backwards branches are unrolled [18, 19] in the I-Fetch unit, with all but the last instance of the backwards branch converted to a forwards branch to enable/disable loop iterations appropriately. The unrolling gives good utilization of the E-window for small loops and improves performance.

4.2 Large Memory Latencies – Modified Memory System

The deep E-window in Levo provides a large tolerance to main memory latency. In prior work [8] we have shown that Main Memory latencies of 800 cycles or more can be tolerated without significant performance degradation. Similar observations for high ILP machines have recently been made by Karkhanis and Smith [7]. These latencies are typical of what is expected in the next few years. While Levo is L2 cache latency tolerant, it is not very tolerant of L1 cache latency.

Store data is buffered 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 accumulated stores from the previous column, holding only the latest value for a given memory store address. PCB's are used to distribute the store buffering throughout the machine and reduce the amount of buffered information. Once committed, and thus in column 0, one PCB's worth of stores (from column 0) are sent to the L1 D-cache to update the memory state. Each L0 entry also has an LSTT register associated with it, used in the same way as the LSTTs are in the AS's.

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 sends it on its column bus (forced broadcast) for snarfing by the requesting load. Normally, stores to L0 from outside its column are not immediately broadcast on the L0's column bus in order to reduce bus traffic.

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. We now consider potential coherency problems with this arrangement.

First, a nullified store transaction might arrive after the stale data has been saved in the cache. 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.

Secondly, a store by a committed instruction could occur whose data is later loaded by an instruction not yet in the window. First, note that all of the cache copies write the same data. If the instruction has just been committed, it is guaranteed that at least one of the PCB's has a copy of the data. Even if the new instruction does both a cache access (potentially getting the wrong data) and an MFU backwarding bus access (PCB has the right data), the instruction will (eventually) use the data from the PCB, because its time tag is greater than that of the cache's data. Note that the difference between column commit and cache write update, in the floorplan shown in Figure 7, is only 3 cycles in the worst case. PCB contents can be kept for many cycles, as necessary.

5 Physical Considerations: Column Renaming and Chip Floorplan

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 avoids 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 their values re-used. Therefore the column renaming greatly simplifies the machine wiring, and eliminates the power consumption associated with a physical shift.

A Levo chip floorplan is shown in Figure 7. The goal is to demonstrate 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.

The geometry used is: 8-4-8, 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). 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' spanning buses 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 the reduced clock frequency of about 13%. This is offset much more by the IPC speedup of Levo for the geometry considered, at least a factor of 2.

The Levo chip as described above is estimated to use about 600 million transistors; this is derived from both actual VHDL synthesis of key components [20] as well as rough estimates from the EV8 work. For a billion transistor chip, this leaves plenty of hardware for other purposes. The current cost of the branch predictors is included in the above estimates, but the predictors are not included in the floorplan since they have not been tuned. The data value predictors are not included in either the cost or the floorplan since they currently add little to the performance, and thus are not needed.

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

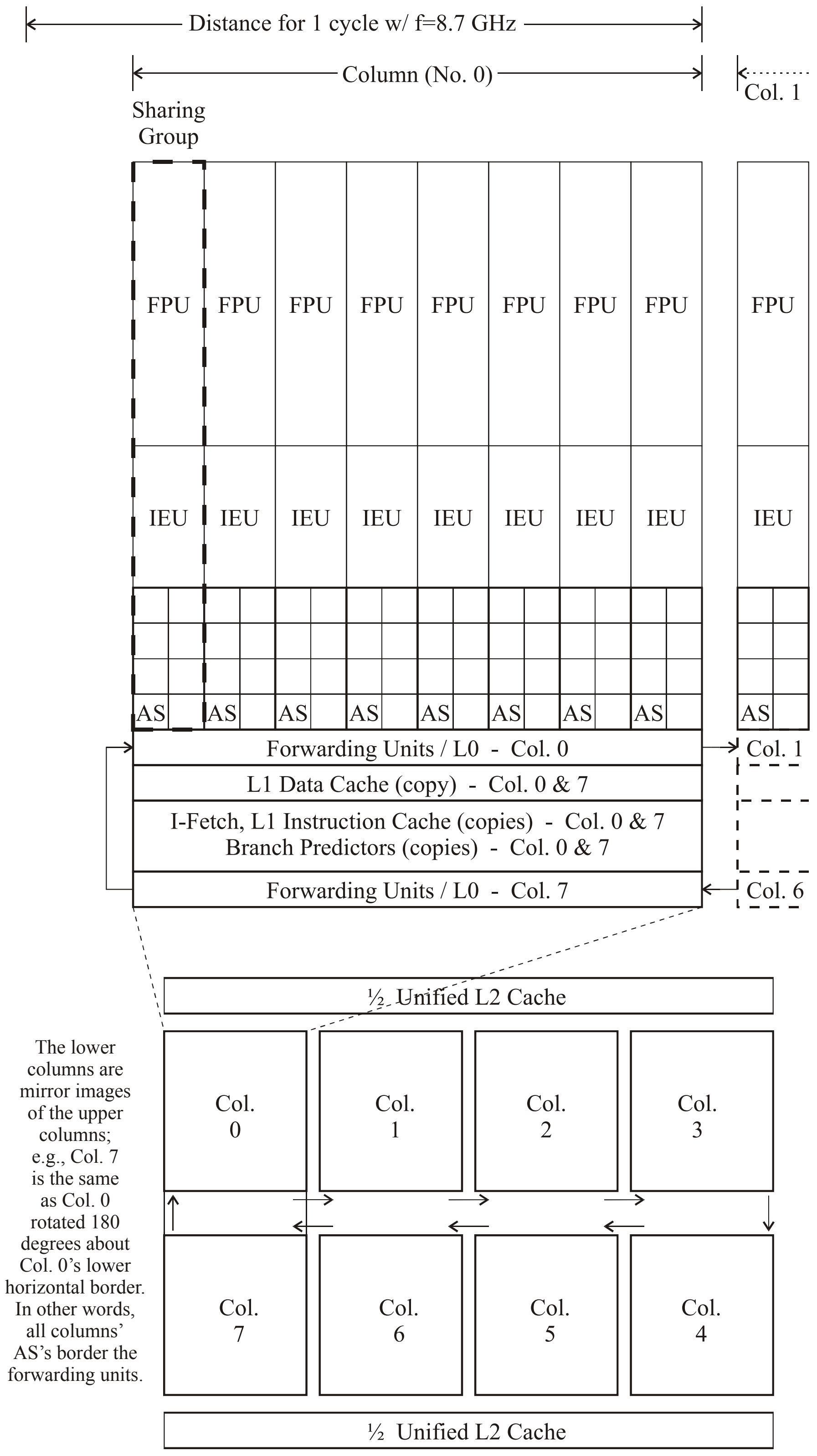

Distance for 1 cycle w/ f=8.7 GHz
Column (No. 0)
Col. 1
Sharing Group
FPU
IEU
AS
Forwarding Units / L0 - Col. 0
L1 Data Cache (copy) - Col. 0 & 7
I-Fetch, L1 Instruction Cache (copies) - Col. 0 & 7
Branch Predictors (copies) - Col. 0 & 7
Forwarding Units / L0 - Col. 7
Col. 1
Col. 6
½ Unified L2 Cache
Col. 0
Col. 1
Col. 2
Col. 3
Col. 7
Col. 6
Col. 5
Col. 4
½ Unified L2 Cache
The lower columns are mirror images of the upper columns; e.g., Col. 7 is the same as Col. 0 rotated 180 degrees about Col. 0's lower horizontal border. In other words, all columns' AS's border the forwarding units.

Figure 7. Levo chip floorplan for an 8-4-8 geometry. The elements are drawn to scale. Sharing Groups (SG) communicate via the centrally located spanning buses in the 'Forwarding Unit' sections. The L1 caches, I-fetch units and predictors are replicated once per vertical column pair.

6 Experimental Methodology

A cycle-accurate trace-based simulator (FastLevo) was written to model Levo's key structures and measure its performance with different machine parameters. FastLevo uses traces of MIPS-1 machine code (32-bit machine). The latter is generated from the benchmarks with a native SGI compiler using the '-O' optimization and '-o32' MIPS-1 switch. (FastLevo also simulates the few MIPS-2 instructions occurring in the SGI libraries.)

Ten SPECInt benchmarks were simulated (their reference inputs were used):
SPECInt95: compress, go, ijpeg
SPECInt2000: bzip2, crafty, gcc, gzip, mcf, parser, vortex

Each benchmark was simulated for 100 million instructions with data gathering turned off, to warm up the predictors and caches and ignore program initialization. Data was gathered during the simulation of the next 500 million instructions.

The common machine assumptions are shown in **Table 1**.

Table 1. Levo default parameter values.

Parameter	Value
Branch predictor	2-level gshare w/ 1024 BHT and 4096 GPHT, 2-bit saturating counter, one per AS row, one for I-fetch.
Data value predictor (from [4])	computational-stride predictor w/ 4096 entries, 2 source operands per entry, 2-bit saturating counter per operand, one per AS row.
L0 hit latency	0 cycles
L0 size	32 one-word entries
L0 configuration	Direct-mapped
L0 block size	1 word
L1-I,D hit latency	1 cycle
L1-I,D size (each)	64 KBytes
L1-I,D configuration	2-way set associative
L1-I,D block size	32 bytes
L2 (unified I/D) hit latency	10 cycles
L2 size	2 MBytes
L2 configuration	Direct-mapped
L2 block size	32 bytes
Main memory latency (no misses)	100 cycles
Main memory interleave factor	4
Forwarding Unit delay (no bus contention)	1 cycle
Spanning bus delay (no contention)	1 cycle

7 Experimental Evaluation and Characterization

The major sets of experiments were: performance sensitivity to machine geometry, and performance effect of ideal/real I-Fetch and memory systems. For a point of comparison, SimpleScalar gave an IPC of ??? for the pisa machine model (similar to MIPS-1) assuming an unrealizable 32-way conventionally-constructed superscalar machine (reorder buffer, etc.). Single-column D-paths were used.

7.1 Levo Geometry Effects on Performance

In this set of experiments each machine geometry dimension was varied with the other two dimensions held constant. Each RFU and MFU was assumed to input/drive two buses, to reduce bus contention, while each PFU input/drove one bus. The baseline for comparison was always a geometry with 4 units of the dimension being varied. See Figure 8 for the results.

Most often, increasing any dimension increased the performance, frequently dramatically. The smallest changes occurred with an increased number of columns, with a 10% increase on average when going from 4 to 12 columns. The largest changes were seen with increased Sharing Groups per column, with a 70% increase in performance going from 4 to 12 SG/col. While increasing columns and SG/col gave monotonically increasing trends, increasing the AS/SG gave varying trends. This is to be expected: the former changes increase the number of PE's, while the latter only determines PE utilization, which varies across benchmarks due to code variations. On average, there was no need to go above 8 AS/SG, which gave a 20% improvement over 4 AS/SG. We are investigating the reason(s) why the gain in performance for increased SG/col is so much greater than that for an increased number of columns.

7.2 Absolute IPC Performance

In the second set of experiments the effects of ideal I-fetch and an ideal memory system were examined for several different machine geometries. Ideal I-fetch is simulated by employing an oracle for the I-fetch branch predictor. An ideal memory system is realized by assuming 100% L1 data and instruction cache hit rates. Each forwarding unit was assumed to have 4 buses in order to reduce bus contention as a factor in this set of experiments. The results are presented in Figure 9; all four combinations of ideal/real – I-fetch/memory system are shown, each for four machine geometries. IPC ranges from a low of about 3 to a high of about 80.

Overall, we have five big conclusions from these results. First, with realistic assumptions and a current-sized geometry (8-4-8), we are not yet able to realize high IPC (the harmonic mean is about 4 IPC). Second, good news, there is still more IPC to get, given the high ideal numbers (about 10 IPC for the 8-4-8 geometry). Third, increasing the size of the machine can markedly increase the performance (IPC of about 16 for a 16-8-8 geometry for ideal conditions). Fourth, the memory system is functioning well, primarily leaving the I-fetch system to be improved. Lastly, the ideal I-fetch performance figures are very close to those of the combined ideal I-fetch and ideal memory system figures; thus, substantially improving I-fetch will get us close to the maximum IPC realizable, modulo the use of other ILP-enhancing methods. One possible solution is to simply use a much better branch predictor for I-Fetch.

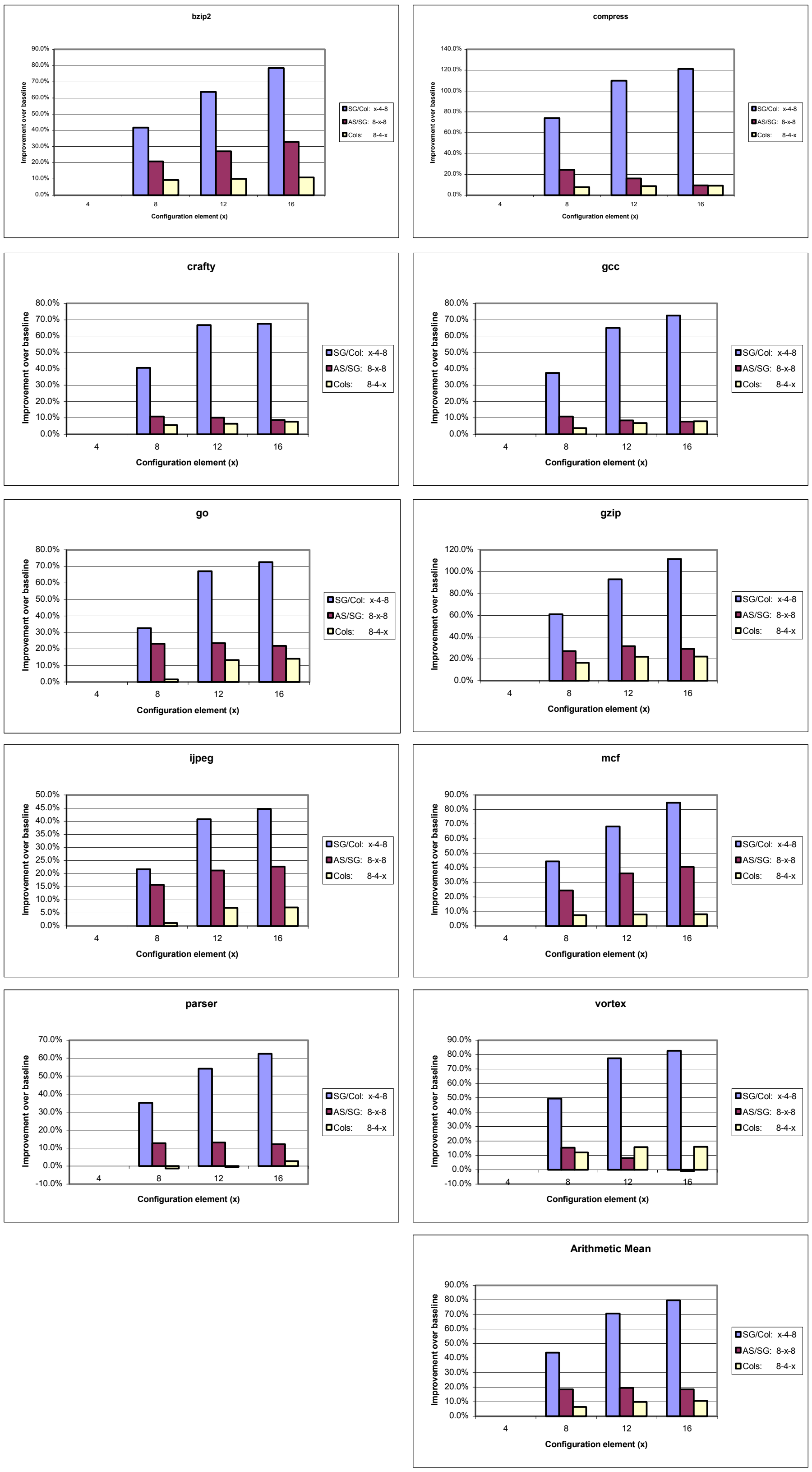


Figure 8. Performance effects of Levo geometry changes.

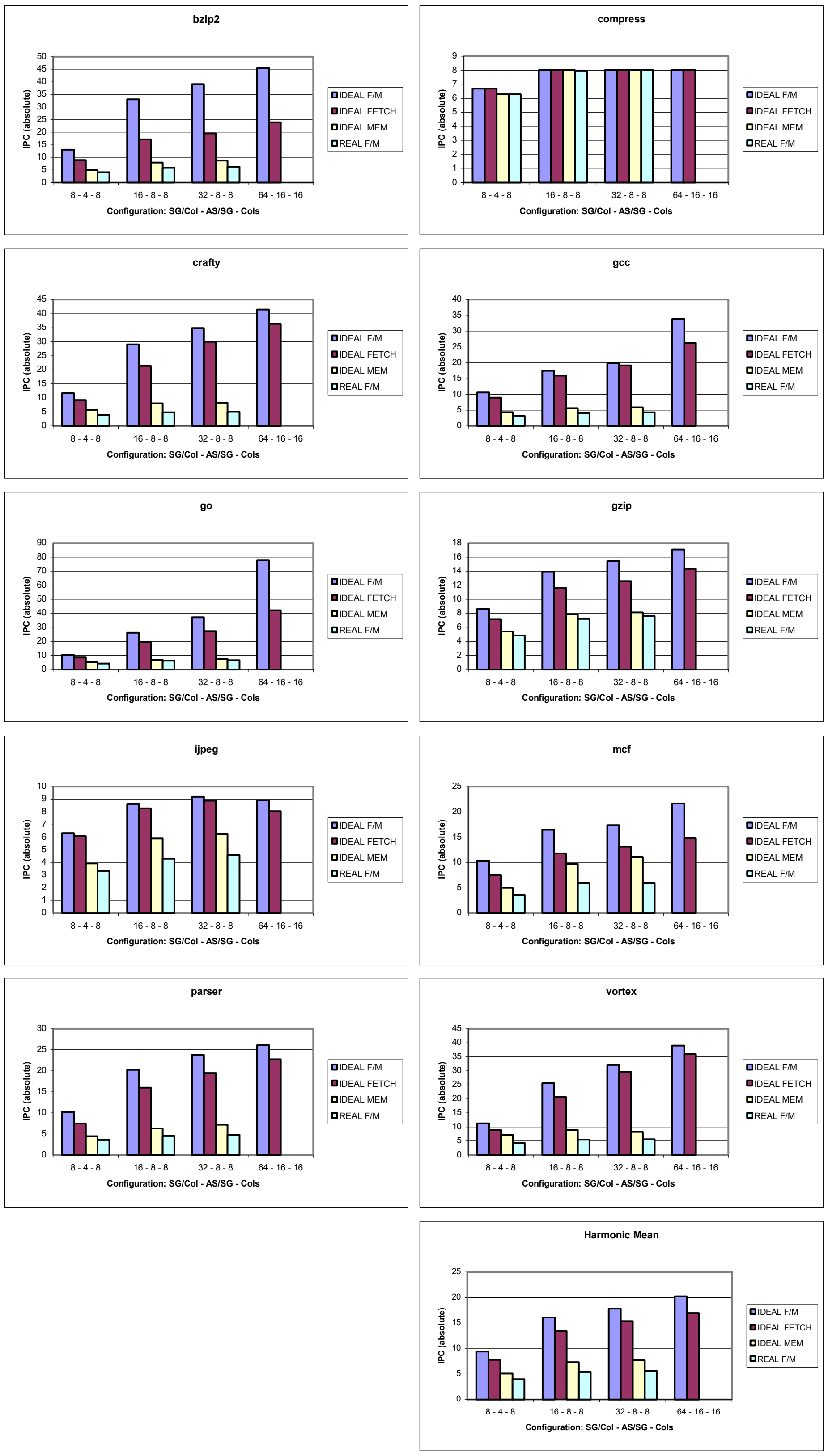


Figure 9. Performance in absolute IPC of Levo with Ideal Fetch and Memory, Ideal Fetch, etc.

7.3 Other Results

A small number of buses is needed to get most of the performance. For the 8-4-8 geometry, having 4 buses per forwarding unit only improved performance by about 3%, while having only 1 bus per forwarding unit reduces performance by about 10%.

The use of value prediction improves performance by less than 0.5%. The problem is that while the initial data value predictions have typical accuracies at instruction load time, the AS operands are overwritten with incorrect values when initially in the E-window, almost completely negating the benefits of the predictions. This is due to the rampant speculation of the resource-flow execution model. Having AS's ignore some operand updates may help.

Doubling the number of columns per D-path while keeping the total number of D-path columns constant resulted in a net performance loss of about 6% for an 8-4-8 geometry. Thus, for smaller machines single column D-paths are superior.

As mentioned earlier, the use of D-paths improved performance by about 50%, for an 8-4-8 geometry.

The use of per-row branch predictors, a necessity, reduces performance by less than 0.3% from an ideal model using a single branch predictor of the same size having access to all branch outcomes.

8 Summary

Several novel techniques were presented, including time tags, active station, specific implementations of hardware predication and DEE, and resource-flow computing.

Levo is scalable, realizable in the near term on high-density chips, and exhibits large IPC. The chief impediment to very high IPC's is I-Fetch. This is a subject for future work.

Thus resource-flow execution is viable, gives good performance now and has the potential for much greater performance.

References

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

[1] 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.

[2] 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.

[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] 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.

[5] 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.

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

[7] T. Karkhanis and J. E. Smith, "A Day in the Life of a Data Cache Miss," in *Proceedings of the 2nd Annual Workshop on Memory Performance Issues (WMPI), at the 29th International Symposium on Computer Architecture (ISCA 2002)*. Anchorage, Alaska, May 2002.
[8] A. Khalafi, D. A. Morano, D. R. Kaeli, and A. K. Uht, "Realizing High IPC Through a Scalable Memory-Latency Tolerant Multipath Microarchitecture," Department of Electrical and Computer Engineering, University of Rhode Island, Kingston, RI 02881-0805, Technical Report 032002-0101, December 21, 2001.
[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 Report 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, "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.
[19] 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.
[20] 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.