Journal of Instruction-Level Parallelism 5 (2003) 1-28 Submitted 12/02; published 8/03

Levo - A Scalable Processor With High IPC

Augustus K. Uht UHT@ELE.URI.EDU
Department of Electrical and Computer Engineering
Microarchitecture Research Institute (µRI)
University of Rhode Island
4 East Alumni Ave.
Kingston, RI 02864 USA

David Morano MORANO@COMPUTER.ORG
Alireza Khalafi AKHALAFI@ECE.NEU.EDU
David R. Kaeli KAELI@ECE.NEU.EDU
Department of Electrical and Computer Engineering
Northeastern University Computer Architecture Research Laboratory (NUCAR)
409 Dana Research Center
Northeastern University
Boston, MA 02115 USA

Abstract

The Levo high IPC microarchitecture 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 lengths are constant, that is, do 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 highly-novel General-Purpose (GP) processor exhibiting large IPC (Instructions Per Cycle) with realistic hardware constraints, scalability and little increase in cycle time. The Levo core (not including the Instruction Fetch Unit) exhibits IPC's greater than 10 on such complex SPECInt benchmarks as gcc and go. The basic Levo 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.

Levo approaches the problem of improving CPU performance problem as a whole, keeping all necessary constraints satisfied. While Levo does use many transistors, billion transistor chips are becoming a reality [4]; further, the trend has always been to use hardware less efficiently as chip transistor densities increased, vis-à-vis all common digital systems. Power and energy consumption are also issues, but we believe it is first necessary to establish the basic performance potential of the microarchitecture; that is the focus of this paper.

In this paper we describe Levo and its operation. We provide detailed simulation results characterizing Levo over a large range of its possible geometries, and present evidence of Levo's even larger potential performance. This paper builds upon [24, 42] by providing an in-depth

description of the microarchitecture, a description of the physical layout (a floorplan), and many new simulation results.

The paper is organized as follows. In Section 2 we review major impediments to high IPC realization. Section 3 provides the Levo logical description, and discusses Levo's solutions to the high IPC problems. Other implementation issues are addressed in Section 4. Section 5 describes the physical operation of Levo and presents a possible Levo single-chip floorplan. In Section 6 we discuss other related work. Section 7 gives our experimental methodology, while Section 8 presents and discusses our simulation results. We conclude in Section 9.

2. High IPC Problems

There are three major impediments to high IPC: 1) high and/or unscalable hardware cost; 2) degradation of (increase in) cycle time, negating IPC performance gains; and 3) lack of high-IPC extraction methods. Prior work has shown that there is much ILP (Instruction Level Parallelism) in typical GP code [18]. Large instruction windows and reorder buffers are necessary to realize a fraction of this ILP [37]; these structures greatly exacerbate the first two high-IPC impediments. We define "cost" as the transistor count or equivalent chip area. Note that this is not the same as "complexity," a measure of the randomness of a system's design. The prior art tends to use complex structures; Levo uses simple ones, easier to lay out, etc. A system is said to be "scalable" if its cost grows linearly or less with an increase in the number of Processing Elements or other key elements.[1]

2.1. High Cost

Typical microarchitectures, such as the Pentium P6 [27] and the Alpha EV8 [31], use a large reorder buffer to maintain the logical correctness of the code executing out-of-order (OOO). The cost of reorder buffers and other dependency checking/maintaining types of structures [44] is large and does not scale with the number of entries; the typical cost is $O(k^2)$ where k is the number of entries in the reorder buffer and/or instruction window, since elements of each entry must be compared to elements of every other entry.

In particular, for a reorder buffer, as the machine size increases the number of both executing and reorder-buffer register results grows. Each executing register result's address and window position must be compared with all of those in the reorder buffer, hence $O(k^2)$ cost growth.

Large pipeline depths also have issues: for a dynamically scheduled high-performance pipeline $O(p^2)$ forwarding paths are necessary to reduce or eliminate the ill performance effects of data dependencies between data in different stages, where p is the number of pipeline stages. In general, for each additional pipeline stage, forwarding paths must be added to all preceding stages; this is $O(p^2)$ cost growth.

[1] As many researchers have observed over the years, no system is truly scalable by this definition: as the system grows, eventually some element grows visibly faster than $O(k)$, often at $O(k*\log(k))$. Within some large values of k, Levo is effectively scalable by our original definition, that is, the multiplier constant for the $k*\log(k)$ term is much less than that for the k term; the k term dominates. In traditional systems, the k^2 term(s) dominate.

2.2. Unscalable Microarchitecture

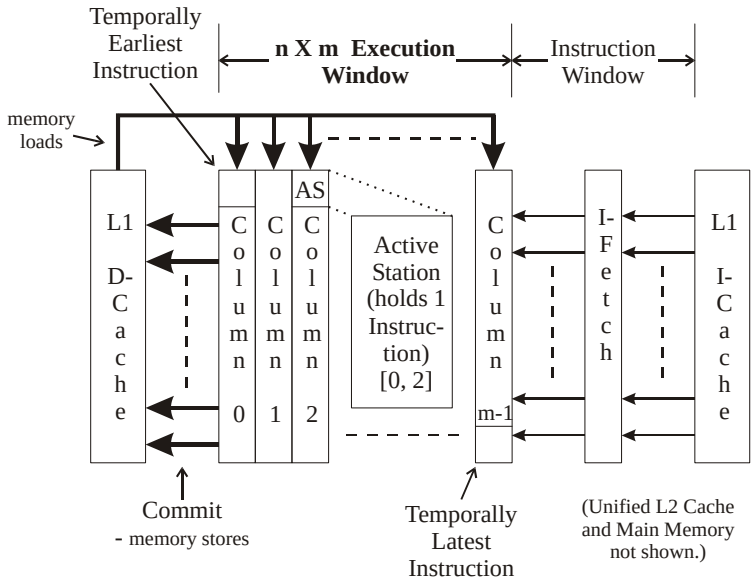
As chip feature sizes shrink, buses become electrically long (high RC time). This leads to longer cycle times and hence reduced overall performance, as does the unscalable hardware mentioned in Section 2.1. Centralized resources such as architectural register files exacerbate the problem. They exhibit longer bus delays and a prohibitively high number of register ports [31]. 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 machines that did well, such as [20]. Part of the problem is that high-ILP methods and combinations of such methods have not been included in realistic designs.

3. Levo High IPC Solutions and Description

Levo consists of 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* instruction Execution Window (E-window).



Processing Elements (PEs) are distributed among AS's.

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

Levo operates as follows. Instructions are fetched from the L1 I-Cache into the Instruction Window and assembled into a block one E-window column high (*n* instructions). When the first column (0) in the E-window commits, the entire E-window contents are logically shifted left and the new instruction block is shifted into the last E-window column (*m*-1). Column 0 commits when all of its instructions have finished executing: the memory store results in Column 0 are sent to the L1 D-Cache, and the ISA register results are sent to later columns. Processing resources are located uniformly throughout the E-window. All instructions in the E-window,

including memory operations, are eligible for execution at any time. Store results, as well as register operation and branch operation results (predicates), are broadcast forward (to the right) in the E-window and snarfed by instructions with matching operand addresses. Load requests are satisfied either from earlier in the E-window or directly from the L1 D-Cache.

There are two key novel features of the E-window that make it scale and ensure that each operand (eventually) gets the right result as its input. First, the broadcast bus is divided into *segments*, each one typically a column long. The bottom, or end, of one segment is coupled to the top of the next segment via storage elements having a small delay. Thus, additional columns can be added to the E-window without impacting Levo's cycle time.

The second novel feature is Levo's own particular use of *time tags*. Each instruction in the E-window has a unique time tag corresponding to its position in the E-window. The time tags provide the proper result-operand linkage with scalable hardware, since all comparisons are made simultaneously with an amount of hardware directly proportional to the machine size. The time tags are used for all dependency checking and for all data operations: memory, register and predicate (branch).

In detail, the E-window holds *n*m* Active Stations (AS). An Active Station is a more intelligent form of Tomasulo's reservation station [40]. Each AS holds one instruction. Small numbers of physically close AS's form Sharing Groups (SG); see Figure 4. All of the AS's in an SG share a Processing Element (PE). Each AS in the E-window has a corresponding *time tag* indicating its instruction's nominal temporal execution order. Time tags are formed by the concatenation of the AS's E-window column number and row number.

Levo's microarchitecture is alterable to match any ISA, with varying performance benefits. So far we have fully realized one GP ISA, the MIPS-1, in our simulator and obtained high performance. No compiler support is needed for Levo, thus legacy code can be executed without recompilation. Adding Levo-specific compiler optimizations is a subject for future work.

While Levo cannot be described in detail as a pipeline without obscuring its logical operation, we will now attempt some kind of analogy with a classic pipelined machine in order to help the reader have some point of reference. (Note that the E-window columns are NOT distinct pipeline stages.) The analogy is as follows, classic pipeline-stage to Levo elements/operation.

1. FETCH : Fetch of instructions by the I-window.
2. DISPATCH : I-fetch buffer is loaded into the rightmost E-window column, that is, the last column's AS's are loaded with instructions.
3. ISSUE : AS contents (operands) sent to a PE.
4. EXECUTE : PE performs appropriate operation, e.g., 'xor' or address computation.
5. WRITE-BACK : There is no Re-Order Buffer.
    - For memory stores : data from E-window column 0 sent to L1 D-cache.
    - For register results : no action, since the result data has already been sent to later columns in the E-window (no explicit architectural register file is used).
6. END-OF-COMMIT : E-window logically shifted left.

Note that in Levo multiple 'ISSUES' and 'EXECUTES' happen concurrently and possibly more than once for the same instruction, and potentially without any 'DISPATCHES' or 'FETCHES' occurring for some time. We have ignored the key Levo elements of operand snooping, snarfing, and broadcasting. Levo's detailed operation is described in the following sections.

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. Figure 2 (a) shows the program code sequence considered and its outcome. Instruction 9 (I9) uses the closest previous value of R4 as its input. Figure 2 (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 Figure 2 (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 the instruction number). Snooping active stations now also compare the broadcast result time tag (ResTT) with that held in the LSTT. If either the result is later than that last snarfed (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 loaded with ResTT. This ensures that only the closest previous version of an operand is used by an instruction for its own last execution. Thus, in the figure, if I1 executes first, I9 executes twice: once with R4=1 (from I1) as its input, and the final time with R4=2 (from I5) as its input. If I5 executes first, I9 only executes once; it ignores the broadcast result from I1 when I1 does execute.

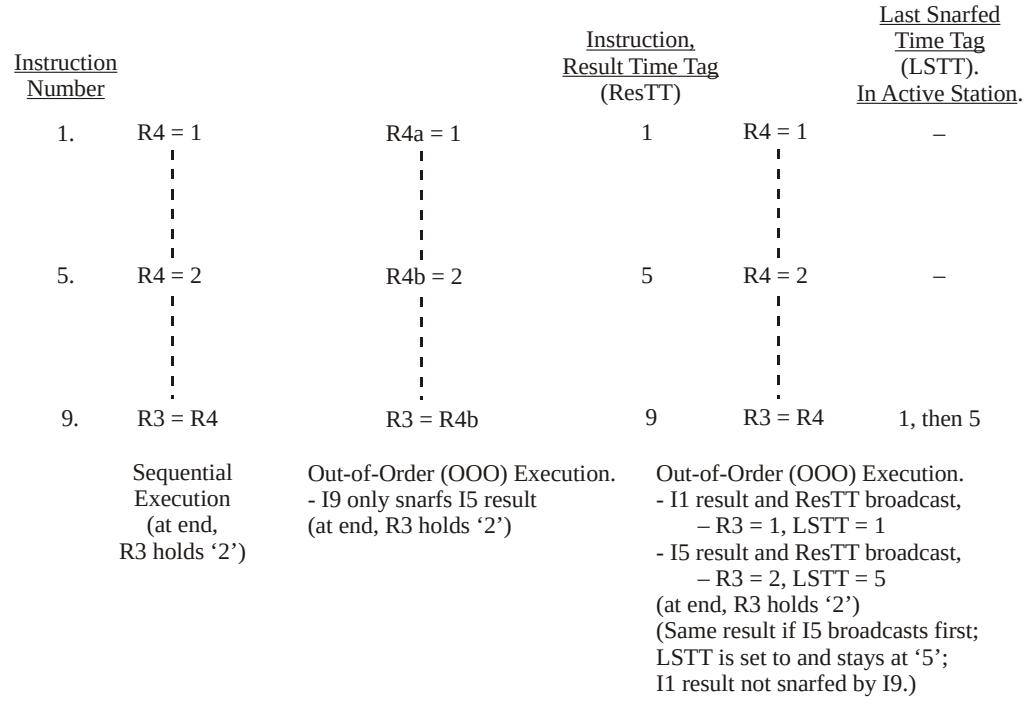


Figure 2: Time-tagged execution of code sample, with comparisons to other methods. OOO – Out-of-Order; ResTT – Result Time Tag; LSTT – Last Snarfed Time Tag.

Time-tagging thus features only linear cost growth, O(*k*), with the number of instructions held in the execution window, and is thus scalable. Further, its execution algorithm is simple.

With memory operands, the memory address (about 32 bits) is used instead of a register address (about 8 bits) to match store results with load operands. The associated PE is used to compute the memory address from a load or store's input operand register values. Otherwise, the operation of memory instructions in AS's is the same as register instructions, and store results are linked to dependent load inputs. Therefore, memory data dependencies are also minimized.

The hardware-generated predicates (described in Section 3.3.1) use the generating-instruction's time tag as the predicate register address. Predicates are handled separately from memory and register values; they are all independent.

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 [20]; 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 Disjoint Eager Execution (DEE) [46]; and lastly, the operand must be from an instruction prior to the AS (ASTT > ResTT).

While using time tags is more complex than the operation of a conventional Tomasulo algorithm, the added comparisons should not add a significant time delay since the tags are quite small. The likely Levo machine versions examined herein typically use time tags 8-10 bits long.

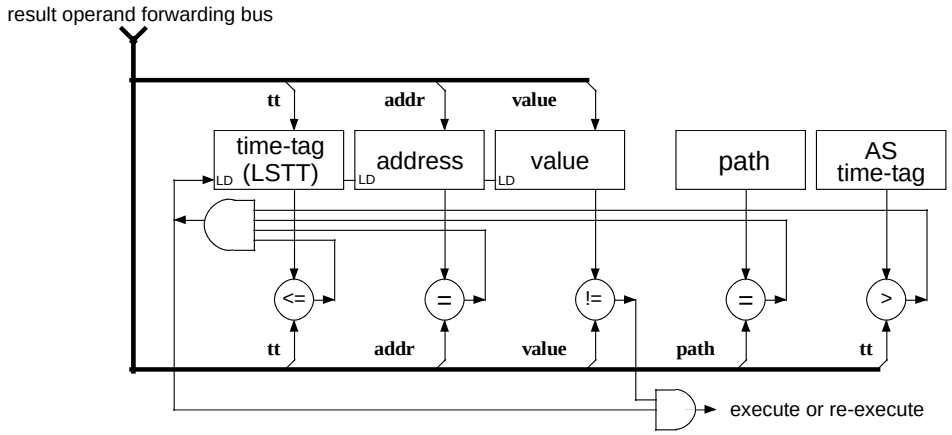


Figure 3: Levo Active Station (AS) operand logic, showing comparison operations necessary for operand snarfing and instruction execution. ASTT – AS time tag.

Time tags are used in common processors to squash instruction results occurring after a mispredicted branch, as well as to maintain instruction order in general [30]. A timestamping method was originally proposed for microarchitectures in the Warp Engine [6]. A strict ordering of timestamps is maintained at receiving processes (in our case, instructions); Levo does not require this. The Warp Engine relied on the use of either floating point numbers or very large integers for the time tags; in Levo, the instructions' E-window positions are the time tag values, and hence are just small binary integers. Further, Warp only tagged memory references, required the use of bandwidth-consuming "anti-messages" for mis-speculation rollback, and used the tags only for control-flow ordering.

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 avoids a performance penalty because a result is likely to be used soon after it has been created [8, 37]. 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 hidden. Segment length is independent of column height. Since the length of segments need not change with the size of the machine, the spanning buses help make Levo scalable.

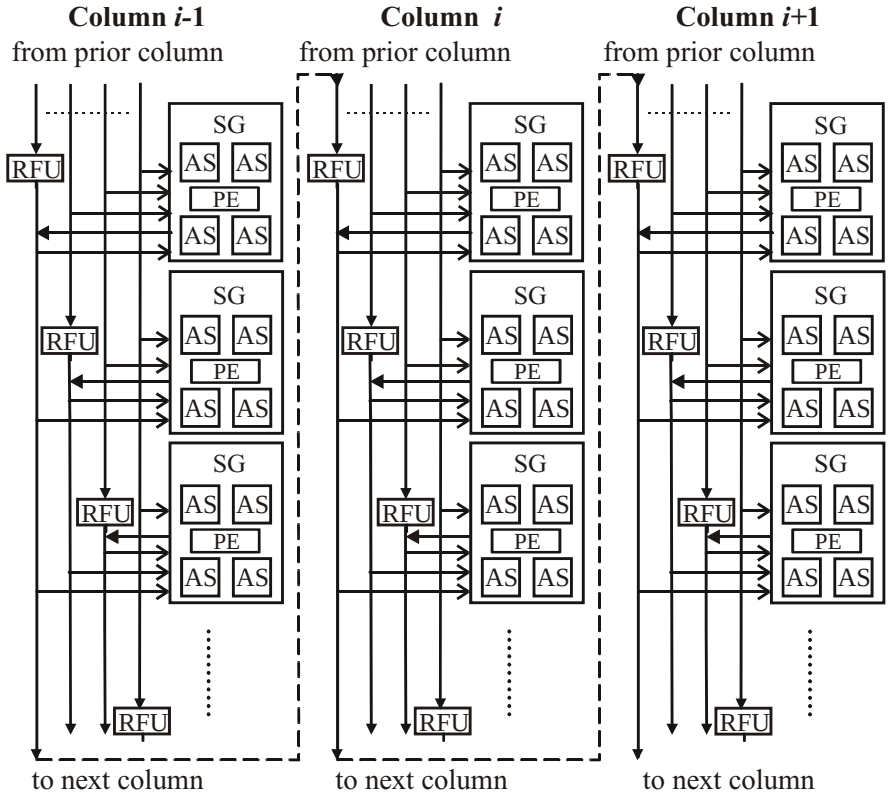


Figure 4: Spanning buses in a generic Levo E-window. A bus's length does not change as columns are added to the machine. (Physically, the last column connects to the first column, forming a loop of columns. In the floorplan of Figure 8, the loop is constructed so that the delay across all bus segments is the same.) 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.

RFU's hold versions of the Instruction Set Architecture (ISA) register state. Time tags are forwarded along with their corresponding register values. RFU's also provide a filtering function: multiple writes to the same ISA register in an RFU are combined, keeping the later time tag, and only one result value for that register is forwarded.

There is one RFU per sharing group and nominally one spanning bus per RFU. There are also Memory Forwarding Units (MFU), Predicate Forwarding Units (PFU), and corresponding spanning buses (not shown). The number of ports to/from RFU's, MFU's and PFU's are small and are constant with respect to the size of the machine; this helps ensure scalability.

Other novel features are the elimination of a centralized register file, and the simplification of state commitment, both by using RFU's. To see this, assume in Figure 4 that column *i*-1 is column 0, where instructions are committed. 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 RFU's in later columns, and since a new column's (m-1) RFU's are initialized with the contents of the prior column's RFU's, there is always at least one RFU in the E-window that holds the equivalent of the ISA state, no matter the time difference between writing and reading an architectural register; therefore it is unnecessary to save the ISA register state in a separate register file. The same is true of the predicate state. The memory values, however, must be written to the L1 D-cache, since an MFU cannot hold all possible memory locations.

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. (Physically, the two types of busses may share a single bidirectional bus.)

3.3. ILP Enhancement Methods → High IPC

3.3.1. Hardware Predication

Full *hardware-based predication* is a new implementation of Minimal Control Dependencies [44]. With MCD, all branches may execute concurrently, and the instructions after a branch's *domain* [44] may execute independently of the branch. (A forward branch's domain consists of the static instructions from the branch to its target, exclusive. For a backward branch the domain is inclusive.) Former hardware-based methods required $O(k^2)$ hardware to realize MCD, k being the number of instructions in the E-window, since the control dependency relations of every instruction in the window need to be stored and/or determined with every other instruction in the window. In Levo the cost is $O(k)$, since the amount of predicate storage and computation logic in each AS is constant with respect to the size of the E-window.

In our method predicates are generated completely at run-time. Every branch in the E-window generates a predicate. They are predicted and evaluated solely with hardware, allowing the use of legacy code. Each branch has a predicate output associated with it, held in the AS. Each AS can also hold a branch target address, and holds the station's instruction program address. 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. The size of the table is small and constant with respect to the window size, since table overflow is allowed. The table is used to help determine branch domains for the predication method.

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), 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 (and incorrectly) predicted 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 is detected 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. With one or more entries in each table, the snarfing instructions (I5, I9) are disabled and 'branched around.'

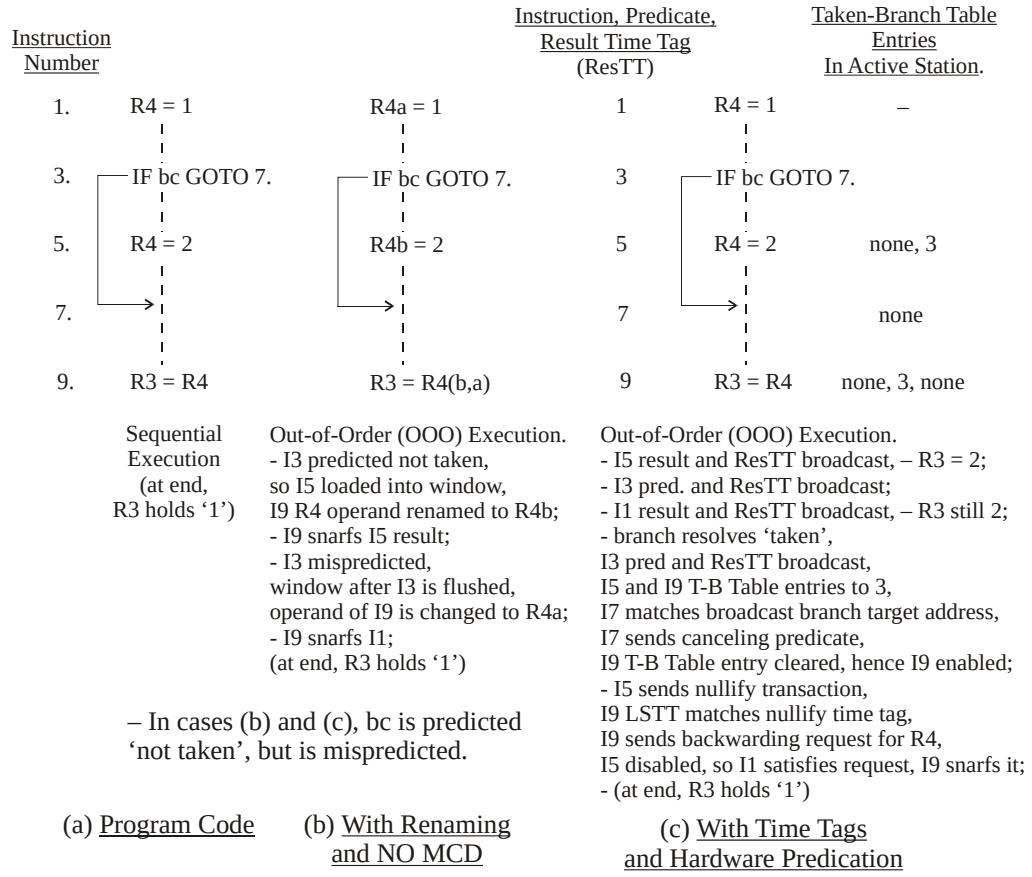


Figure 5: Example of Hardware Predication. Compared to both sequential execution and traditional superscalar (non-MCD) execution. pred – predicate.

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 [44] (I4-I6) and should thus be unaffected by the branch's execution. This station still snarfs the predicate (not 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 (I3) no longer affects the operation of the stations (I7-I9), the desired effect.

Once the misprediction is detected and the branch resolves, 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, the mis-executed instruction (I5) broadcasts 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 request for the nullified operand (R4) on a backwarding bus. A prior instruction with a valid result (I1), or an RFU, satisfies the request, and execution (of I9 et al) resumes normally.

Not including the beneficial effects of Disjoint Eager Execution, the misprediction penalty due to branches whose domain is entirely contained in the E-window is one cycle or more; but this is only for instructions directly or indirectly dependent on the branch. There are usually many instructions that are not dependent on the branch whose operation and execution time are unaffected by the misprediction, that is, experience no misprediction penalty. The misprediction penalty is further reduced with the use of Disjoint Eager Execution, discussed in Section 3.3.2., and can actually be zero cycles, depending on the characteristics of the code in the E-window.

There are other nuances to the correct operation of hardware predication, including overflow of the taken-branch table, an unlikely occurrence. See [22] for more information.

The cost of hardware predication is low, since most of the extra state storage only takes a few bits in the AS. More buses are needed, but not many more than already exist, and most are only 1 to 8 bits wide. This hardware stays the same for all AS's and columns with respect to machine size; thus, hardware predication is scalable.

3.3.2. Disjoint Eager Execution (DEE)

DEE is an optimal form of speculative execution; a proof is in [46]. Both MCD and DEE are needed for very high ILP [46]. DEE requires the most likely instructions to be executed be given priority for execution resources. In Levo likelihoods are not expressly calculated; instead an approximation of the "static tree" heuristic of [46] is used. Therefore this is a form of multipath execution in which there is the predicted or *mainline* path (M) as well as 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 the Sharing Groups; see Figure 6. Typically, Levo has as many D-path AS's as M-path AS's. In effect this means that each Levo E-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 45%.

Conditional branches are assigned to a free D-path, that is, the not-predicted path is *spawned*, after they enter the E-window. A branch spawns a D-path when the branch has not been committed and the branch is the earliest (leftmost) branch that has not already spawned a D-path; see Figure 6. Spawning can take place anywhere in the E-window. The spawned D-path column is loaded broadside with instructions typically already resident in an I-Fetch buffer. Referring to Figure 8, the buffer is replicated between each vertical column pair across the chip. Thus, each replicate (there would be four on the sample chip) resides in the center of the chip, with fast and broad access to potential D-paths in both columns of the column pair.

After spawning, D-paths and M-paths execute concurrently, greatly reducing branch misprediction penalties. While DEE operation is somewhat detailed, the example given in Figure 6, based on that in Figure 5, illustrates the basic concepts. It takes one cycle to switch paths on a branch misprediction, and this is overlapped with instructions' execution. The combination of DEE with MCD provides a system with a very low effective branch misprediction penalty.

Note that D-paths need not occupy the same E-window column as their corresponding M-path column. D-paths can also be multi-column. A D-path can be in any E-window column(s). M-

path columns are usually, but not always, in order from left-to-right, holding adjacent code sections in adjacently numbered M-path columns.

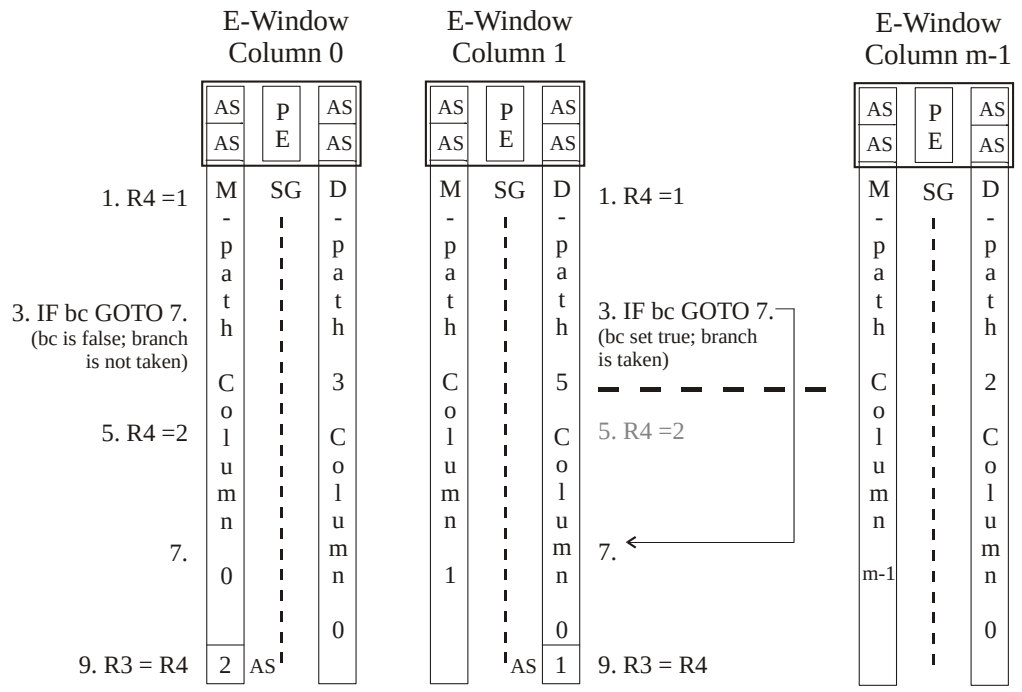


Figure 6: Sample arrangement and DEE operation of Mainline (M) path columns and DEE (D) path columns. The code example is from Figure 5. D-path spawning: D-path 5 is available and is spawned from the branch (I3) in M-path 0 by broadside loading it with the same instructions as in M-path 0, from an I-Fetch buffer (not re-fetched from memory); the D-path branch is set to the opposite state of the spawned branch in M-path 0. Then both paths execute: respective I9's hold different values of R3. After the spawned branch resolves (it was mispredicted), D-path 5 becomes M-path 0: I9 now has the correct result for R3 (1); the old D-path 5 results are rebroadcast to other M-paths (1 to m-1); the old M-path 0 state is thrown out.

4. Other Issues and Levo Solutions

4.1. Instruction Window and I-Fetch

The I-Fetch unit fetches whole column(s) of instructions from the I-cache and loads them into the E-window once early column(s) there commit. 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 two-thirds the size of the E-window, in instructions). In that case the fetch becomes dynamic, resuming from the branch target. Initial predicate values for the branches in the new column(s) are predicted concurrently with multiple branch predictors. This all realizes a simple I-Fetch and high I-fetch bandwidth. It also helps keep branch domains with their branches, so that MCD and DEE can be fully exploited.

The operand data of the loaded instructions is initialized to whatever values the last RFU holds. Thus, the instructions execute as soon as they are loaded into the E-window. While this can lead to wasted computations in instructions in columns loaded later, we have observed little negative effect on performance from this part of the rampant speculation. This is likely due to the later instructions' executions' inherent speculative nature. A key benefit of this approach is the execution algorithm's simplicity.

As discussed in Section 3.2, results are always propagated forward to later RFU's and intervening instructions will re-execute as necessary; therefore either the RFU for the new column has the correct data, or the correct data is on its way. Thus, a result of a register write by an instruction that is later committed is kept in the E-window for use by later, even unloaded, instructions. The current value of such a register is maintained even without intervening writes to the same register.

Backwards branches are unrolled [44, 46] in the I-Fetch unit, with all but the last instance of the backwards branch converted to a forwards branch to enable or disable loop iterations. The overall loop body is wrapped around the E-window and continues to execute as long as the last instance of the backwards branch commits taken. When it commits not taken, the loop exits. Unrolling gives good utilization of the E-window for small loops and improves performance.

Subroutine calls are conditionally inlined in hardware by the I-Fetch unit: when a call is encountered fetching is retargeted to the start of the subroutine, if the call is not in the domain of a predicted-taken branch. Subroutine returns are unconditionally inlined: when a return is encountered, fetching is retargeted to the return address. Return stack(s) aid the process.

Handling exceptions is straightforward in Levo; we follow the approach in [43]. In the case of, say, a page fault, all instructions prior to the faulting instruction (earlier in the E-window) are executed and committed before the fault is handled; no later instructions are allowed to commit. For interrupt-handling a range of possible precise-interrupt points (anywhere in the E-window) is possible, with a corresponding response-time/wasted-computation tradeoff.

4.2. Large Memory Latencies – Modified Memory System

The deep E-window in Levo provides a large tolerance to main memory latency, up to 800 cycles or more [14] with assumptions similar to those in Section 7. Similar observations have been made by Karkhanis and Smith [13]. These latencies are typical of what is expected in the next few years.

Store data is buffered in the E-window; see Figure 7. 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, exactly one for each AS in the previous column; therefore, PCB overflow is impossible. The PCB 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. Since not all AS's in a column are necessarily stores, the amount of data sent to the L1 D-cache is likely to be much less than *n* data words.

Load requests are handled with memory backwarding buses, and are satisfied by earlier Active Stations, earlier MFUs, the L1 D-cache or higher up in the hierarchy. The result is forwarded as a store. Each entry in the L0 cache includes circuitry similar to that for an AS (see Figure 3) to ensure that only the latest data is kept. In the current model, the L0 cache has 32

entries. The L0 caches reduce the pressure on the rest of the memory hierarchy. We have found that 33% of the loads are satisfied by the L0 caches [23].

The MFUs logically shift left with their associated columns whenever the first column commits. (This is true for all of the Forwarding Units, and any other structures associated with a single column.) As will be discussed in Section 5, no actual physical shifting takes place.

As will also 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 cache copies hold the same data, within a few cycles, with all of them replacing the same lines at the same time. While the loads from the different cache copies are likely to be different, the stores are always the same to all of the copies.

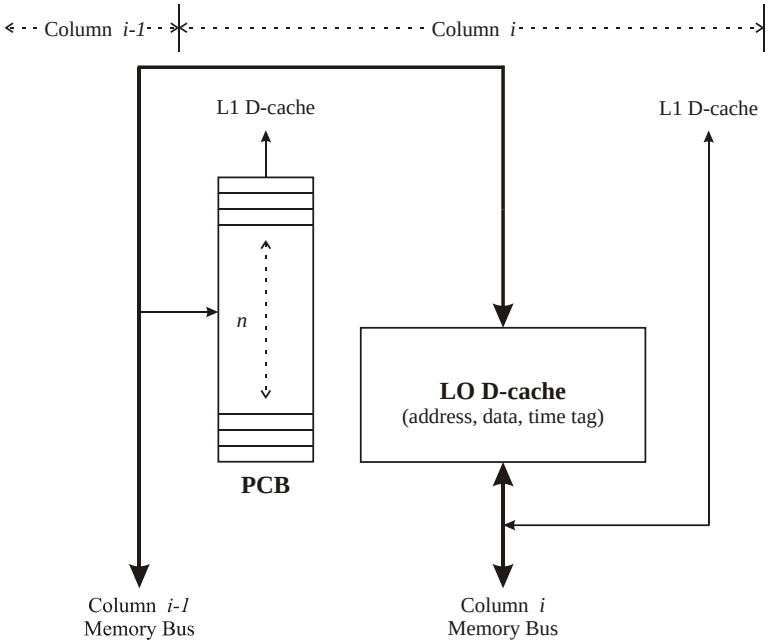


Figure 7: Memory Forwarding Unit (MFU). The Previous Column Buffer (PCB) only holds store data from the prior, *i-1*, column. The PCB's data is only read when its column (*i*) is committed; its contents are sent to the L1 D-cache. There are exactly *n* entries in the PCB, one per AS in the previous column. Load requests from column *i* go to both the L0- and the L1-data caches; if there is no hit in the L0 cache, the requests are sent to the previous column (*i-1*).

5. Physical Considerations: Column Renaming and Chip Floorplan

Levo avoids physically shifting the E-window by renaming the columns. Each physical column has one or more registers associated with it that hold its logical column number. This includes part of the LSTT register in the AS's. When a logical left shift occurs, the logical column numbers of all of the columns are decremented. The column numbers are only a few bits long (about 3 bits for machines considered herein). 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 largely eliminates the power consumption associated with a physical shift.

A Levo chip floorplan is shown in Figure 8. The goal was to demonstrate Levo realizability 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 [31] were used to size similar Levo structures, and to ensure that Levo's critical path is not substantially increased.

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) form the PE of each 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 and in a loop to keep the critical path length low. Thus, spanning bus length is not a substantive issue for Levo, and hence the number or construction of the spanning busses in general are not issues. 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, a performance loss 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 [48] as well as rough estimates from the EV8 work. 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 yet been tuned for the microarchitecture. 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 of the center channel and inserted in the physical loop; the cycle time is unaffected.

In order to reduce cost, the number of FPU's could be reduced if the applications are mainly General Purpose code; that is, some reduction and sharing of FPU's is possible. IEU's could also be constructed of separate Functional Units and shared, as is commonly done in current microprocessors. Also, using a 32-bit data path instead of 64-bits would approximately halve the required hardware. These methods also reduce power consumption.

6. Other Approaches and Related Work

Multithreading originally appeared in the Denelcor HEP machine [36]. In recent years multithreading has become very popular [41], being an attractive method of achieving higher resource utilization and hiding memory latency; it is used in the Intel Pentium 4 Xeon and HT (Hyper-Threaded) processors. Normally threads are from different processes, but can be extracted from a single process with a compiler or, in some cases, the hardware itself [9]. However, while multithreading can exploit some ILP, including that from the use of multipath methods [47], it has not realized much IPC for a single process. Levo is able to achieve the latency hiding of multithreading, but on a single thread. Realizing multithreading on Levo is straightforward.

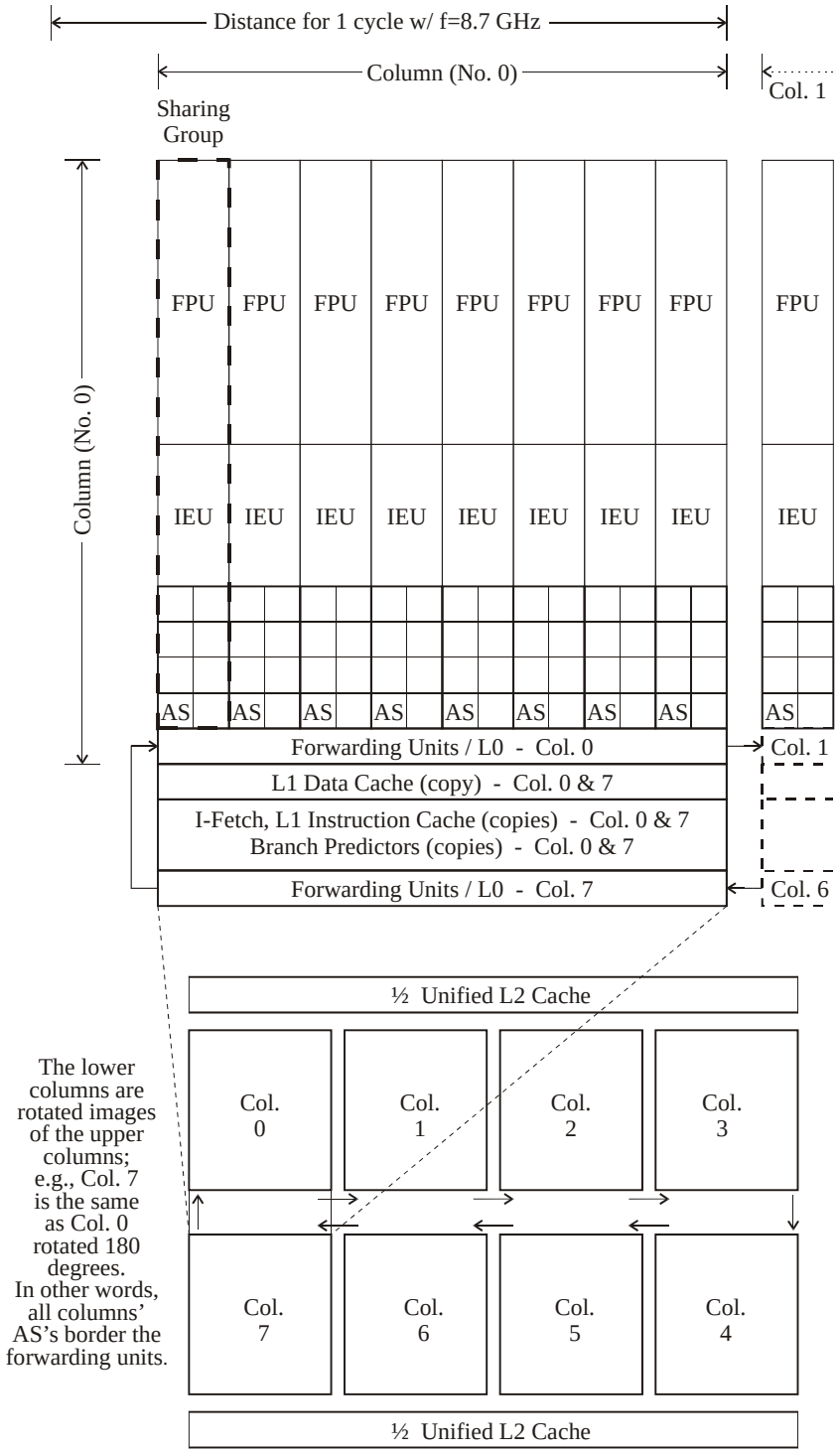


Figure 8: Levo chip floorplan for an 8-4-8 geometry. The elements are drawn to scale. Sharing Groups (SG) communicate via the centrally located spanning buses in the 'Forwarding Unit' sections. The L1 caches, I-fetch units and predictors are replicated once per vertical column pair.

Dataflow machines have been extensively studied [1]; they did not achieve expectations. While Levo shares some characteristics with dataflow machines, on the whole it is much different: Levo uses time tags throughout, has rampant speculation, full predication, and requires no compiler support, none of which are true for classic dataflow CPU's. Limited data flow such as the Tomasulo algorithm [40] used in current superscalar microarchitectures is more successful, but is not current for our goals: predication is not handled, and it does not scale.

Other approaches for dependency enforcement and code ordering use *relational matrices* to hold both explicit data and control dependency information [43-45] or ordering information [39]; these have quadratically-increasing costs with instruction window size. A compiler-assisted form of run-time predication appears in [16]; only simple hammocks are considered.

While methods have been proposed to keep time-ordering information of memory accesses within the memory system, e.g., [11], we believe Levo's distributed solution based on a standard memory system (from L1 on up) is preferable. As previously stated, Levo is well able to tolerate main memory latency, and allows for speculative memory accesses within the E-window.

Lam and Wilson [18] conducted an extensive set of revealing simulations on some of the SPEC89 benchmarks assuming different control dependency resolution techniques. Their most advanced non-Oracle model was the equivalent of simple single-path control speculation with MCD. For this model, an average speedup of 40X was achieved on the integer benchmarks; unlimited execution resources were assumed. For an Oracle predictor (a branch predictor that obtains 100% prediction accuracy), a speedup of 158X was obtained.

Data address and value speculation [21] are promising areas to enhance Levo's performance. Gonzalez and Gonzalez [10] performed a study on the potential impact of these on ILP, using the SPEC95 benchmark suite. When infinite execution resources were assumed a speedup of 42 was obtained. With a restricted window, and including branch prediction, the addition of data speculation provided a 2-3x speedup. In future work, Levo should be able to tap into this ILP resource and produce even larger IPC's.

Levo also achieves dynamic vectorization on legacy code, as is considered in [26] and done in [45]. The latter approach, however, proved to be unattractive to realize in hardware.

Studies of multipath execution began with Riseman and Foster [33] and assumed infinite resources. It has also been studied with restricted resources, e.g., [5, 46].

In recent work [19, 32] larger instruction windows have been examined to improve performance. However, the realized IPC is small (less than 2.5, for the SPECInt 2000 benchmarks). [2] also presents the equivalent of a large instruction window, but with less hardware. While it is cheaper, and tolerates wire latency well, no IPC gains result.

The Ultrascalar machine [12] achieves *asymptotic* scalability, but only realizes a small amount of IPC, due to its conservative execution model.

VLIW [7] or EPIC [17] architectures were promising, but their promise has not been realized; it has become apparent that it is very hard for compiler ILP techniques alone to realize a high IPC (or operations per cycle) on general-purpose code.

In [15] a very novel distributed microarchitecture is proposed for high clock-rates and simplicity. However, there is no improvement in IPC over a conventional superscalar processor, although certainly overall performance would likely increase (the latter is not explicitly measured in the paper). Another non-legacy processor is the RAW machine [38], prototyped in silicon. Tasks must be partitioned either with the limited compiler or by hand. RAW does not appear to be applicable towards general-purpose applications: SPECfp was evaluated, but SPECInt was not.

Grid architectures such as that in [25] are becoming popular in the research community. Levo, on the surface, is similar to such machines but differs widely in its abilities and implementation. In the case of [25], compiler support is necessary for its operation (legacy code cannot be used), some of the resources are still centralized, and the IPC realized is not yet substantial when realistic assumptions are used. In more recent work [34], the arithmetic mean IPC for a TRIPS grid processor on some subset of SPECInt codes (not including `gcc` or `go`), with several ideal assumptions, and requiring compiler support, is about 2.0.

Caches below the L1 level have been proposed; they are similar to our L0 caches, but ours are distributed. In [49] a compiler-assisted microcache-management algorithm for EPIC applications is proposed. It obtained similar results as the Levo L0 caches in terms of the percentage of the loads bypassed, but Levo requires no compiler-assist.

It is difficult to reduce the effects of a large number of ports on the ISA register file [29]. We find it best to eliminate the issue; our approach is supported in [35]. Concerning register-clustering, while it has been suggested that point-to-point networks are preferable to bus interconnects [28], and while they may be for traditional microarchitectures, the assumptions in the latter paper do not apply to Levo in general.

7. Experimental Methodology

A cycle-accurate combined trace- and execution-driven simulator (FastLevo) was written to model Levo's key structures and measure its performance; it accurately models wrong-path behavior. 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 compiler libraries.)

Ten SPECInt benchmarks were simulated:

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 benchmarks' reference inputs were used, except for compress, for which the buffer size was reduced so that compress completed its initialization section within the first 100 million instructions.

The common machine assumptions are shown in Table 1.

8. Experimental Evaluation and Characterization

The major sets of experiments were: microarchitecture assumptions' verification, performance sensitivity to machine geometry, and performance effect of ideal/real I-Fetch and memory systems. Our baseline point of comparison is the SimpleScalar/PISA [3] machine model (similar to MIPS-1 with a conventional superscalar processor's construction). We ran the PISA model assuming an unrealizable 32-way issue unit and with other resources unlimited. It exhibited a harmonic mean IPC of 1.96, with the same benchmark assumptions.

Parameter	Value
Branch predictor	2-level gshare w/ 1024 BHT and 4096 GPHT, 2-bit saturating counter, one per E-window row.
Data value predictor	Computational-stride predictor w/ 4096 entries, 2 source operands per entry, 2-bit saturating counter per operand, one per E-window row.
Word size	32 bits
Processing Element latencies/pipelining	Same as MIPS R4000.
L0 hit latency	1 cycle
L0 size	32 one-word entries
L0 configuration	Fully-associative
L0 block size	1 word
L1-I,D hit latency	1 cycle (cache access time itself; additional 1 cycle bus delay is included in the simulations)
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
Return stacks	2, 16 entries each. (in I-Fetch unit)
Spanning bus delay (no contention)	1 cycle
Forwarding Unit delay (no bus contention)	1 cycle
Buses per RFU and per MFU	2 input and 2 output buses
Buses per PFU	1 input and 1 output bus
M-path to D-path column switch	Switch itself: 1 cycle. D-path results broadcast as bus resources permit.
Columns per D-path	1 column; same number of columns as for M-paths

Table 1: Levo default parameter values.

8.1. Microarchitecture Assumptions' Verification

We first hypothesized that buses can be segmented with non-zero delay forwarding units inserted between the segments. Figure 9 presents the performance degradation experienced when the forwarding unit delay is increased from 0 to 3 cycles. It is seen that the typical delay, 1 cycle, is easily tolerated, having a performance loss of 6%, confirming the hypothesis.

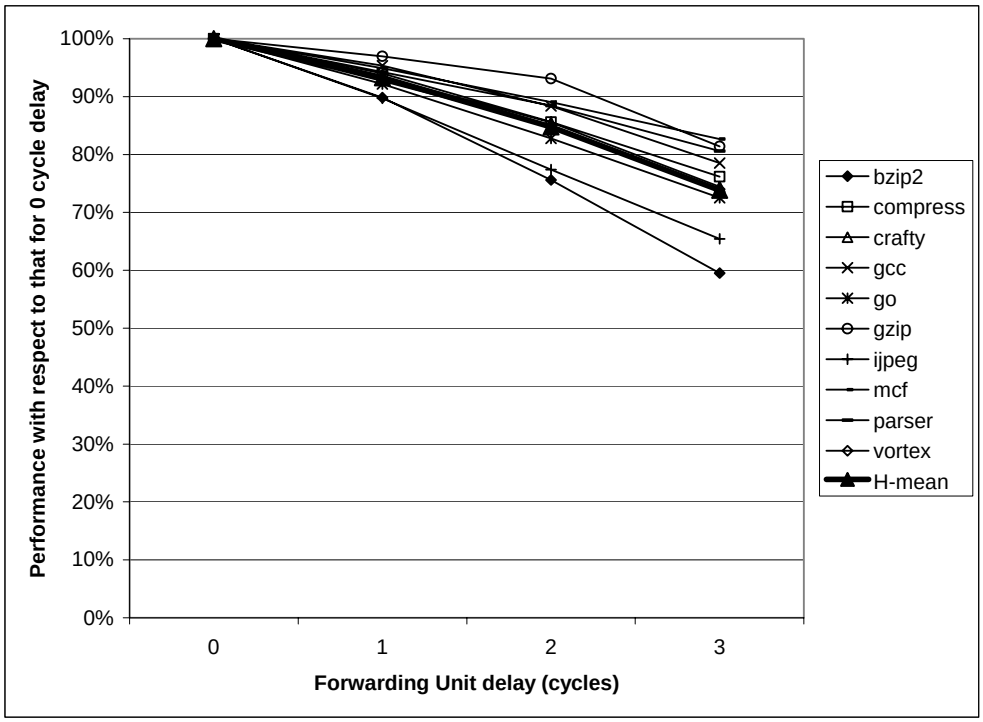


Figure 9: Performance versus Forwarding Unit delay. The baseline is the performance with 0 cycle FU delay. Performance is relatively insensitive to FU delay. 8-8-8 Levo geometry used with spanning bus length of 8 SGs.

We also hypothesized that buses need only be some fixed length (as a machine design increases) to capture most of the performance. Figure 10 shows the performance improvement with increasing bus length, conservatively assuming a constant spanning bus delay of 1 cycle. It is seen that there is only an 11% performance increase when doubling the spanning bus length from 8 to 16 SGs, with a much larger improvement of 29% going from 4 SGs to 8 SGs; therefore, a spanning bus length of 8 is necessary and adequate, and the hypothesis is confirmed.

8.2. Levo Geometry Effects on Performance

In this set of experiments each machine geometry dimension was varied with the other two dimensions held constant. See Figure 11 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 16 columns. The largest changes were seen with increased Sharing Groups per column, with a 55% 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 12 AS/SG, which gave a 19% improvement over 4 AS/SG. Part of the reason(s) why the gain in performance for increased SG/col is greater than that for an increased number of columns, but only part of the reason, is the short spanning bus length (4) of the baseline (4-4-8) for SG/col; see Figure 10.

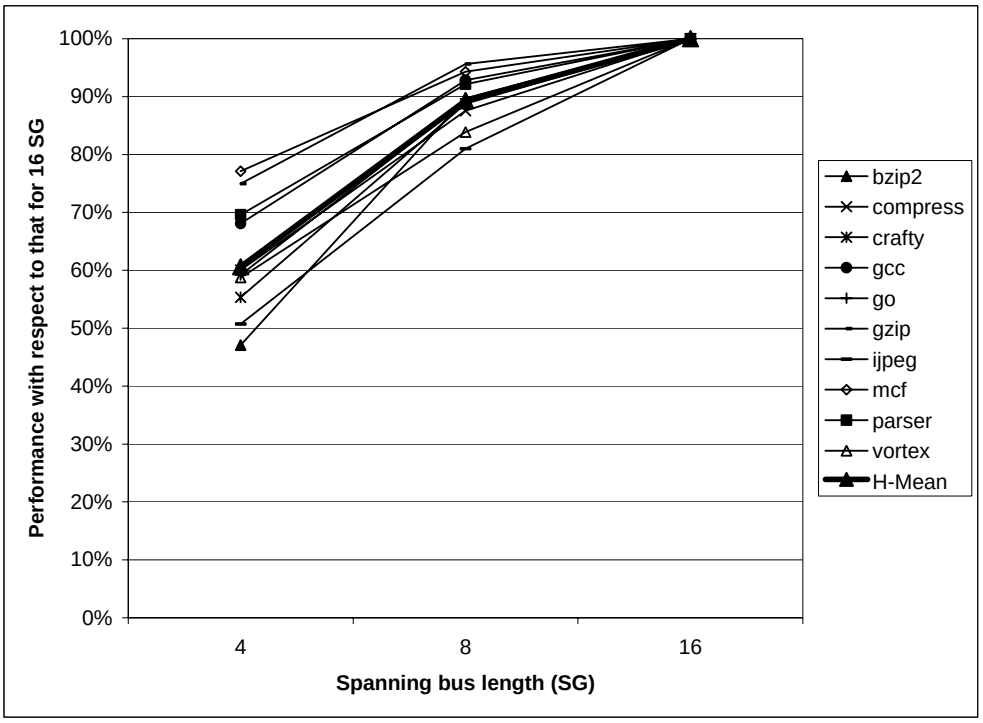


Figure 10: Performance versus spanning bus length. The baseline is the performance with 16 SGs. Most of the performance, 89%, is realized with a spanning bus length of only 8 SGs. 16-8-8 Levo geometry used.

8.3. Ideal/Real IPC Performance

The effects of ideal/real I-Fetch and an ideal/real memory system were examined for several different machine geometries. Ideal I-Fetch is realized by using oracles for the branch predictors at instruction load time. Note that the branch direction can change within the E-window during program execution; thus, the oracle only applies to I-Fetch, not execution. An ideal memory system is realized by assuming 100% L1 data and instruction cache hit rates. The results are presented in Figure 12; all four combinations of ideal/real – I-Fetch/memory system are shown, each for four machine geometries. IPC ranges from a low of about 4 to a high of about 31. We have also seen IPC's of up to 80 with a 64-16-16 geometry, ideal instruction fetch, and 4 buses per forwarding unit (not shown in the figure).

Overall, we have three major 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). Lastly, the memory system functions well with or without Ideal I-Fetch, primarily leaving the I-Fetch system to be improved. We are currently studying many possible solutions to the I-Fetch issues.

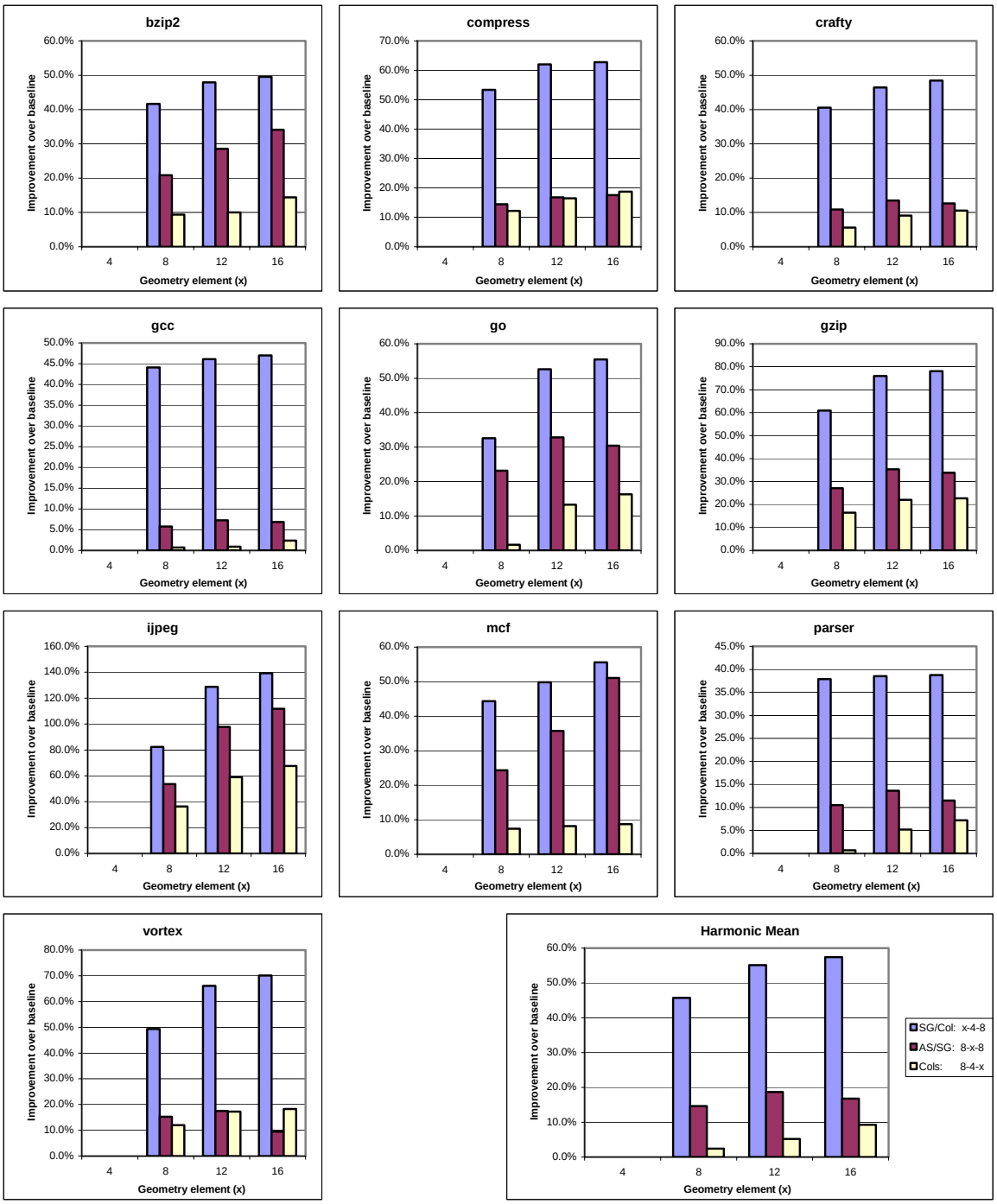


Figure 11: Performance effects of Levo geometry changes. (Legend: in lower right-hand corner.) Each geometry variable is varied in turn, holding the other variables constant. Each baseline is the performance of the geometry having quantity 4 elements of the corresponding independent variable. All geometries used 8 SGs for the spanning bus length, except for 4-4-8, which used 4 SGs. (Recall that the geometry notation is: SG's per column – AS's per SG – M and D paths, each, per column.)

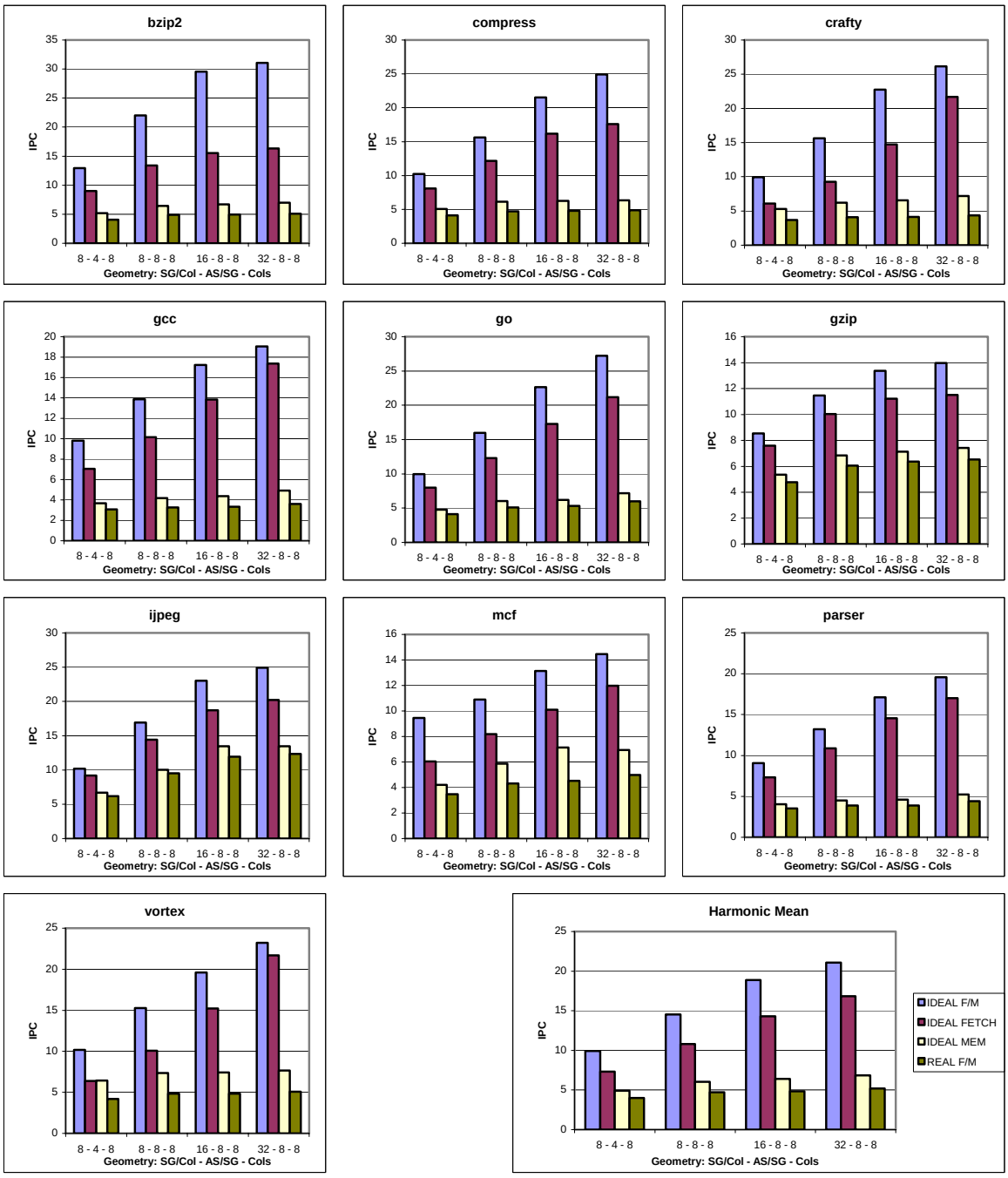


Figure 12: Performance in IPC of Levo with Ideal Fetch/Ideal Memory, Ideal Fetch/Real Memory, Real Fetch/Ideal Memory, and Real Fetch/Real Memory (see legend, lower right corner). (Again, the geometry notation is: SG's per column – AS's per SG – M and D paths, each, per column.)

8.4. Other Levo Characteristics' Results

Five other additional experiments were performed, again over all benchmarks, to investigate certain other performance characteristics of Levo. The experiments, their results and conclusions are presented in Table 2. A Levo 8-4-8 geometry was used throughout.

Experiment	Result(s)	Conclusions
Varying number of buses per Forwarding Unit. Baseline: 2 buses per Register FU and Memory FU, and 1 bus per Predicate FU.	To 1 bus/FU: 14% IPC loss To 4 buses/FU: 3% IPC gain	For current or near in geometries, the baseline is a good design point. Little is gained by going to more buses.
Removal of value prediction.	IPC loss of less than 0.8%	Don't use a traditional value predictor with current Levo design.[2]
Going from 1 to 2 columns per D-path, total D-path columns held constant.	IPC loss of about 8%	Single column D-paths are preferred, at least for smaller machines.
Use of D-paths.	IPC gain of about 45%	Keep D-paths in Levo.
Use of per-row branch predictors, a necessity, vs. baseline of a single branch predictor of the same size using all branch outcomes.	IPC loss of about 0.4%	Using per-row predictors, even with their limited view, does not significantly reduce performance.

Table 2: Other Levo experiments.

9. Summary

One billion transistor microarchitectures have many daunting requirements: high IPC, high main memory latency tolerance, high clock rates, and ability to execute legacy codes. Further, such machines must honor hard chip realization constraints such as scalable structures and short buses. This paper has proposed the Levo microarchitecture, targeted to satisfy all of these requirements. The reorder buffer and scalable bussing structure issues have been thoroughly addressed and resolved in Levo. The performance simulation results are very encouraging, both verifying the basic tenets of the resource flow model and demonstrating IPC's in the 10's for the Levo core E-window and memory system. We are currently pursuing improvements including more accurate I-Fetch and data value prediction.

[2] 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. We are also exploring other value prediction approaches.

Acknowledgements

We are extremely grateful to Bob Colwell, Ed Davidson and John Shen for taking the time to thoroughly review a draft of this paper. Their comments were most helpful. We also thank the anonymous reviewers for their detailed and helpful comments.

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. Alireza Khalafi was also supported by IBM and Compaq. Patents applied for.

References

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

[2] E. Brekelbaum, J. Rupley II, C. Wilkerson, and B. Black, "Hierarchical Scheduling Windows," in *Proceedings of the 35th Annual International Symposium on Microarchitecture*. Istanbul, Turkey: IEEE, ACM, November 2002.

[3] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2," URL: http://www.simplescalar.com/docs/users_guide_v2.pdf, created 1997, accessed: June 14, 2002.

[4] D. Burger and J. R. Goodman - Guest Eds., "Billion-Transistor Architectures," *IEEE COMPUTER*, vol. 30, no. 9, September 1997.

[5] 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.

[6] 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.

[7] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," in *Proceedings of the Second International Conference Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*: ACM and IEEE, September 1987, pp. 180-192.

[8] 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.

[9] R. Ginosar and R. Kol, "Adaptive Synchronization," in *Proceedings of the 1998 International Conference on Computer Design*, 1998.

[10] 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.

[11] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi, "Speculative Versioning Cache," University of Wisconsin, Madison, Technical Report TR-1334, July 1997.

[12] 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.

[13] 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.

[14] 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, April 2, 2002, URL: http://www.ele.uri.edu/~uht/papers/Levo4TR032002-0101.pdf.

[15] H.-S. Kim and J. E. Smith, "An Instruction Set Architecture and Microarchitecture for Instruction Level Distributed Processing," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*. Anchorage, Alaska, USA: ACM, May 25-29, 2002.

[16] A. Klauser, T. Austin, D. Grunwald, and B. Calder, "Dynamic Hammock Predication for Non-predicated Instruction Set Architectures," in *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*. Paris, France, October 1998, pp. 278-285.

[17] K. Krewell, "Intel's McKinley Comes Into View," *Cahners Microprocessor*, vol. 15, no. 10, pp. 1,5, October 2001.

[18] 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.

[19] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A Large, Fast Instruction Window for Tolerating Cache Misses," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*. Anchorage, Alaska, USA: ACM, May 25-29, 2002.

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

[21] 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.

[22] 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, URL: http://www.ele.uri.edu/~uht/papers/Levo3TR032002-0100.pdf.

[23] D. Morano, A. Khalafi, D. R. Kaeli, and A. K. Uht, "Implications of Register and Memory Temporal Locality for Distributed Microarchitectures," Dept. of Electrical and Computer

Engineering, Northeastern University, Boston, MA, USA, Technical Report, October 2002, URL: http://www.ece.neu.edu/groups/nucar/publications/intervals.pdf.

[24] D. Morano, A. Khalafi, D. R. Kaeli, and A. K. Uht, "Realizing High IPC Through a Scalable Memory-Latency Tolerant Multipath Microarchitecture," in *Proceedings of the Workshop On Chip Multiprocessors: Processor Architecture and Memory Hierarchy Related Issues (MEDEA2002), at PACT 2002*. Charlottesville, Virginia, USA, September 22, 2002. Also appears in ACM SIGARCH Computer Architecture Newsletter, March 2003, URL: http://www.ele.uri.edu/~uht/papers/MEDEA2002final.pdf.

[25] 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.

[26] A. Pajuelo, A. Gonzalez, and M. Valero, "Speculative Dynamic Vectorization," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*. Anchorage, Alaska, USA: ACM, May 25-29, 2002.

[27] D. B. Papworth, "Tuning the Pentium Pro Microarchitecture," *IEEE MICRO*, vol. 16, no. 2, pp. 8-15, April 1996.

[28] J.-M. Parcerisa, J. Sahuquillo, A. Gonzalez, and J. Duato, "Efficient Interconnects for Clustered Microarchitectures," in *Proceedings of the Eleventh International Conference on Parallel Architectures and Compilation Techniques*. Charlottesville, Virginia, USA: IEEE, September 22-25, 2002.

[29] I. Park, M. D. Powell, and T. N. Vijaykumar, "Reducing Register Ports for Higher Speed and Lower Energy," in *Proceedings of the 35th Annual International Symposium on Microarchitecture*. Istanbul, Turkey: IEEE, ACM, November 2002.

[30] V. Popescu, M. Schultz, J. Spracklen, G. Gibson, B. Lightner, and D. Isaman, "The Metaflow Architecture," *IEEE MICRO*, vol. 11, no. 3, June 1991.

[31] 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.

[32] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt, "A Scalable Instruction Queue Using Dependence Chains," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*. Anchorage, Alaska, USA: ACM, May 25-29, 2002.

[33] 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.

[34] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*. San Diego, California, USA: ACM and IEEE, June 9-11 2003.

[35] A. Seznec, E. Toullec, and O. Rochecouste, "Register Write Specialization Register Read Specialization: A Path to Complexity-Effective Wide-Issue Superscalar Processors," in *Proceedings of the 35th Annual International Symposium on Microarchitecture*. Istanbul, Turkey: IEEE, ACM, November 2002.

[36] B. J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer," *Society of Photo-optical Instrumentation Engineers*, no. 298, pp. 241-248, 1981.

[37] 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.

[38] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro*, vol. 22, no. 2, pp. 25-35, March-April 2002.

[39] G. S. Tjaden and M. J. Flynn, "Representation of Concurrency with Ordering Matrices," *IEEE Transactions on Computers*, vol. C-22, no. 8, pp. 752-761, August 1973.

[40] 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.

[41] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*: ACM, June 22-24 1995, pp. 392-403.

[42] A. Uht, A. Khalafi, D. Morano, M. d. Alba, and D. Kaeli, "Realizing High IPC Using Time-Tagged Resource Flow Computing," in *Proceedings of the Euro-Par 2002 Conference, Springer-Verlag Lecture Notes in Computer Science*. Paderborn, Germany: ACM, IFIP, August 28, 2002, pp. 490-499. URL: http://www.ele.uri.edu/~uht/papers/Euro-Par2002.ps.

[43] A. K. Uht, "Hardware Extraction of Low-Level Concurrency from Sequential Instruction Streams," PhD thesis, Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburgh, December 1985, 200 pages.

[44] 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 appears in the tutorial "Instruction-Level Parallel Processors", Torng, H.C., and Vassiliadis, S., Eds., IEEE Computer Society Press, 1995, pages 171-182.

[45] 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.

[46] 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.

[47] S. Wallace, B. Calder, and D. M. Tullsen, "Threaded Multiple Path Execution," in *25th Annual International Symposium on Computer Architecture*: ACM, June 1998, pp. 238-249.

[48] 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, URL: http://www.ele.uri.edu/~uht/papers/HDLevo.pdf.

[49] Y. Wu, R. Rakvic, L.-L. Chen, J. Fang, C.-C. Miao, and G. Chrysos, "Compiler Managed Micro-cache Bypassing for High Performance EPIC Processors," in *Proceedings of the 35th Annual International Symposium on Microarchitecture*. Istanbul, Turkey: IEEE, ACM, November 2002.