

Incremental Commit Groups for Non-Atomic Trace Processing

Matt T. Yourst Kanad Ghose
Department of Computer Science
State University of New York at Binghamton
{yourst, ghose}@cs.binghamton.edu

Abstract

We introduce techniques to support efficient non-atomic execution of very long traces on a new binary translation based, x86-64 compatible VLIW microprocessor. Incrementally committed long traces significantly reduce wasted computations on exception induced rollbacks by retaining the correctly committed parts of traces. We divide each scheduled trace into multiple *commit groups*; groups are committed to the architectural state after all instructions within and prior to each group complete without exceptions. Architectural state updates are only visible after future commit points are deferred using a simple hardware commit buffer. We employ a commit depth predictor to predict how many groups a trace will complete, thereby eliminating pipeline flushes on repeated rollbacks. Unlike atomic traces, we allow instructions to be freely scheduled across commit points throughout the trace to maximize ILP. Commit groups are formed *after* scheduling, allowing the commit points terminating each group to be inserted more optimally. Commit groups promote significantly faster convergence on optimized traces, since we salvage partially executed traces and splice the working parts together into new optimized traces. We use detailed models to demonstrate how commit groups substantially improve performance (on average, over 1.5× on SPEC 2000) relative to atomic traces.

Keywords: *binary translation, VLIW, commitment, trace prediction*

1. Introduction

Traditional out of order superscalar microprocessors have reached a performance plateau despite their rapidly escalating complexity, thus forcing computer architects to explore alternative designs. Binary translation (BT) is a key technique used to improve performance, reduce power and complexity and provide architectural compatibility. In

a BT system, code for a source ISA is transparently translated to a different *native* instruction set, typically composed of micro-operations (uops), where it is executed on hardware (usually a VLIW processor core) specially optimized to support BT transparent to the applications and operating system in the source ISA.

In BT systems, a *trace* is a commonly executed sequence of basic blocks identified through profiling techniques (software instrumentation or hardware support); traces in existing systems generally contain at most several hundred instructions. Once each trace is identified, the translated uops within it are scheduled and optimized into native VLIW instructions using scheduling techniques implemented as part of the BT system. Profiling data can be used to guide the scheduling process, for instance by respecting observed memory ordering relationships and verifying them at runtime. Scheduled and optimized traces are stored in a *translation cache* (TC) invisible to user code. With this arrangement, BT systems are always either interpreting and translating user instructions to native uops, or executing previously optimized traces from the TC. Since control never passes outside of this loop, the operating system and all programs perceive that they are executing directly on the bare hardware rather than through a BT system.

The BT system is often implemented as software running directly on the native machine. By far the most well known BT implementations are the Transmeta Crusoe [6] and Eficeon processors [8], which use a BT layer called *code morphing software* on top of a VLIW core. IBM's Daisy [9] and BOA [10] processors use similar approaches for implementing the PowerPC ISA. The Intel Pentium 4 [12] also uses elements of BT, relying on hardware to perform x86 to uop decoding, but still uses a traditional dynamically scheduled out of order execution core fed by a small on-die trace cache containing decoded but un-scheduled uops.

1.1. Atomic versus Non-Atomic Traces

Traces in traditional BT systems are *atomic*: the entire trace must execute without exceptions or mis-speculations

for the architectural state to be updated. Exceptions include actual program errors, memory speculation failures (e.g. load/store aliasing [1]) and failed control flow assertions [26] corresponding to mispredicted branches within the trace (by far the most common type of exception). Exceptions cause a *rollback* to the start of the trace, at which point the BT software typically interprets the corresponding user instructions sequentially until the exception is resolved. This lets the optimizer freely reorder memory and control flow uops within the trace at the expense of wasted cycles on rollbacks should exceptions occur. Existing approaches use a variety of techniques for checkpointing and rolling back both architectural registers [2] and memory [3, 21].

Atomic traces are limited in length in existing BT systems for two reasons. First, the BT system must amortize the computational cost of scheduling and optimizing traces over only the most frequently executed “hot” traces since the optimization overhead is so large. This means that it must ensure it does not waste effort on traces that will constantly roll back, so extensive profiling is needed before optimization even begins. Second, the rollback penalty in wasted cycles for long traces can be severe, and often requires slow sequential interpretation [6] or in-order execution [8] for recovery. For these reasons, traces in traditional systems are often limited to at most a hundred or so uops.

The use of short traces seriously limits the amount of ILP that can be found in a single thread of control. In this paper we introduce a mechanism called *incremental commit groups* to support efficient *non-atomic* execution of long traces in our BT-based processor. This allows a high degree of ILP to be harvested without the penalty of undoing a significant amount of useful work on exceptions. In our processor design, called Peptidal, traces consist of scheduled VLIW instructions, each instruction consisting of several uops, as in [6, 8, 9, 10]. However, after scheduling and optimizing very long traces of up to 1024 uops, *commit points* are inserted in accordance with the scheduled trace’s structure. On exceptions, the architectural state only needs to be rolled back only to the preceding commit point rather than to the start of the trace. The sequence of uops between consecutive commit points make up a *commit group*. Unlike most existing work on VLIW processors, uops from different groups can be hoisted and intermixed in the schedule; architectural state updates corresponding to future commit groups are deferred using special hardware structures. In essence, commit groups adopt from dynamically scheduled processors the idea of decoupling the execution order from the commit order, yet do so in a very fine grained way compatible with statically scheduled VLIW processors. This allows VLIW processors to implement true forward speculation across multiple levels of control flow.

In addition to safely allowing longer high ILP traces

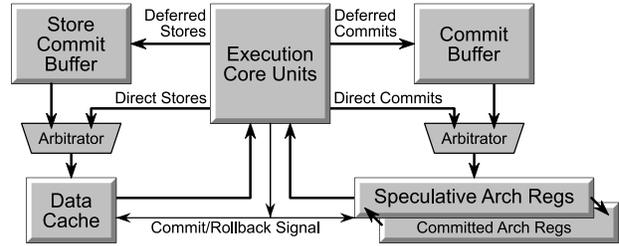


Figure 2. Peptidal Microarchitecture with Execution Core and Commit Buffers

without high rollback penalties, commit groups also promote significantly faster convergence on optimized traces. Our approach permits us to salvage partially executed traces and splice the working parts together into new optimized traces, rather than having to start from scratch after an atomic trace fully rolls back. In effect, this lets us naturally “evolve” the pool of active traces over time, similar to a genetic algorithm.

In this paper:

- We introduce a simple datapath component called the *commit buffer* to manage architectural state updates in extremely long non-atomic traces;
- We present an algorithm for inserting commit points into an existing VLIW schedule without impacting the schedule’s ILP or efficiency, and provide a way to minimize the number of commit points to reduce hardware complexity and avoid commit related pipeline stalls;
- We address trace prediction techniques which can eliminate the pipeline flush penalty from partially executed traces by predicting how many commit groups the trace will complete. In effect, this transforms rollbacks from events to avoid into a natural and predictable part of speculative execution.
- We demonstrate through comprehensive cycle accurate simulations that incremental commit groups provide a substantial performance benefit over atomic traces.

Since the primary goal of this paper is to show the advantages of non-atomic traces, for the sake of brevity, we do not discuss the details of trace generation, scheduling and optimization, nor do we describe the other features of the Peptidal microarchitecture not related to non-atomic traces. Further details and common techniques for these steps are discussed extensively in the literature [6, 7, 10, 9, 11].

2. Commit Groups

Peptidal is our experimental 64-bit microprocessor, designed to execute the full x86 and x86-64 instruction set

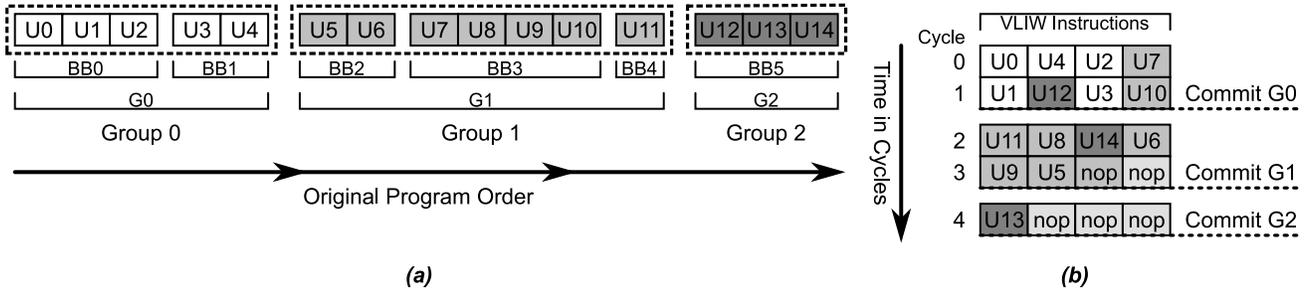


Figure 1. Example trace divided into basic blocks and groups; scheduled form with commit points inserted.

through binary translation technology similar to [6, 7, 8, 10, 9]. *PTL* (Peptidal Translation Layer), akin to the code morphing software used in Transmeta processors, is our combination of native software and specialized but simple hardware responsible for translating, scheduling and optimizing traces of native Peptidal uops. Peptidal is internally an 8-issue VLIW like microprocessor capable of issuing two complex integer uops, two loads (or ALU), two stores (or ALU) and two floating point uops per cycle. The processor has a 256-entry physical register file, and maintains the architectural state in a separate 64-entry architectural register file. Peptidal allows traces of up to 1024 uops, 64 branches and 256 loads and stores to be in flight simultaneously. *PTL* applies a wide range of optimizations comparable in scope to [6, 7, 8, 10, 9] at several different optimization levels depending on trace execution frequency. The pipeline and clustered bypass network is designed for extremely high clock rates while still allowing high ILP, however these details are beyond the scope of this paper. The key elements of our microarchitecture are shown in Figure 2.

In our incremental commitment model, traces are composed of multiple *commit groups*. Each commit group consists of an integral number of *basic blocks*, with each basic block terminated by either a branch (conditional, unconditional or indirect) or by reaching internal length limits.

Figure 1(a) shows an example sequence of 15 uops in a single trace. Each uop is labeled U_n , where n is a sequential tag number denoting the original program order of the uop. The uops are grouped into 6 basic blocks BB0-BB5, with each basic block terminating at uops U2, U4, U6, U10, U11, U14, respectively. Furthermore, these 6 basic blocks have been placed into 3 commit groups, denoted as groups G0, G1, G2, respectively. Basic blocks BB0 and BB1 have been placed into group G0, basic blocks BB2, BB3, BB4 are in group G1, and basic block BB5 is alone in group G2. The commit group to which a given uop belongs is termed its *home commit group*. Figure 1(b) depicts a typical schedule into which the uops of Figure 1(a) have been placed. The uops have been rearranged out of program order into VLIW instructions so as to satisfy dependencies amongst the uops. Notice that each VLIW instruction may contain a mixture

of uops from different home commit groups.

The *commit point* C for a given commit group G is the final cycle in which any uop within G (or by extension, any groups in program order before G) appear within the schedule. In Figure 1(b), for commit group G0, the latest uop to execute is U3, which executes in cycle 1. Therefore, the commit point for commit group G0 occurs at cycle 1 as shown. This means that after the VLIW instruction executed in cycle 1 has completed, the results of all uops within and before group G0 have been committed to the user visible architectural state. There may also be unrelated uops for commit groups after G0 speculatively executes in the same cycle, such as U12 in group G2 and U10 in group G1. Additional commit points appear after cycles 3 and 4, for groups G1 and G2, respectively. *Notice that commit groups always complete in their original program order, even if the constituent uops execute completely out of program order.* The arrangement of basic blocks into commit groups is done such that only one commit group completes per cycle; if two or more basic blocks would complete on the same cycle, they would be collapsed into a single commit group. Algorithms for merging basic blocks into commit groups and inserting commit points at specific cycles are discussed later in Section 5.

At any given time, the processor is executing instructions within a specific *currently executing commit group*: the next group in program order awaiting commitment. For instance, in Figure 1(b), the currently executing commit group during the first two cycles is commit group G0. After control successfully passes the commit point for group G0, G1 becomes the currently executing commit group. If any uop within the currently executing commit group causes an exception, including branch mispredictions and memory ordering or aliasing violations, the entire architectural state must be rolled back to the architectural state that existed at the end of the last successfully committed group (rather than the start of the entire trace as in existing systems). It should be noted that uops belonging to *future* commit groups do *not* immediately cause a rollback; instead their results are just tagged as invalid such that once execution reaches their home commit group, a rollback will immediately occur.

3. Hardware Facilities

The incremental commit group concept described above allows updates to the architectural state within a given trace to be classified by commit group, since we may need to recover a specific *version* of the state (i.e., at the completion of a given commit group and by extension all groups before it), rather than the all-or-nothing approach of atomic traces. To achieve this goal for register commits, we use a *register commit buffer* comprising of a fixed number of slots (128 slots in the current design), similar to a register file as shown in Figure 2. For any unique architectural destination A written with value(s) during commit group G , exactly one commit buffer slot S will be allocated to the uop generating the corresponding value; slot S may be deallocated and reused after the commit point for G has passed. Slot S will only be written by the final uop in program order to write to A within group G ; all other uops earlier in program order also targeting destination A within group G will not be allocated a commit buffer slot. In the most general form of the commit buffer, the architectural destination may address any component of the architectural state, including architectural registers or memory addresses for stores. Section 4 describes how stores are handled by a structure similar to the register commit buffer.

The register commit buffer is used to defer speculatively generated architectural register updates until the home group in which the results were generated becomes the currently executing commit group. In our implementation, the register commit buffer consists of 128 slots divided into two banks. Each slot in the commit buffer has four fields: R (1 bit), C (1 bit), Reg (6 bits) and $Data$ (64 bits): The Reg field identifies the architectural register that commit buffer is intended to eventually update, while the $Data$ field contains the data to write. The R (ready) and C (commit) bits together indicate which of four states a given slot is in, as described in Figure 3.

The hardware allows between one and a finite maximum number of commit groups to exist per trace; our implementation allows up to 16 groups. To separate which slots belong to which groups at runtime, our processor maintains 16 *membership masks* $M[G]$, one per commit group G . Each membership mask is a 128 bit vector, corresponding to the number of commit buffer slots. If a given uop U is encoded to commit to commit buffer slot S during group G , bit S of the membership mask $M[G]$ is set at the time U executes. In this way, the set of slots belonging to each commit group is dynamically constructed and identified without the need for the hardware to support associative search using the currently executing group number.

Just as in other VLIW processors, physical registers store live results for later consumption within the same trace. It should be noted that instructions never read back values

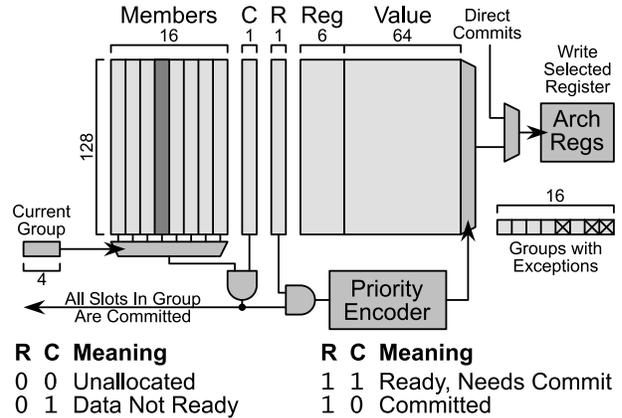


Figure 3. Commit Buffer Bit Vectors and Meanings

from the commit buffer; all information flow is one way. At the end of each trace, the 64 committed architectural registers are atomically latched into the upper 64 physical registers to propagate live-ins to the next trace. If a VLIW instruction in a subsequent trace needs a value generated in the current trace that has not yet been propagated to the architectural registers, the pipeline may stall until the value is ready; scoreboard logic on the architectural register file is used to enforce this constraint. Fortunately, our commit group formation and scheduling techniques are specifically designed to avoid such stalls. This means that as committed values in the last commit group of the current trace fall through into the speculative architectural register file (see Section 3.2), they become instantly available to consumers in the first groups of the next trace.

3.1. Commit Buffer State Machine

As shown in Figure 3, each commit buffer slot can be in one of four states defined by its R and C bits. When a given uop U targeting slot S first attempts to commit, it updates the membership mask and then sets the C bit of the corresponding slot, since that slot still must eventually commit. If the data is immediately ready, the R bit is also set. Otherwise the slot remains in the $R=0, C=1$ state. Every cycle, the processor computes the bitwise AND of the three 128-bit vectors $M[G], C, R$ over all commit buffer slots in parallel to determine which slots are eligible to commit within the currently executing commit group G . Based on this combined bit vector, the hardware selects up to 4 register results per clock for commitment. Any selected (register, data) pairs are then sent to the speculative version of the architectural register file for commitment. The corresponding slot's C bit is then cleared so that it is not committed twice.

When the processor encounters the commit point for group G (i.e. the commit bit is set on the final VLIW instruction in G), it must check that all commit buffer slots

belonging to group G are already committed to the architectural state. This is done by computing the bitwise AND of 128-bit vectors $M[G]$ and C , if the result is zero, all required results have been committed and the speculative architectural state is copied to the user visible committed state. Otherwise, the commit unit stalls the commit stages of the pipeline until all slots in G have been committed. Stalls can happen for two reasons: either some slots are not yet ready (e.g. because of load misses) or there was simply not enough commit bandwidth to move results from the commit buffer into the architectural registers. Once the stall condition is resolved, all entries belonging to the current group are reset to the unallocated state. Finally, the currently executing commit group number is incremented to the next group in sequence.

An additional 16-bit exception vector is maintained to indicate which groups contained at least one uop causing an exception. Since uops from different groups can be intermixed in the schedule, this vector is needed such that execution does not proceed into a future group already marked invalid; instead the trace is immediately exited without a rollback if such an invalid group is entered after leaving the last correctly committed group.

3.2. Direct Commitment

If the home commit group of a given uop U happens to be the currently executing commit group (i.e., U was not scheduled early in the trace prior to entering its home group G), there is no need to send U 's result to the commit buffer or allocate a commit buffer slot for it, since it will immediately fall through into the speculative architectural register file during the currently executing group. In this situation, U 's result bypasses the commit buffer and is directly written into the architectural register file in what we call a "direct commit". To accomplish this, any uops meeting the previous condition are encoded to perform a direct commit. An arbitrator balances the flow of deferred commits from the commit buffer against direct commits to the architectural registers within the four write port limit. The direct commit mechanism can never be used with variable latency uops (e.g. loads), even in the currently executing group, since the speculative architectural register file itself has no means of being updated later when the loaded data actually arrives; such uops are always assigned to a commit buffer slot.

3.3. Shadowed Register File

Values in the commit buffer constantly "fall through" into the speculative architectural register file in arbitrary order whenever the currently executing commit group matches the home commit group of slots in the commit buffer, or when direct commits are used as described in Section 3.2. Since this occurs continuously before it is known

if all uops within the current group will complete without exceptions, the architectural state saved at the start of the current group must be readily available for state restoration in case of a rollback. To accomplish this, we use the well known shadow bitcell approach [2] to hold this saved state in an alternate version of the architectural register file. As each commit point is successfully passed (i.e. all commit buffer slots and direct commits in the current group are now in the speculative architectural register file), all bitcells in the speculative architectural register file are atomically latched into the committed shadow bitcells, overwriting the last known good state in a single cycle. As described in [24, 2], shadow bitcells do not materially impact the complexity or access time of the multi-ported register file. Whenever a rollback is required, all shadow bitcells are latched back into the architectural register bitcells in a single cycle, restoring the precise state. Rollbacks also cause the entire commit buffer to be reset, since a new trace must be started.

If the final branch of any commit group (including the last branch in the trace) mispredicts, it is not necessary to force an exception and rollback of the group containing such a branch. Technically, all results computed before and including the branch itself are still correct, so the only action needed is to redirect instruction fetching to the off-trace destination actually specified by the branch. This is obviously not possible with non-final branches embedded within groups, since such branches are in effect control flow assertions [26], not regular branches. For mispredicted embedded branches, uops after the excepting branch in program order may have corrupted the speculative state and hence will require a rollback.

3.4. Recovery from Rollbacks

Even in traces composed of incremental commit groups, exceptions may still occur in the first group of the trace, causing a full rollback just as in an atomic trace. To recover, we must execute another trace (the *recovery trace*) starting at the same user architectural program counter, known as the *instruction pointer (IP)* in x86-64, as recorded in the last known good architectural state. Obviously executing the same trace that caused the exception would invoke an infinite loop, so instead we begin recovery at the single basic block sized trace starting at the last committed IP; this trace also has all loads and stores in their original program order. This conservative recovery policy is needed since such basic block traces are not capable of causing either memory ordering exceptions nor branch mispredicts.

4. Committing Stores

The *store commit buffer* is used in a manner nearly identical to the register commit buffer described in Section 3;

unlike the register commit buffer, the architectural destination targeted by each store commit buffer slot is a memory address, not an architectural register, and store commit buffer slots matching the currently executing commit group are written into a speculative version of the data cache instead of a register file.

As described in Section 3, only the final operation to write to a given architectural destination within a given commit group updates its assigned commit buffer slot. This same principle applies to the store commit buffer, wherein the architectural destinations are physical memory addresses. Unlike operations targeting architectural registers, the specific addresses targeted by a given store are not generally known until execution time, so it may not always be possible to know ahead of time which store is the last store in program order within a given group to target a specific memory location.

Peptidal uses profiling techniques similar to those used to eliminate redundant stores [1, 14, 13, 15, 24] to speculatively identify which stores are final within each group. Simple set associative hardware structures are then used to verify this speculation at runtime; any later store which touches the same memory location already written by a predicted final store within a group will cause an exception and the trace must be rescheduled. While further details are beyond the scope of this paper, we have developed mechanisms and heuristics to ensure extremely accurate final store prediction. When mismatches do occur, the commit group mechanism ensures a minimal amount of work is wasted.

In our implementation, the store commit buffer consists of 64 entries; the buffer can accept two stores from the core and commit two stores to the cache every cycle. Each store commit buffer entry is similar to a register commit buffer entry (Section 3), but with several key differences. The C (committed) bit operates the same as with the register commit buffer; i.e. it is set for entries still waiting to be committed. Notice that there is no R (ready) bit, since stores are always ready by the time they are executed. Additionally, each entry contains the 64-bit aligned data to be stored and the target physical address, similar to a store queue in an out of order processor.

The process of moving stores from the commit buffer into the data cache is also very simple. Similar to how the register commit buffer maintains group membership masks (see Section 3), the store commit buffer is associated with a separate set $M[G]$ of 16 64-bit masks. Assuming we're currently executing in commit group G , the bitwise AND of the 64-bit vectors ($M[G]$ and C) is computed, and the store commit buffer slot S corresponding to the first set bit of ($M[G]$ and C) is selected as the store to commit. Slot S is then committed to the data cache according to its address and data. Direct stores are also possible when a given store's home commit group matches the current commit

group; these are handled in exactly the same way as with the register commit buffer (Section 3.2). Like the register commit buffer and shadow architectural register file, the L2 cache is transactional so as to allow commits and rollbacks. This is accomplished using a locked and dirty line scheme as in [21, 8].

5. Commit Group Formation

The commit group formation algorithm is responsible for arranging basic blocks into commit groups and determining the cycles in the schedule where commit points should be inserted. It should be noted that commit group formation runs *after* all uops are scheduled, meaning that it is trivial to add on top of a traditional atomic trace scheduler, and in no way impacts the level of ILP the scheduler can produce.

PTL, our binary translation system, uses the following algorithm to insert commit points into the VLIW schedule. First, as each uop U in basic block B is scheduled in cycle C , we maintain an array F (the final cycle array) to track the final cycle in which any uop in each basic block B is scheduled, where $F[B] = \max(F[B], C)$ for each uop. After all uops are scheduled, we generate a *collapse map*, with one bit for each basic block to record which basic blocks terminate their own commit group. If a commit point occurs after basic block B , $\text{collapse}[B]$ is set; otherwise it is clear. If two bits a and b are set in the collapse bitmap, this means that all basic block numbers between $a+1$ and b (both inclusive) will belong to the same commit group.

Our algorithm repeatedly processes the final cycle array F , each time starting at the branch B currently being examined. The algorithm finds any basic blocks later in program order than B but with all instructions completed before or during the cycle $F[B]$ in which basic block B completed. These basic blocks are then merged into the same commit group as B , since they will be complete by the time B completes. The inner loop stops at the first basic block that does not satisfy this condition (i.e. it completes later than $F[B]$ and can therefore not possibly fit in B 's commit group). We then skip to the basic block after the end of the commit group the algorithm just established and continue with a new commit group. This repeats until all basic blocks (up to 64 per trace) are assigned to a commit group.

Given only a collapse map, it is now trivial to construct a mapping from basic blocks to their corresponding commit group by just iterating through the bits in the collapse map to generate a mapping array. If basic blocks B_1, B_2, \dots, B_n all belong to group G , the cycle C in which we should insert a commit point for group G is simply the highest cycle in which any of basic blocks B_1, B_2, \dots, B_n complete, i.e., $\max(F[B_1], F[B_2], \dots, F[B_n])$. Similarly, the cycles in which commit points should be inserted is easily derived from the array F above.

Since the hardware only allows up to 16 commit groups per trace even though each trace may encompass up to 64 branches, we need a mechanism for removing some commit points from very long traces to satisfy the hardware constraints. In our implementation, this is done by considering overlapping groups of three commit points C_1, C_2, C_3 and finding the distance in cycles between C_3 and C_1 . This list of distances is then sorted and the commit point C_2 between the most closely spaced pair of adjoining commit points C_1, C_3 is removed. This process is repeated until the number of commit points is at most 16. Notice that the trace need not be rescheduled after re-assigning commit points.

6. Commit Slot Assignment

As described previously, each uop U within commit group G targeting a given architectural destination A must only write to the commit buffer (or directly to the speculative architectural state if a direct commit is possible) if U is the final uop in program order within group G to write architectural destination A . For register to register and load uops, A is an architectural register number. For stores, A is a physical memory address.

The mechanism used to determine the final writers in this manner is a variant of that used for atomic traces and is similar in concept to future files [28]. To determine which uops meet the conditions above, we maintain a two dimensional matrix $M[G][A]$ to map each commit group G and architectural destination A to the final uop U to write to A during group G . In each case, the uops are processed in their original program order, and for each uop U in group G targeting A , we set $M[G][A] = U$. After all uops are processed, the final writer to each destination in each group has been identified.

At the very end of the trace generation process, uops in scheduled VLIW instructions are saved to the translation cache. As this occurs for each uop U (in group G writing architectural destination A) in scheduled order, U is encoded to commit if and only if U matches $M[G][A]$. If U is marked to commit and is scheduled to execute before its home group is reached, it is assigned a commit buffer slot in either the register commit buffer or the store commit buffer. If it was scheduled within its home group, direct commitment is possible and no commit buffer slot is allocated, except for variable latency operations like loads as explained in Section 3.2.

Commit buffer slots are assigned to any uops needing to defer commitment into the future if and when their home group becomes the currently executing group. Slots are allocated from either the register commit buffer or the store commit buffer, depending on the type of uop. After the last VLIW instruction at the commit point for a given group G is written out, all slots allocated to G can be freed and

reused as needed. The assignment algorithm currently used by PTL maintains free slot bitmaps and assigns slots in a round robin fashion as needed by uops in scheduled order. It should be noted that slots can be assigned anywhere in the commit buffer, unlike a reorder buffer (ROB), which limits the instruction window size by forcing ROB slot assignment in program order. Another assignment policy is used by PTL only for traces with fewer than four commit groups containing commits to at most 32 unique registers: commit buffer slots 0-31 are assigned to group 0, slots 32-63 belong to group 1, and so on.

7. Trace Prediction

In this section, we describe two prediction mechanisms associated with trace execution and a technique for super-trace formation.

7.1. Next Trace Prediction

In any trace based processor, there must be a mechanism for predicting the next trace to fetch and execute before the current trace's exit point is known. The Peptidal mechanism for predicting the next trace to execute is very similar to that of [27]. We maintain a shifting history buffer of the last 4 traces, where each entry contains the PTL internal entrypoint of the trace. To form a prediction, we hash the the history to index a slot S_0 in a predictor table (4096 entries in the implementation described). We then add the currently executing trace to the history buffer, and rehash the updated history to obtain and access the new table index S_1 . Each predictor table entry contains the PTL internal entrypoint of the predicted next trace, so uop fetching can proceed immediately after a prediction. This is most similar to systems like [6, 8, 9] in that the instruction caches are addressed using physical addresses inside the PTL translation cache.

The prediction is considered correct if and only if the predicted trace's IP matches the IP the previous trace actually finished with. This may vary depending on how many commit groups completed and the direction or target of the final branch in each group. If the prediction was incorrect, the processor takes a small flush penalty (5 cycles in our implementation) to refill its frontend pipeline with the correct trace. The predictor is updated after a trace completes; this is done by overwriting predictor table slot S_0 (which lead us to the currently executing trace) with the trace that will actually execute next. The processor also maintains a *replacement map*, a small 4-entry fully associative array used to map one trace to a substitute trace. This mechanism is used when two traces A and B are merged into trace AB : the replacement map is updated to map any predictions for trace A to the new trace AB . Using this mechanism, newly scheduled traces can be quickly installed into the predictor.

7.2. Commit Depth Prediction

In many cases, rollbacks to a specific commit point in a trace are so common that it makes no sense to continue fetching VLIW instructions beyond that commit point; instead, we should start fetching from the recovery trace (Section 3.4) that would be executed once the rollback occurs. Without this capability, mispredicted branches and other excepting operations would not only waste however many cycles into the commit group are required to encounter the exception, but would then incur an additional 5 cycles to refill the core pipeline with the correct trace. In effect, this technique transforms rollbacks from events to avoid into a natural and predictable part of speculative execution.

To achieve this goal, our design uses a *commit depth predictor* to complement the next trace predictor. The commit depth predictor predicts how many commit points the selected trace will correctly commit in sequential order before encountering a possible rollback and subsequent transfer to another trace. If the trace is predicted to only pass n commit points, as soon as the VLIW instruction signaling commit point n enters the pipeline, fetching is redirected to an alternate trace on the next cycle. If the predicted rollback never occurs and the fetch queue for the current trace becomes empty, the 5-cycle frontend pipeline flush stall is required to discard the next trace and resume the current trace. Otherwise, if the rollback *does* occur where expected or the full trace commits normally, the correct next trace would already be in the pipeline ready to execute. The commit depth predictor is a 1024-entry 2-way set associative table, with each slot containing a 4-bit predicted commit depth. As mentioned above, the commit depth predictor is accessed *after* the next trace predictor has selected a trace. The selected trace's PTL entry point is used to index into the commit depth predictor and read out a predicted 4-bit commit depth to be used.

After a given trace executes, we update its commit depth predictor slot with the number of commit points it actually passed; this depth will then be the predicted depth the next time the trace is executed. Overall, this simplistic scheme works well, since once a trace starts rolling back at a given commit point, this behavior generally repeats for subsequent executions until program conditions change. For instance, consider the case of very long traces for nested unrolled loops. Typically the trace forming the inner loop will commit to the same depth every iteration (and hence is very predictable) until the outer loop enters its next iteration, where the commit depth behavior generally changes.

7.3. Supertrace Detection

To construct long traces, Peptidal uses a combination of hardware counters and software instrumentation to profile which basic blocks and traces repeatedly execute in the

same sequence. In our implementation, if the same pattern of two traces (and the number of groups each successfully completes) occurs often enough to exceed a threshold, the two traces are merged and rescheduled by PTL into a longer trace. PTL employs a variety of scheduling algorithms, ranging from classical reservation table based greedy scheduling, to critical path scheduling for the most frequently executed traces; these and other heuristics and algorithms for merging traces in BT systems are similar to those described in [11, 9, 10, 7, 25]. Statistical threshold models similar to [11, 25, 10, 9] are used to balance the potential ILP gains of forming longer traces against the cost of optimizing long traces.

8. Experimental Evaluation

To evaluate the Peptidal microprocessor design, we used PTLsim, an advanced cycle accurate simulation infrastructure capable of supporting the full x86 and x86-64 instruction sets. In the following sections, we analyze key benchmarks from the SPEC 2000 suite to illustrate specific points. Each benchmark was examined and a span of code between two representative points in the code (e.g. one iteration of the main loop) was selected for profiling. Most spans comprise roughly 2 billion x86-64 instructions per benchmark. All benchmarks were compiled using gcc 3.4.3 with maximum optimizations to target the 64-bit x86-64 instruction set. Since one of the main innovations in Peptidal is the use of non-atomic commit group based traces, we will compare the performance of atomic versus non-atomic traces.

Figure 4 and the corresponding rows in Table 1 shows how execution time is spent in each benchmark for both atomic and commit group based traces.

In Figure 4, we show a pair of stacked bars for each benchmark, with the left bar corresponding to the overall execution time of traditional atomic execution (100%) and the right bar corresponding to our non-atomic trace execution using commit groups. The right stack, for commit group traces, is shorter since with commit groups enabled, the benchmarks take less total time to execute, relative to the number of baseline cycles required. Each stack is broken down into several sub-bars to show the cycle distribution. *Stalls* indicates the percentage of cycles (always relative to the baseline cycle count) the processor is stalled waiting on missed loads. *Flushes* indicates cycles wasted flushing the pipeline after a trace misprediction. *Commit* shows the time spent stalled at commit points, either waiting for missed loads to arrive or for the commit buffer to flush itself (these bars are difficult to see since commit stalls are such a minor part of the execution time). *Rollbacks* indicates the percentage of cycles wasted executing instructions which will eventually be rolled back. Finally, the *Execution* bars, accounting for all the remaining time, represent cy-

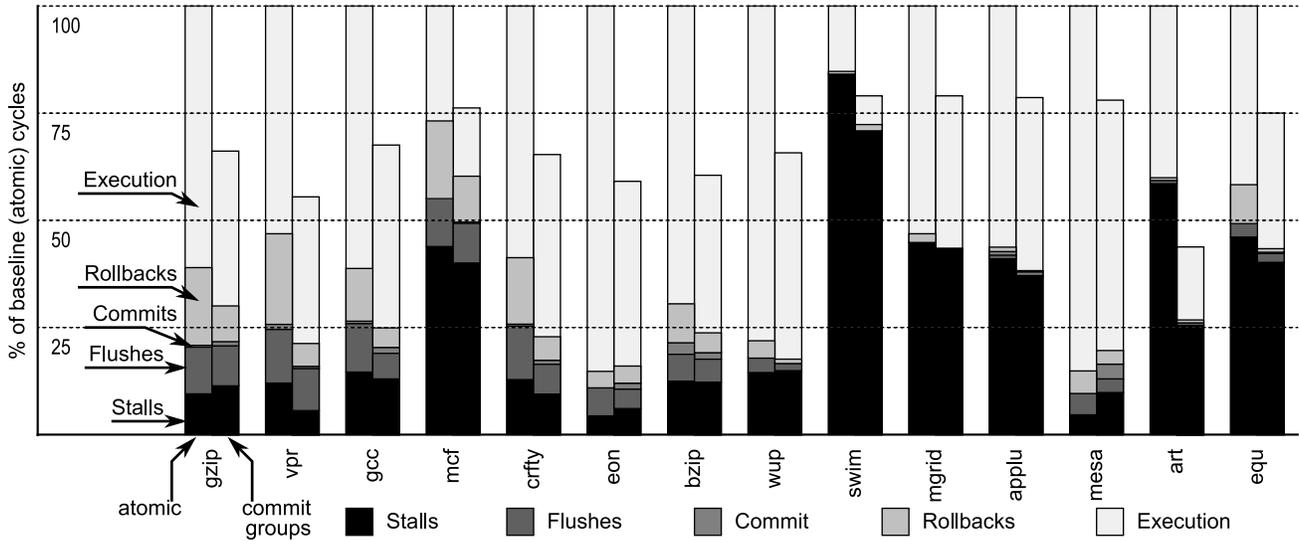


Figure 4. Distribution of Cycles by Benchmark. Table 1 also provides a numerical summary.

	Average		Integer		Float	
	AT	CG	AT	CG	AT	CG
x86 Insns	22.4		7.8		14.5	
Cycles	20.6	13.4	8.2	5.0	12.4	8.3
CG speedup	1.45×		1.6×		1.5×	
Stall %	29	22	16	13	42	32
Flush %	5.8	4	10	6.8	1.7	0.8
Commit %	0.5	1.0	0.3	1.0	0.8	0.9
Rollback %	9	3	14	5.7	3.2	1.0
Opt %	39	87	24	78	54	95
Pred %	86	71	80	56	91	86
Length	40	126	10	31	69	221
Compl %	95	75	92	66	98	83

Table 1. Comparison of atomic traces (AT) versus commit group (CG) based traces, separated by benchmark type.

- Cycles:** Total cycles to complete benchmark (billions)
- x86 Insns:** Total x86 instructions committed (billions)
- Stall %:** cycles spent stalled on missed loads (relative to total atomic trace cycles)
- Flush %:** cycles wasted flushing pipeline after mispredicts
- Commit %:** cycles stalled waiting at commit points
- Rollback %:** cycles wasted executing rolled back uops
- Opt %:** Percent of all cycles spent in optimized traces
- Pred %:** Trace predictor accuracy (both commit depth and target IP match)
- Length:** Average committed trace length in uops
- Compl %:** Percent of all traces which execute to completion

cles spent making forward progress in the program. Table 1 gives the equivalent numerical values, averaged across the benchmarks.

By examining the difference in total cycles between the atomic and commit group bars in Figure 4, it is evident how non-atomic commit group based traces hold a significant performance advantage over atomic traces (on average 1.5×), particularly in branch heavy integer benchmarks (which are typically the most difficult type to optimize). Notice how the use of commit group traces cuts the average rollback penalty by over 65% (from 9% of the total cycle count with atomic traces compared to 3% with commit group traces), as shown by the *Rollback* bars and Table 1.

One key performance metric is the percentage of cycles spent executing fully optimized traces. As Table 1 shows, with commit groups almost 90% of the time is spent in fully optimized traces over the first 1 billion cycles of each benchmark. In contrast, only around 30-50% of the time in the atomic trace case is spent in optimized traces. This indicates that the traditional supertrace formation heuristics are unable to find most sequences of traces profitable enough to reschedule into a longer atomic trace; basic block sized traces are used the remaining time. As stated in Section 3.4, traces with a single basic block are also used for recovery when rollbacks occur. It was mentioned above that additional threshold tuning might allow more atomic traces to be merged and optimized, however it is fairly clear that by design, non-atomic commit group traces would still hold an advantage, particularly during startup, in any applications where branches remain difficult to predict.

One of the key advantages of commit groups is that the technique is essentially free from a performance standpoint:

even if a given trace always commits in full, the commit points can remain inserted without impacting performance in most cases. However, with commit groups, Table 1 shows how more time is spent stalled at commit points waiting for outstanding loads to commit (1.0%) than with atomic traces (0.5%), especially for integer benchmarks with many closely spaced commit points. If commit points are spaced too closely in such traces, artificial stalls may be incurred as the pipeline waits for loads to resolve solely so they can be committed, even if the first computational consumer of the load lies much later in the schedule. Fortunately we did not find commit related stalls to be a significant performance degradation (both commit group and atomic traces waste less than ~1% on such stalls). Nonetheless, PTL includes heuristics to find traces which experience a very large number of commit stalls. These traces are converted to atomic form by removing commit points. This is very fast, since as described in Section 5, no rescheduling is needed to adjust commit point locations. Having too many groups per trace may also limit optimization opportunities. In atomic traces, redundant computations and stores may be eliminated if they do not contribute to the set of results eventually committed to the architectural state. With commit groups, such dead code elimination must respect the live-outs at each commit point, not just the trace end, and unsafe speculations must insert check uops before the end of each corresponding commit group. This allows a tradeoff between the degree of optimization and rollback agility.

The accuracy of the next trace predictor must also be considered. Table 1 shows that trace prediction accuracy is very high (around 80-90%) with atomic traces, which is to be expected since atomic traces are formed very conservatively only when branch paths are known with very high certainty. In contrast, prediction accuracy is much lower (on average 60-70%) with commit group traces, given the high amount of variability in correctly predicting where a given trace will exit. As described in Section 7.1, commit depth prediction is used to help offset this penalty, resulting in 30% fewer cycles wasted with commit groups than in atomic traces. Nonetheless, trace prediction in the context of commit group based traces is a much more difficult problem than atomic trace prediction; as such it is a focus of our future work. It should be noted that the supertrace formation heuristics in PTL were explicitly tuned to the assumption that the vast majority of traces would at least partially commit. With atomic traces, different heuristics might be more appropriate. For instance, less aggressive trace merging (using higher thresholds) might let the optimizer avoid excessive rollbacks until it is absolutely sure that two traces should be merged. Additionally, the next trace predictor could be better tuned to handle the all-or-nothing approach of atomic traces. Since atomic traces were not the focus of this study, we have presented the results as is, by sim-

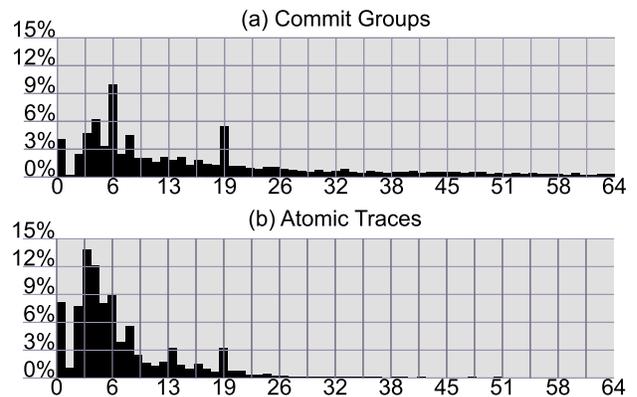


Figure 5. Histogram of commit depths (in uops) for *gcc* between commit groups and atomic traces

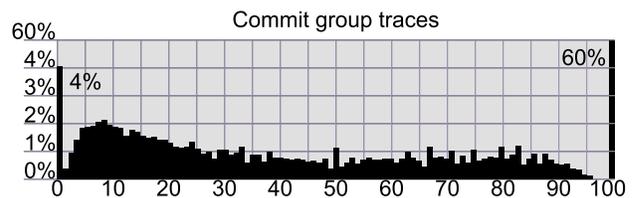


Figure 6. Histogram of commit depths (in percent of all uops in trace) for *gcc* with commit groups. Atomic traces fully completed 92% of the time and fully rolled back 8% of the time.

ply forcing all traces to have only one commit group. Even with a perfectly tuned atomic trace implementation, commit groups still have an advantage, particularly at program startup, when partially executed traces can be spliced together to ramp up performance very quickly.

We also observed that out of all commits to architectural registers, on average 70% were deferred via the commit buffer, while only 30% of the results committed directly to the architectural register file within their home group. However, for stores, only 25% of all committing stores were deferred in the store commit buffer; the remaining 75% committed directly to the data cache within their home group. This suggests that the commit buffer mechanism is far more useful with architectural registers, at least given the instruction mix in typical x86 programs.

Figures 5 and 6 examines one well known representative benchmark, *gcc*. Notice the distribution of committed trace lengths in graphs 5(a) and 5(b): with commit groups, on average around 38 uops commit per trace, whereas with atomic traces, the average is half that, at around 15. In graphs 6(c) and 6(d), notice that only 60% of traces execute to completion; the remaining 36% exit at various com-

mit points throughout the trace, indicating that the commit group mechanism is working exactly as designed. With commit groups, only around 4% of traces encounter a full rollback, while 8% of atomic traces roll back, despite being much shorter.

As mentioned in the introduction, IBM's BOA architecture [10] provides some interesting statistics for an alternative binary translation system based on atomic traces, albeit for PowerPC rather than x86. In [10], on the SPEC95 integer benchmarks it was shown that only 70% of traces run to completion; all others (30%) encounter a full rollback, wasting significant work. With Peptidal, on average only 5-10% of all traces encounter a full rollback; 90-95% make forward progress thanks to our commit group mechanism. Table 1 provides additional details here: notice how on average only 70% of all traces fully execute with commit groups, compared to a 90% completion rate for atomic traces. This is to be expected, since commit group traces can take side exits with no penalties.

Also in [10], the average committed trace length on the same benchmarks was on average around 30-40 PowerPC instructions. Even though this is not directly comparable to our trace lengths in Peptidal uops (as shown in Table 1), it does indicate that commit groups still allow us to form long traces on difficult integer benchmarks. Specifically, Table 1 shows that with commit groups, the *average* committed trace length increases dramatically relative to the atomic case, from 10 to 31 uops for integer and 68 to 221 uops for floating point. The average committed trace length varies widely across benchmarks; for instance, in *mgrid*, the average length was 480 uops.

9. Related Work and Conclusions

The basic concept of inserting multiple commit points into a flow of instructions has been proposed before, but in the context of dynamically scheduled out of order processors. In [22, 23, 21, 24], out of order commitment is supported with structures similar to [2, 3] by creating regularly spaced checkpoints in the instruction stream. As with commit groups, checkpoints are only committed to the architectural state after all instructions in program order before the checkpoint have been completed. However, these methods only work with dynamically scheduled processors, and hence inherit all the problems of such processors BT systems were designed to overcome [6, 11, 10]. Checkpoints in the previously mentioned dynamically scheduled processors are inserted at low confidence branches, whereas we rely on trace formation to filter out low confidence branches. We also insert zero-overhead checkpoints where they naturally fall *after* scheduling is complete, rather than doing so sub-optimally before the final schedule is known. Finally, we utilize our commit depth predictor to further reduce the

pipeline flush penalty of rollbacks compared to the previous work.

Boosting [16] is a superficially similar technique in which loads are hoisted earlier across branches and can be selectively annulled when branches mispredict. However, boosting is limited to a few branches, while commit groups can be of any size with a commit buffer hardware complexity independent of the number of branches, even with 64 branches per trace as we allow. Additionally, boosting only supports branches, while we allow arbitrary speculation (load/store ordering, value prediction, etc). Finally, our commit buffer is much simpler and does not involve counters like boosting.

In [17], Rau proposes a technique for maintaining binary compatibility of VLIW code across implementations. A delay register file (DRF) with scoreboarding logic is used to hold results generated out-of-order. These results are written back to the main register file in the original VLIW program order like a traditional reorder buffer. The DRF thus appears to be similar to our commit buffer but there are notable differences. Commitment from the DRF is fine-grained and incremental. Our commit buffer commits a single commit group (or a set of collapsed basic blocks) in a single atomic operation; this allows much more aggressive optimization in the context of a binary translation based processor. In [17], explicit instructions ("phase 2 instructions") are dynamically scheduled to move instructions from the DRF into the main register file. Our commit buffer mechanism commits missed loads asynchronously as they arrive without stalling the processor.

In [20], a variety of techniques for handling interrupts in VLIW processors are discussed, including the use of replay buffers that use a separate thread for interrupt handling, and the self-draining approach, where execution is stalled on an interrupt and all partially executed instructions are allowed to complete before interrupt processing takes place. Doing this avoids resource contention with the interrupt servicing routines. The Cydra 5 VLIW processor and the Multiflow family of processors use this approach. Another method, the restart technique, aborts all operations following the one generating the exception, services the exception and then restarts all subsequent instructions. Ozer et al [18] propose a fast interrupt handling scheme for VLIW processors that falls into the "restart" category. On an exception, all operations and instructions prior to the operation that generated the exception are flushed and the exception is serviced. The processor then enters a special mode that completes all other remaining operations. Operations that completed correctly are converted to NOPs and thus not re-executed. To do this, information is maintained as bit vectors in what is called a current state buffer (CSB), reminiscent of the technique of Tornig and Day for dynamically scheduled processors [19]. Normal execution resumes when all operations prior

to and including the offending operation are completed. To permit the state to be recovered correctly in this manner, the CSB relies on compiler support to convert unsafe anti-dependencies to flow dependencies. A similar technique is described in [5], in which enable bits are used following an exception to partially commit known good instructions prior to the excepting instruction in program order. The exception/interrupt handling techniques for VLIW processors described above are quite different from our approach. We specifically focus on systems using binary translation and use explicit trace optimization techniques that are geared to the implementation of non-atomic traces, taking advantage of the commit buffer hardware to implement checkpoints within a long trace.

In a recent patent [4] (since withdrawn), a mechanism for incrementally committing traces is disclosed. Although superficially similar in concept to our scheme, there are many fundamental differences. Scheduling across commit points is much more limited (particularly for branch evaluation), explicit commit instructions are needed at all exits from a trace on mispredictions, results only visible after future commit points must occupy resources in the core instead of being offloaded to a commit buffer for commitment as they become ready, and no commit depth prediction is present to reduce pipeline flushes.

In this paper, we described the principles, hardware structures and scheduling methods behind incremental commit groups, and the trace prediction and merging mechanisms required to efficiently support them. To empirically support the advantages of non-atomic commitment, we demonstrated that on the SPEC suite, over 50% higher performance can be achieved with commit groups, largely because of reduced rollback penalties (often 40-60% lower) and the ability to aggressively schedule long supertraces with the assurance that even partially committed traces will provide forward progress and form the basis of a more quickly adapting pool of dynamically optimized traces than if atomic semantics were required. We have also described how commit depth prediction is an integral part of trace prediction for non-atomic commit group based traces. Finally, we have shown that commit groups are easy to add on top of traditional VLIW scheduling, and the hardware requirements are minimal, mainly consisting of a relatively small banked register file for the commit buffer outside the critical timing path.

Acknowledgements

This work supported in part by the NSF under award numbers EIA 9911099 and CNS 0454298. We would also like to thank Tom Conte and the anonymous reviewers for their suggestions.

References

- [1] M. Wing et al. *Method and apparatus for aliasing memory data in an advanced microprocessor*. U.S. Pat. 5926832, filed 26 Sep 1996. Transmeta Corp.
- [2] E. Kelly et al. *Host microprocessor with apparatus for temporarily holding target processor state*. U.S. Pat. 5958061, filed 24 Jul 1996. Transmeta Corp.
- [3] M. Wing et al. *Gated store buffer for an advanced microprocessor*. U.S. Pat. 6011908, filed 23 Dec 1996. Transmeta Corp.
- [4] L. Torvalds et al. *Method for translating instructions in a speculative microprocessor featuring committing state*. U.S. Pat. 6871342, filed 13 Oct 1999, issued 22 Mar 2005 (withdrawn). Transmeta Corp.
- [5] B. Coon et al. *Use of enable bits to control execution of selected instructions*. U.S. Patent 6738892, filed 20 Oct 1999, issued 18 May 2004. Assn. Transmeta Corp.
- [6] A. Klaiber. *The Technology Behind Crusoe Processors*. Transmeta Technical Report, January 2000.
- [7] J. Dehnert et al. *The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges*. CGO 2003.
- [8] K. Krewell. *Transmeta gets more Efficeon: Transmeta delivers new core, code-morphing software*. *Microprocessor Report*, 1-Oct-2003.
- [9] E. Altman, K. Ebcioglu. *DAISY Dynamic Binary Translation Software*. Software Manual for DAISY Open Source Release, 2000.
- [10] E. Altman and M. Gschwind. *BOA: A Second Generation DAISY Architecture*. ISCA 2004.
- [11] B. Fahs et al. *The Performance Potential of Trace-based Dynamic Optimization*. University of Illinois Technical Report, UILU-ENG-04-2208, Nov 2004.
- [12] G. Hinton et al. *The Microarchitecture of the Pentium 4 Processor*. Intel Tech Journal, Vol. Q1, 2001.
- [13] G. Tyson, T. Austin. *Improving the Accuracy and Performance of Memory Communication Through Renaming*. Proc. MICRO 1997.
- [14] G. Chrysos, J. Emer, *Memory Dependence Prediction using Store Sets*, ISCA 1998
- [15] A. Moshovos and G. Sohi. *Speculative Memory Cloaking and Bypassing*. Intl. J. Parallel Proc. 27(6) 1999, p. 427-456.
- [16] M. Smith et al. *Efficient Superscalar Performance Through Boosting*. ALPLOS 1992.
- [17] B. Rau. *Dynamically Scheduled VLIW Processors*. MICRO 1993.
- [18] E. Ozer et al. *A Fast Interrupt Handling Scheme for VLIW Processors*. PACT 1998.
- [19] H. C. Tornig, Martin Day. *Interrupt handling for out-of-order execution processors*. IEEE Trans. on Comp., Jan 1993
- [20] K. Rudd. *VLIW Processors: Efficiently Exploiting Instruction Level Parallelism*. PhD Dissertation, Dept of Elec Eng., Stanford Univ. 1999.
- [21] J. Martinez et al. *Cherry: Checkpointed Early Resource Recycling in Out-of-Order Microprocessors*, MICRO 2002
- [22] H. Akkary et al. *Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors*, MICRO 2003
- [23] A. Cristal et al. *Out Of Order Commit Processors*, HPCA 2004
- [24] O. Ergin et al. *Increasing Processor Performance Through Early Register Release*. Proc. ICCD 2004.
- [25] H. Chen et al. *Profile-based optimizations: Dynamic trace selection using performance monitoring hardware sampling*. CGO 2003.
- [26] S. Patel et al. *Increasing the Size of Atomic Instruction Blocks Using Control Flow Assertions*. MICRO 2000.
- [27] Q. Jacobson et al. *Path-Based Next Trace Prediction*, MICRO 1997.
- [28] J. Smith et al. *Implementing Precise Interrupts in Pipelined Processors*. IEEE Trans. Comp. Vol 37 Issue 5, 1998.