Published in the 5th International Symposium On High Performance Computer Architecture, January 1999.

Instruction Recycling on a Multiple-Path Processor

Steven Wallace Dean M. Tullsen Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{swallace,tullsen,calder}@cs.ucsd.edu

Abstract

Processors that can simultaneously execute multiple paths of execution will only exacerbate the fetch bandwidth problem already plaguing conventional processors. On a multiple-path processor, which speculatively executes less likely paths of hard-to-predict branches, the work done along a speculative path is normally discarded if that path is found to be incorrect. Instead, it can be beneficial to keep these instruction traces stored in the processor for possible future use.

This paper introduces instruction recycling*, where previously decoded instructions from recently executed paths are injected back into the rename stage. This increases the supply of instructions to the execution pipeline and decreases fetch latency. In addition, if the operands have not changed for a recycled instruction, the instruction can bypass the issue and execution stages, benefiting from* instruction reuse*. Instruction recycling and reuse are examined for a simultaneous multithreading architecture with multiple path execution. It is shown to increase performance by 7% for single-program workloads and by 12% on multiple-program workloads.*

1 Introduction

Modern processors spend much of their time doing repetitious tasks — fetching the same instructions and executing them over and over again, sometimes even with the same operand values. This is true of speculative processors, which may execute an instruction down a speculative path, throw it away, and execute the same instruction down the correct path which has merged with the previous incorrect path. This is even more evident in processors capable of executing along multiple speculative paths [17, 5, 16, 7, 1, 18], where the same instantiation of an instruction may be executed by several threads or virtual processors. The fetch unit is highly repetitious on any iterative task, even on a non-speculative processor; but much more so on a multiple-path processor.

Modern processors exploit these recurrent instructions via caches; the instruction cache avoids memory access by the fetch unit, and the data cache saves the data for re-executed loads with the same address. However, all other stages of execution are repeated, even if the inputs of an instruction are unchanged. This paper examines a technique for recycling previously-fetched instructions back through the processor, saving fetch and decode bandwidth in the worst case, and execution resources and latency in the best case. Recycled instructions can augment the instructions coming through the normal fetch path, which is typically hampered by branch and cache line boundary fetch limitations. This increases the bandwidth of instructions into the machine in three ways. It increases the raw bandwidth into the processor by merging recycled instructions with fetched instructions. It increases fetch parallelism, as instructions from more contexts can be introduced into the machine in a single cycle. Third, recycled instructions re-enter the processor in the form of a trace, bypassing branch and cache line boundaries.

Although many of the techniques examined in this paper will also work for more conventional speculative processors, we study them within the context of a multiple-path processor. These techniques apply to most of the multiple-path architectures recently proposed, but our baseline architecture is derived from our Threaded Multipath Execution (TME) architecture described in [18]. A TME processor uses idle hardware contexts on a simultaneous multithreading (SMT) [14, 15] processor to execute down both paths at conditional branch points, potentially eliminating the branch misprediction penalty. Our study [18] showed that performance can be increased by creating threads in hardware to execute instructions down both paths of certain hard-to-predict conditional branches. TME differs from other multiple-path architectures by also allowing multiple programs to be sharing the processor via SMT.

Other techniques that have been proposed to preserve instruction cache and execution bandwidth are the Trace Cache [10] and the Reuse Buffer [12]. The Trace Cache only

bypasses the instruction cache, as all instructions still use all pipeline stages, including fetch. The Reuse Buffer does not bypass the fetch unit, but does bypass the execution stages for instructions whose operands have not changed.

Unlike those mechanisms, we modify existing structures to enable recycling, which minimizes the cost of additional storage. We recycle the instructions directly from the active lists (similar to a reorder buffer) in the TME architecture. In addition, we examine recycling in an environment (multiple-path execution) where the incidence of redundant fetch and execution is much higher than traditional single-thread processors.

Instruction recycling and reuse increase single-program performance over TME by 7% on average. With multiple programs running, where TME has been shown to be less effective, recycling and reuse achieve a 12% increase (for four programs running on an eight-context processor) by easing the contention for fetch resources.

This paper is organized as follows. We describe the multiple-path architecture on which this research is based in Section 2. The additional hardware to permit instruction recycling is given in Section 3. Our evaluation methodology is described in Section 4, and Section 5 presents our performance results for a number of different architectural alternatives. Section 6 describes related research, and Section 7 summarizes our results.

2 Threaded Multiple Path Execution

A simultaneous multithreading processor allows multiple threads of execution to issue instructions to the functional units each cycle. This can provide significantly higher processor utilization than conventional superscalar processors or traditional multithreaded processors, which also use multiple hardware contexts (program counters, registers) to boost throughput and latency-tolerance. The ability to combine instructions from multiple threads in the same cycle allows simultaneous multithreading to not only hide latencies, but also to more fully utilize the issue width of a wide superscalar processor.

Threaded multi-path execution extends SMT by using unused contexts to execute both paths of conditional branches. As a result, the processor resources can be more fully utilized, and the probability of executing the correct path is increased. By executing both paths of the branch, TME can eliminate the branch misprediction penalties for hard to predict branches [18].

The register renaming and mapping hardware of an SMT/TME processor is particularly relevant to this discussion. Register renaming takes place via a register mapping scheme (similar to the MIPS R10000 [19]) extended for simultaneous multithreading and TME as shown in Figure 1. Each instruction that writes a register removes a physical register from the free list and writes a new mapping to the mapping table. When that instruction commits from the end of an active list, it frees the physical register used by the previous mapping. Each context has its own active list so that instructions can commit independent of the progress of other threads.

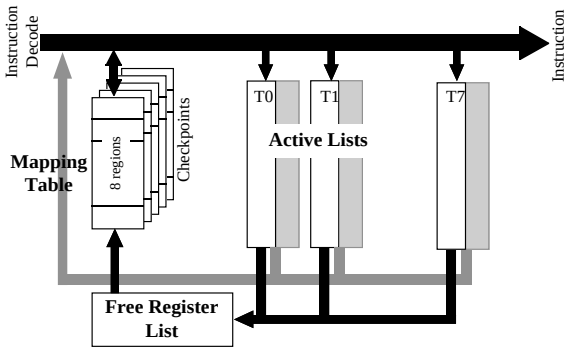

Figure 1: The register mapping scheme for an SMT/TME processor (black data paths), with additions to support recycling (gray paths).

Each context also requires its own mapping region to translate its own set of logical registers, so the mapping table of an 8-context SMT/TME processor has 8 mapping regions as shown in Figure 1. Mapping tables in existing processors are shadowed by *checkpoints*, which are snapshots of the table taken when a branch was encountered. The active list in a conventional processor contains the physical register mapping that will be freed if this instruction retires. In the SMT/TME processor, the active list also contains that instruction's new mapping, which will be freed if the instruction gets squashed. This is necessary because instructions in different threads get squashed independently.

The following terms are used to describe the behavior of the TME architecture: *Primary-path thread* — the thread that has taken the predicted path through the code; *alternate-path thread* — a thread that is executing an unpredicted path through the code; *idle context* — a context that is not executing any thread; *spare context* — a context that is partitioned to a primary thread to be used for alternate path execution; *fork a branch* — take both paths of a branch; *spawn a thread* — create an alternate-path thread.

To effectively follow both paths of a single branch, we must be able to quickly duplicate the register state of the executing context to an idle context. A TME processor has a single shared physical register file, so we can duplicate register state simply by duplicating the first context's register map. The TME architecture does this via the *Mapping Synchronization Bus* (MSB) [18]. The MSB partitions the register map into groups of contexts. Each group includes one primary thread and zero to seven alternate contexts available to spawn alternate paths. All idle threads within a partition

are kept in sync with the primary thread using the MSB, so that they are available for spawning immediately.

TME only spawns alternate threads off of branches in the primary thread. Candidate branches are selected based on branch confidence prediction methods [6]. When a branch that spawned an alternate path is found to be mispredicted, the alternate path thread becomes the primary thread, and all idle threads use the MSB to re-synchronize with the new primary thread. See [18] for a complete description of TME.

Results in [18] show that (1) TME achieved significant speedups when a single (low branch accuracy) program was running, (2) TME does not degrade the performance of SMT when multiple programs are running, and (3) TME provides the most benefit for programs with low branch prediction accuracy. It does not degrade the performance of programs with high branch prediction accuracy, because branch confidence controls the spawning of alternate paths.

TME provides performance advantage when there are idle resources, but it provides diminishing returns in performance as more primary threads (programs or software threads) are executed. In this situation, there is insufficient fetch bandwidth to adequately serve all primary and alternate paths, so the latter starve.

3 Hardware Support for Instruction Recycling

The active lists on a TME processor already contain predicted traces of fetched instructions. In this section, we show how these traces can be exploited by recycling them into the processor to provide higher instruction bandwidth. Recycling saves fetch bandwidth, bypasses fetch limitations (branches and cache lines), and can allow the reuse of instruction values to eliminate instruction latencies.

To enable instruction recycling and reuse, we need to (1) preserve instruction information in the processor and keep it around as long as possible, (2) detect when a thread should stop fetching and begin gathering instructions through recycling, (3) have a mechanism for identifying instructions that need not be re-executed, and (4) have additional datapaths to reinsert instructions back into the processor. In Figure 1, the gray regions represent additions to the register renaming architecture of a TME processor to support instruction recycling. These include additional information stored in the active list to reconstruct each instruction (this includes the decoded opcode and physical and logical register operands) and a new datapath from the active lists to the rename path to inject those instructions back into the processor.

3.1 Managing Spare Contexts to Maximize Recycling Availability

In TME, a spawned path is always squashed (i.e., active list cleared, register mappings freed) as soon as a correctly predicted branch is resolved. However, with recycling, we want to delay the squashing of alternate-path threads as long as possible to maximize the opportunity to recycle instructions. In the recycle architecture, a context can be either active or inactive. An *active* context is currently executing either the primary or an alternate path. An *inactive* context has finished executing, but the active list and registers have not been freed, making it available for recycling. We will only see *idle* contexts (not available for recycling) at startup. Normally, contexts will be kept inactive until just before they are reclaimed for TME spawning.

In the TME/Recycle architecture, a primary path has a number of spare contexts associated with it for executing alternate paths. When a correctly-predicted branch is resolved, the corresponding alternate thread (if there is one) stops executing, but it is not squashed, allowing the architecture to use the instructions for recycling — the thread becomes inactive. The current register map of the newly inactive thread is then checkpointed, and the mapping is resynchronized with the primary path. The re-synchronization allows the spare context to be spawned for TME immediately when another low confidence branch is encountered. Upon encountering a low-confidence branch, the architecture identifies the least-recently-used inactive context and reclaims it, squashing the instructions in the active list and freeing the registers.

Under TME, we can have several alternate paths that start at the same instruction, corresponding to various instantiations of the same low-confidence branch. We would like to minimize this duplication. We would rather preserve contexts to fork other branches, creating more unique starting points for recycling. Therefore, instead of creating many alternate paths with the same start point, we can *re-spawn* the existing inactive context, when one already exists with that start address. Re-spawning re-executes the instructions in the inactive thread through the *recycling* data paths, making it active again. Since recycling is used to provide the initial instructions down the alternative path, fetch bandwidth is saved when re-spawning. Therefore, re-spawning maintains the benefits of TME, but with much less contention for fetch resources compared to regular TME.

3.2 Identifying Merge Points

To determine when instructions from a context can be recycled, we need to identify *merge points*, where the current path has merged with another path that is available for recycling. In particular, we want to determine when the primary (most likely) path has merged with another path (that may or may not still be active). Most often, the merge point is the first instruction of the alternate context. For example, if the alternate path begins at a branch target because the primary path predicted the branch not taken, the paths will most likely merge in the future when the branch is later taken (or predicted taken).

To identify merge points the program counter (PC) of the first instruction in the active list is stored with each hardware context. The current PC that is used to fetch instructions for the primary path is used to search the merge PCs of its spare contexts and its own hardware context for a match. If a match is found with an alternate path thread, subsequent instructions will come from the alternate active list once the prior fetched instructions for that thread have cleared the rename stage. If a match is found with the primary path hardware context, we will be recycling instructions from the primary path thread, back into its own active list. The recycling frees the fetch unit to begin fetching from other threads, or perhaps even further downstream for the same thread.

Since backward branches are common from loops in programs, we also record the target address of the *last* backwards branch for each context. It is searched along with the first PC as a possible merge point. These are the only two possible merge points for a particular context that we consider. If another backwards branch is detected, it overwrites the previous backward branch merge point. Also, if an instruction is inserted into the active list which overwrites the first instruction of a backwards branch merge point, then the merge point is invalidated. Hence, only loops smaller than the current active lists are able to benefit from the backward branch recycling.

3.3 Recycled Datapath

To enable recycling, an extra datapath is needed from the active lists back through the renaming logic to be able to reintroduce this data into the pipeline. A recycled instruction is read from the originating active list, makes a new pass through register renaming, writes into a (potentially different) active list, and (if not marked for reuse) into the instruction queue for execution. Recycled instructions, at a minimum, bypass the fetch and decode stages (Figure 2). Reused instructions can bypass the queue and execution (and surrounding) stages as well.

Instructions which are recycled and instructions which are fetched and decoded need to be merged in the rename stage. We give highest priority to instructions from the fetched paths, filling in empty slots with recycled instructions. The only constraint is that program order is maintained for a thread that has instructions from both sources, which may involve blocking instructions either trying to recycle or coming from the decode stage.

Instruction fetch priority of a thread is determined by the number of instructions in the pipeline. This is identical to the ICOUNT fetch scheme used by Tullsen, et al. [14] (with the same modifications for TME recommended in [18]), except when instructions are recycled. In this case, the number of instructions recycled is added immediately to the instruction counter used for fetching priority. When multiple threads want to recycle, a separate instruction counter is used to determine the priority of those threads for insertion into the rename stage. This counter uses the number of instructions in the rename and queue stages only.

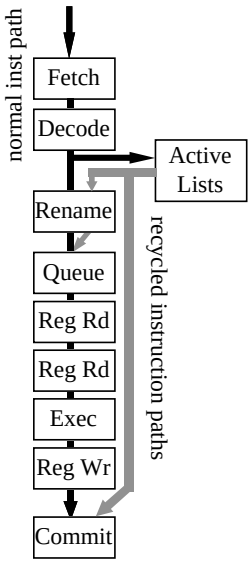

Figure 2: Instruction recycling allows instructions to bypass parts of the processor pipeline.

We assume no expansion of the rename stage to accommodate the two paths, which is 16 instructions wide in the processor we simulate.

3.4 Recycling Instructions

When a merge point is detected, the processor begins recycling instructions from the corresponding active list. Fetching immediately continues from where recycling will complete. This is enabled by saving the PC of the instruction after the last instruction in the active list, which will become the new fetch target. The branch prediction previously used for the recycled instructions can be used. The global history register used for branch prediction is then updated with that prediction. Alternatively, when the instructions are read from the active list, if the branch prediction differs, recycling can stop and fetching continues on the newly predicted path. For this paper, we use the latter method. The former method still requires one more branch prediction per cycle than normal SMT/TME, because in the event that the predicted fetch point for a thread is a merge point, we would like to be prepared to use the fetch opportunity for other threads that cycle. The latter method requires even higher prediction throughput and should be considered an aggressive approach.

Each cycle, when the primary thread prepares to fetch, it will compare its fetch PC (and following addresses) with the merge points of itself and its alternate contexts. Also, each thread that fetches will also compare its PC with its own backward-branch merge point. If the match is on the initial

PC, then there is no need to fetch from the instruction cache for this thread, and another thread is sought for fetching. In the case a match is found in the middle of a fetch block, instructions are fetched up to the matching instruction, and recycling begins after it.

3.5 Instruction Reuse

For some recycled instructions, execution can also be bypassed. If none of the operands of a recycled instruction have been changed, and the instruction was actually executed, the old computed value can be *reused* [12]. We accomplish this by re-using the old register mapping (writing it into the new mapping table entry) instead of re-mapping the instruction. Subsequent instructions, either fetched or recycled, then use the value already in the register. Reuse for backward branch recycling is not allowed, given the way we track register changes for re-use detection. Reuse is only allowed for alternate to primary thread recycling.

Instructions can be reused more than once, but this complicates register and context reclaiming — in particular, we need to ensure we do not free a register (by squashing a context in preparation for spawning) which another context is still accessing due to re-use of the register mapping. We do not free the registers used by a recyclable active list until (1) TME wants to use the context to spawn a new path, and (2) all other reuses of instructions in this path by the primary path have completed. To implement this, we assume each alternate path keeps track of the last reuse by its primary path. When the primary path commits a result, it checks to see if it is the last reuse among alternate paths and clears any matches. Then the alternate path will be able to free all its registers when it is used again to fork off a low-confidence branch.

In order to determine if a register has changed since an instruction was executed, a new structure is introduced. A *written* bit-array of contexts indexed by logical registers is used. When a new path is started on a context, the column of register bits for that context is reset. When the *primary* context makes a new instance of a register, then the row of context bits for that register is set. As a result, when an operand of a recycled instruction is checked during the renaming, the corresponding bit for that register and context is looked up in the array. If it is reset, then it has not been changed. If it is set, then it has been changed and the instruction cannot be reused.

We assume that load operations can be reused if the source register, and thus the address, has not been changed and there are no intervening stores to the same address. A Memory Disambiguation Buffer (MDB) can be used to keep track of loads whose values can be reused. The MDB provides hardware support to determine if a load instruction's address has been overwritten since the last time the load was executed. The MDB is used to store the load PC and the effective addresses. When subsequent stores are executed, it searches the MDB for its effective address. If the store finds its address in the MDB, the load PC and address are removed from the MDB. When recycling the load PC, if it is still located in the MDB, then we can reuse its value. Otherwise, the load has to be re-executed.

4 Evaluation Methods

All of our results are obtained using execution-driven simulation of a multiple-thread processor running Alpha executables. The simulator models a simultaneous multithreading processor extended for threaded multipath execution and recycling, as described in Section 3. Instruction latencies are based on the DEC Alpha 21264.

Our workload consists of eight of the SPEC95 benchmarks. Six of the programs are integer benchmarks (compress, gcc, go, lisp, perl, and vortex) and two are floating point (su2cor and tomcatv). Although most of the SPEC95 floating point programs do not benefit from TME due to high branch prediction accuracy, there is potential for benefit from recycling due to primary-path to primary-path recycling. All of the benchmarks were compiled with the DEC cc (version 5.2) and f77 (version 4.1) compilers with full optimization (-O5 -om).

We will look at recycling both with single-thread workloads and multiple-thread workloads (multiple single-thread applications running simultaneously). For the multi-thread workloads, the results shown consist of the average of eight permutations of the benchmarks that weight each of the benchmarks evenly in the results.

4.1 Baseline Architecture Model

We evaluate instruction recycling in the context of a future-generation 16-wide SMT/TME processor with 8 hardware contexts. It has the ability to fetch eight sequential instructions from each of two different threads each cycle. Those instructions, after decoding and register renaming, find their way to one of two 64-entry instruction queues. Instructions are issued to the functional units (6 floating point, 12 integer, 8 of which also can do load-store operations) when their register operands are ready. This is an aggressive design, but exposes many of the problems future processors will exhibit. We also examine more conservative architectures in Section 5.3.

The simulated memory hierarchy has 64KB direct-mapped instruction and data caches, a 256 KB 4-way set-associative on-chip L2 cache, and a 4 MB off-chip cache. Cache line sizes are all 64 bytes. The on-chip caches are all 8-way banked. Throughput as well as latency constraints are carefully modeled at all levels of the memory hierarchy. Conflict-free miss penalties are 6 cycles to the L2 cache, an-

other 12 cycles to the L3 cache, and another 62 cycles to memory.

Branch prediction is provided by a decoupled branch target buffer (BTB) and pattern history table (PHT) scheme [2]. We use a 256-entry BTB, organized as four-way set associative. The 2K x 2-bit PHT is accessed by the XOR of the lower bits of the address and the global history register [9, 20]. Return destinations are predicted with a 12-entry return stack (per context).

The assumed processor has a 9-stage pipeline, with a minimum 7-cycle branch misprediction penalty. We assume each register file (fp and integer) has enough registers to store the logical registers of the eight contexts (when all eight contexts are being used), plus 100 more for register renaming. This is the same number used in many previous SMT studies, even though recycling (and the larger machine) puts additional pressure on the renaming registers.

5 Results

This section examines the performance of instruction recycling and reuse, examines policies for fetch and execution after a context becomes inactive, and considers the effectiveness of recycling techniques on some alternate architectures.

5.1 Recycling, Reusing and Respawning

The baseline instruction recycling architecture extends the previously described SMT/TME architecture by saving executed instructions in the active list even after the thread becomes inactive, and enables their injection into the architecture. This eliminates refetching and in some cases re-execution of the instruction, and significantly increases fetch bandwidth.

Figures 3 and 4 show the performance for the default SMT and TME architectures and recycling with and without re-spawning and reuse. The architectural parameters shown are as follows:

- **SMT** This is the base simultaneous multithreading architecture. It can only exploit inter-thread parallelism when there are multiple threads running.
- **TME** Threaded multiple-path execution can exploit parallelism between multiple paths in the same instruction stream, but puts high demand on the fetch and execution units for the occasional benefit of a mispredicted branch. The rest of the results all include TME and one or more of the following optimizations.
- **REC** Recycling. Instructions from alternate paths, inactive threads, or even the primary thread, can be merged at the rename stage with fetched instructions. (This does not include the Reuse and Re-spawn options.)
- **RU** Reuse. Recycled instructions from inactive threads whose operands are unchanged are not dispatched to the instruction queue. Rather, the old result is reused.
- **RS** Re-spawning of identical paths. The REC result suffers from the design decision to not spawn threads with an identical start address as an existing alternate or inactive thread. This increases the number of unique merge points available for recycling, but decreases opportunities for TME. The RS architecture re-spawns an inactive thread that matches the start address of a path TME wants to spawn. But it is respawned via recycling, without consuming fetch bandwidth.

In single-program execution, recycling is effective any time TME is effective, although in one case recycling alone (without reuse) actually under-performs TME (compress). That is also the program where we get the largest benefit from reuse. Reuse, on average, increases performance by about 2%. Re-spawning provides speedup in about half of the applications (about 2% increase on average over REC). The best combination (REC/RS/RU) had an average 7% improvement over TME.

For multiple programs, the benefits of recycling go up significantly, just as the benefits of TME (over SMT) are dropping. With multiple programs, competition for the fetch unit is high, rendering TME ineffective while magnifying the importance of fetch-conservation through recycling. Instruction reuse is not as important, only increasing performance by about half a percent on average. Re-spawning still provides a 2% increase in performance. With the best configuration, performance is improved by 12% over normal TME with four programs running.

Table 1 lists recycling statistics. The first two columns show the percentage of all instructions (including squashed ones) inserted into the rename stage that were recycled and reused, respectively. *Branch Miss Cov* gives the percentage of mispredicted branches that were successfully covered by speculatively forking. The next three columns give the percentage of forked paths used successfully by TME, recycled, or respawned at least once, relative to the total number of branches forked. *Merges Per Alt Path* gives the average number of merges from a given recycled alternate path before it was deleted (this does not include backward branch merges). Finally, *Back Merges* is the percentage of all merges that were from backward branches.

From Table 1 we can see that the level of recycling is generally very high. 17-61% (average of 33% for one thread) of all forked (spawned) paths are used for recycling, and 9-56% (average, 27% for one thread) of instructions introduced into the machine come from recycling. 6% of instructions introduced into the machine are actually reused. As the number

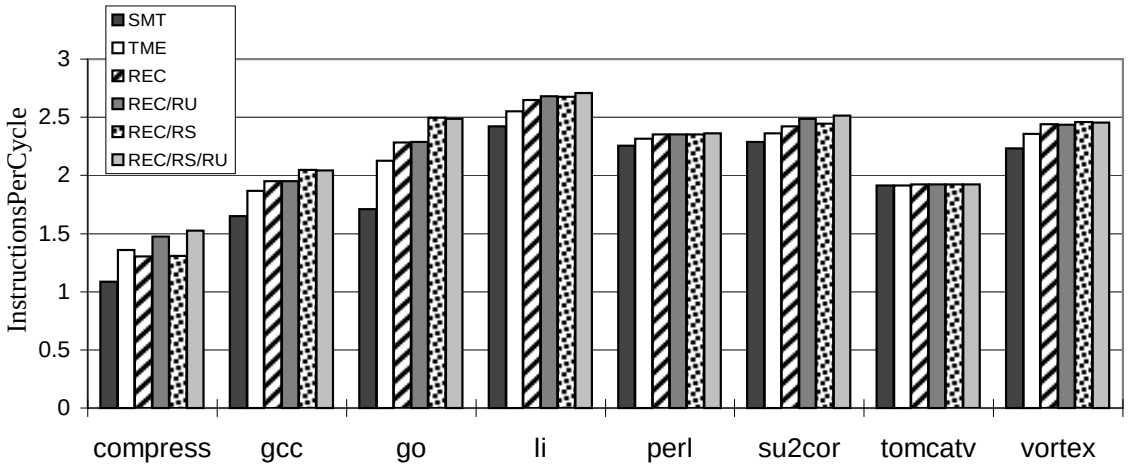

Figure 3: Performance of recycling with reuse and respawning for individual programs.

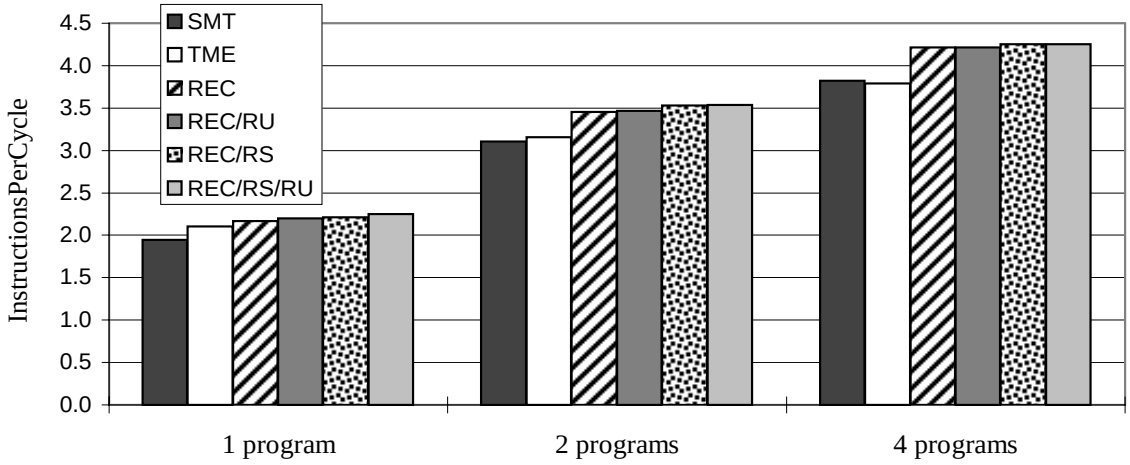

Figure 4: Average Performance of Recycling with Reuse and Respawning when running 1, 2 and 4 programs at once.

of programs increases, reliance on alternate to primary recycling and reuse goes down, while reliance on primary to primary backward branch recycling goes up. This is due to a decrease in the number of alternate contexts available to hold recyclable instructions per program, and explains the lower importance of reuse with multiple programs in Figure 4.

In addition, TME still does a good job at covering mispredicted branch paths with an average coverage of 67% for four program and 72% for single program results.

5.2 Recycling Fetch Limits

With recycling, instructions can be useful even after the branch is resolved that identifies the instructions as on the wrong-path. That may even be true for instructions that have not been fetched or executed yet along this path. We examine continuing fetch and/or execution after the path becomes inactive, up to various cutoff points (in total number of instructions). The danger in stopping immediately is that a context with just one or two instructions does not have enough instructions to enable effective recycling, yet that context can inhibit future spawning of threads. The danger in fetching too long is that we occupy fetch bandwidth for instructions that may have a low likelihood of being used.

Figure 5 shows results for the following policies:

- stop 8, 16, 32 — stop immediately when the branch is resolved and the context becomes inactive. In addition, it does not allow TME to ever follow an alternate path for more than 8, 16, or 32 instructions.

Program	% Instrs Recycle	% Instrs Reuse	% Branch Miss Cov	% Forks TME	% Forks Recyc	% Forks Respawn	Merges Per Alt Path	% Back Merges
compress	55.9	14.2	66.5	18.3	61.1	25.8	1.9	57.3
gcc	22.4	7.8	80.1	17.1	28.4	4.4	1.6	35.2
go	24.3	10.3	84.6	20.3	26.7	3.5	1.5	41.1
li	31.9	5.7	81.8	19.9	41.0	12.6	2.3	47.5
perl	9.0	1.4	92.4	11.3	17.2	3.8	1.8	76.1
su2cor	32.0	4.2	78.5	11.9	34.7	8.8	1.9	41.9
tomcatv	25.1	0.5	3.5	0.4	28.7	17.9	1.2	17.8
vortex	13.7	3.8	85.2	22.3	22.0	3.7	1.7	35.5
1 prog avg	26.8	6.0	71.6	15.2	32.5	10.1	1.7	44.1
2 progs avg	24.5	4.7	77.5	20.4	26.1	4.2	1.5	57.2
4 progs avg	22.0	2.6	66.6	29.1	13.2	0.9	1.1	80.4

Table 1: Recycling Statistics

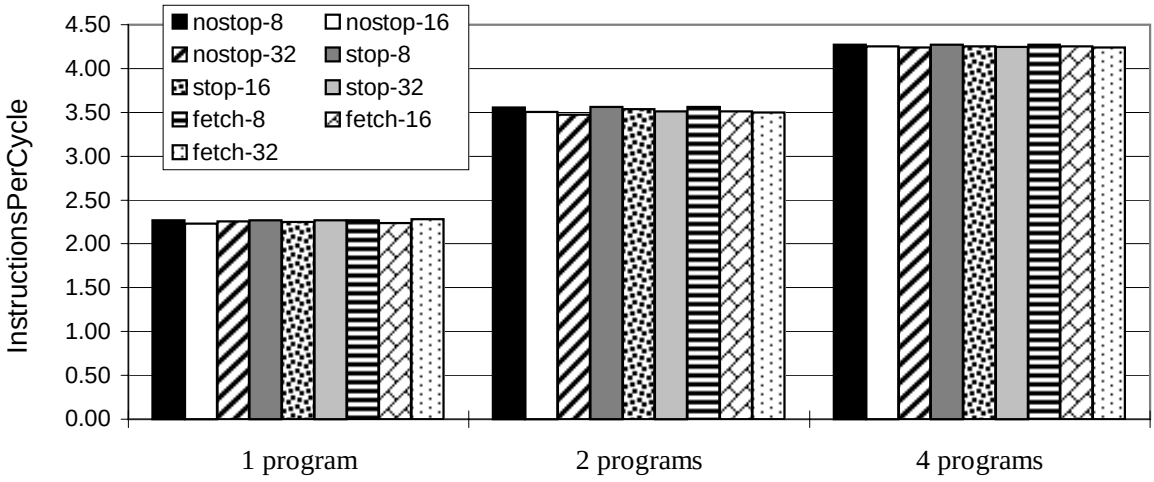

Figure 5: Effects of different recycling fetch limits.

- fetch 8, 16, 32 — do not issue any more instructions for execution after branch resolution, but continue fetching up to the total 8, 16, or 32 instruction limit.

- nostop 8, 16, 32 — continue fetching up to a total of 8 to 32 instructions, and send all of those instructions to the instruction queue to be scheduled for execution.

Although the results indicate this is not a major performance factor, a fetch limit of 8 instructions for an alternate thread achieves some performance gain over fetching more. Most of the performance savings appears to be achieved in the first block of instructions and fetching more blocks results in using too much fetch bandwidth relative to the return received. Stopping immediately after resolution worked well for most programs; however, the results indicate that all of the policies provide acceptable performance.

5.3 Performance for Limited Resource Architectures

Our architecture has assumed an aggressive processor which can fetch 16 instructions from 2 threads each cycle and execute as many as 18 in a cycle. We now examine the performance of recycling, respawning and reuse (we'll call the combination recycling for brevity) with three less aggressive architectures. We will examine three new processor design points. We will look at the same 18 functional-unit processor, but with reduced fetch bandwidth, allowing only one thread to fetch up to eight consecutive instructions per cycle (this is the big.1.8 result, the baseline is big.2.16). We will also look at two machines about half the size (with half the functional units and half the cache and instruction queue sizes as our baseline processor) and the eight-instruction fetch bandwidth filled by one (the small.1.8 result) or two threads (the small.2.8 result). These machines correspond closely to the processors in [14, 18].

For multiple programs, recycling improves performance over TME and SMT for all architecture configurations. Re-

cycling supplies the larger architecture with primary and alternate instructions effectively and it greatly benefits from the additional instructions. For the smaller architecture, recycling improves the performance of 1.8 fetching so that a more complicated 2.8 fetching mechanism becomes less necessary.

While recycling is effective for all of these architectures, it is most effective when the fetch unit is least able to fill the fetch bandwidth — the 1.8 fetch scheme for the small machine, and the 2.16 fetch scheme for the big machine.

6 Related Work

There have been several studies describing architectures that can follow multiple paths through execution concurrently [17, 5, 16, 7, 1, 18]. All of these multiple path architectures create a scenario of high redundancy in the processor, which can be exploited with instruction recycling.

The Speculative Multithreaded (SM) processor [8], and the Multiscalar processor [13] are two speculative architectures that allow aggressive loop-based speculation. If all the hardware contexts are executing similar paths in the loop, these architectures could benefit from instruction recycling. The Speculative Multithreaded processor can take advantage of this in its fetch scheme, since each thread context has its own rename stage. In SM, only one thread context is allowed to fetch from the cache at a time, but the instructions are broadcast to all the thread contexts. Therefore, if other contexts are waiting on the same instructions they will also benefit from the fetch.

In our TME/Recycle architecture, we use the active lists as small caches of instruction traces. This provides a similar effect to a trace cache [10]. A trace cache collects traces of instructions, allowing those instructions to be placed in the processor at a higher rate than they can be fetched from the instruction cache, fetching multiple basic blocks per cycle. However, there are key differences between the recycling architecture and trace caches. The trace cache design does not allow the trace-path instructions and the instruction-cache path instructions to be introduced into the machine in the same cycle — in a multithreaded environment, recycling benefits from the increased parallelism of bringing instructions from multiple sources and multiple threads into the processor in a single cycle. Also, the recycling architecture requires much less additional hardware storage than a large trace cache, both because it takes advantage of some data that is already in the active list, but also because we only keep a few traces that prove to be beneficial (those that are not used are quickly reclaimed). The recycling architecture is also unique in its ability to construct useful traces along paths not yet (or not recently) taken.

Smotherman and Franklin [11] propose a decoded instruction cache to address the complexity of decoding CISC instructions. That cache can operate as a trace cache, holding non-contiguous instructions, but also can reduce the fetch latency when it hits, like recycling. Our proposed recycling architecture differs from that work in all the aspects listed in the previous paragraph.

Sodani and Sohi [12] use the reuse buffer to identify instructions which do not need to be re-executed because their operands have not changed. Our technique for identifying instructions that can be reused is similar to their S_n technique. In fact, their approach to reuse is more general than the one examined in our paper, and may provide a higher rate of reuse in some cases, but at a cost of more specialized hardware to store values. Their scheme concentrates on reuse and does not seek to reduce fetch bandwidth. We are also examining reuse in the context of a very different architecture, one that aggressively executes down low confidence paths, providing higher opportunities for reuse.

The Memory Disambiguation Buffer we describe to record which load values can be reused is similar to the Memory Conflict Buffer (MCB) proposed by Gallagher et. al. [4]. The MCB provides a hardware solution with compiler support to allow load instructions to speculatively execute before stores. The addresses of speculative loads are stored with a conflict bit in the MCB. All potentially ambiguous stores probe the MCB and set the conflict bit if the store address matches the address of a speculative load. Another approach for memory disambiguation was proposed by Franklin and Sohi [3], called the Address Resolution Buffer (ARB). The ARB directs memory references to bins based on their address and uses the bins to enforce a temporal order among references to the same address.

7 Summary

This paper presents a new architecture to enable instruction recycling and reuse. We examine the performance of this new approach in the presence of threaded multiple path execution and simultaneous multithreading. The TME architecture naturally creates traces of instructions that can be moved, through recycling, at high rates and low latency back into execution. Instruction re-use increases the gain by bypassing nearly the entire pipeline for instructions whose operands have not changed. Thread re-spawning allows multiple-path execution to rely more heavily on recycling for creating alternate paths. This greatly reduces the contention for fetch bandwidth that previously rendered TME ineffective with multiple programs.

The results show that instruction recycling achieves an average 11% improvement over SMT and an average 7% improvement over TME when there is one primary thread. With multiple primary threads, instruction recycling achieves an average improvement of 12%. We found that conservative approaches to TME and recycling, stopping after 8 instructions down an alternate or inactive path, perform very well.

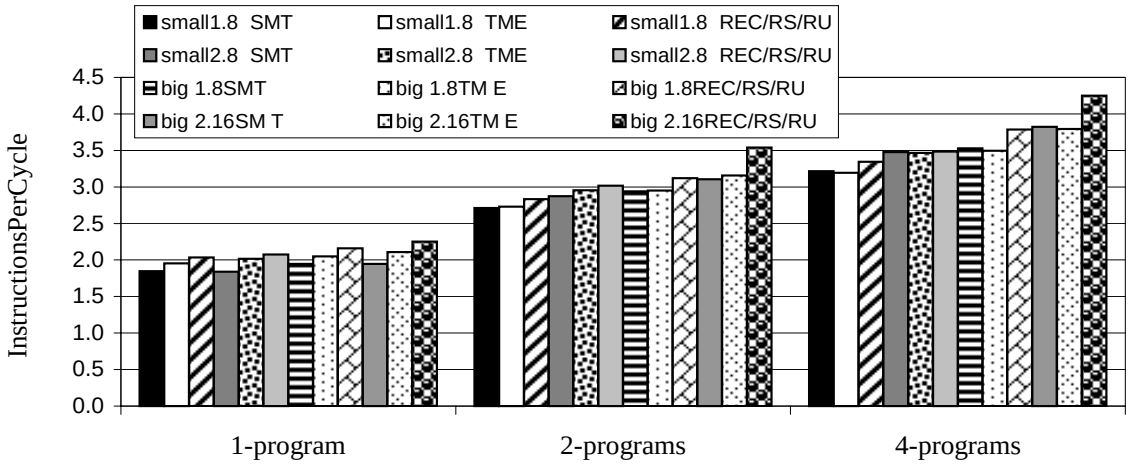

Figure 6: Instruction recycling for different fetch bandwidths.

We also show that recycling is effective on a variety of architectures, from 8-wide to 16-wide.

Acknowledgments

We would like to thank the anonymous reviewers for their useful comments. This work was funded in part by NSF grant No. CCR-980869, NSF CAREER grants No. CCR-9733278 and No. MIP-9701708, and a Digital Equipment Corporation external research grant No. US-0040-97.

References

[1] P.S. Ahuja, K. Skadron, M. Martonosi, and D.W. Clark. Multi-path execution: Opportunities and limits. In *International Conference on Supercomputing*, July 1998.

[2] B. Calder and D. Grunwald. Fast and accurate instruction fetch and branch prediction. In *21st Annual International Symposium on Computer Architecture*, pages 2–11, April 1994.

[3] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 46(5), May 1996.

[4] D.M. Gallagher, W.Y. Chen, S.A. Mahlke, J.C. Gyllenhaal, and W.W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[5] T.H. Heil and J.E. Smith. Selective dual path execution. Technical Report http://www.engr.wisc.edu/ece/faculty/smith. james.html, University of Wisconsin - Madison, November 1996.

[6] E. Jacobsen, E. Rotenberg, and J.E. Smith. Assigning confidence to conditional branch predictions. In *29th Annual International Symposium on Microarchitecture*, pages 142–152. IEEE, December 1996.

[7] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *25th Annual International Symposium on Computer Architecture*, pages 250–259, June 1998.

[8] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative multithreaded processors. In *International Conference on Supercomputing*, July 1998.

[9] S. McFarling. Combining branch predictors. Technical Report TN-36, DEC-WRL, June 1993.

[10] E. Rotenberg, S. Bennett, and J.E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *29th Annual International Symposium on Microarchitecture*, pages 24–34. IEEE, December 1996.

[11] M. Smotherman and M. Franklin. Improving cisc instruction decoding performance using a fill unit. In *28th Annual International Symposium on Microarchitecture*, November 1995.

[12] A. Sodani and G.S. Sohi. Dynamic instruction reuse. In *24th Annual International Symposium on Computer Architecture*, pages 194–205, June 1997.

[13] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[14] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23nd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[15] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[16] G. Tyson, K. Lick, and M. Farrens. Limited dual path execution. Technical Report CSE-TR 346-97, University of Michigan, 1997.

[17] A. Uht and V. Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *28th Annual International Symposium on Microarchitecture*, pages 313–325. IEEE, December 1995.

[18] S. Wallace, B. Calder, and D.M. Tullsen. Threaded multiple path execution. In *25th Annual International Symposium on Computer Architecture*, pages 238–249, June 1998.

[19] K.C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2), April 1996.

[20] T.-Y. Yeh and Y. Patt. Alternative implementations of two-level adaptive branch prediction. In *19th Annual International Symposium on Computer Architecture*, pages 124–134, May 1992.