Multipath Execution on Chip Multiprocessors Enabled by Redundant Threads

Technical Report CESR-TR-01-2

Karthik Sundaramoorthy, Zach Purser, Eric Rotenberg
Center for Embedded System Research (CESR)
Department of Electrical and Computer Engineering
North Carolina State University
{ksundar,zrpurser,ericro}@ece.ncsu.edu

Abstract

The performance penalty of mispredicted branches can be reduced by fetching/executing both paths of unconfident branches. Executing both paths of a branch involves forking a new thread context for the non-predicted path. A key requirement is the ability to quickly copy register and memory state, so that new threads are initiated quickly. Most multipath architectures are built on top of simultaneous multithreading (SMT), because a shared register file and cache enables new threads to inherit state without any actual data movement. Multipath execution on chip multiprocessors (CMP) is largely unexplored because per-thread register files and level-1 caches are physically separated, making it difficult to quickly copy state.

Redundant execution is proposed as an enabling mechanism for dual-path execution on a CMP with two processing elements (PEs). By always executing two programs, redundant state is maintained continuously in preparation for forking. The threads redundantly and independently identify unconfident branches and explore alternative paths. When an unconfident branch resolves, one of the threads falls behind. This is remedied by passing control flow and data flow outcomes from the leading thread to the lagging thread. These are consumed as ideal branch/value predictions, enabling the lagging thread to catch up, ideally before the next unconfident branch. By being pro-active instead of reactive, redundant execution does not exhibit as much overhead as on-demand state copying.

A completely distributed, CMP-based solution will generally underperform SMT-based solutions, yet performance improvements as high as 12% are measured. Latency for communicating predictions between PEs affects performance significantly, but realistic latencies are tolerable.

1. Introduction

Multipath execution is a technique for reducing the performance penalty of mispredicted branches [1,4,8,17,18,19]. A confidence mechanism determines the likelihood that branch predictions are correct [5]. If confidence in a prediction is low, a multipath processor will fetch and execute both paths following the branch. As with conventional speculation, the predicted path is squashed if the prediction was incorrect. The misprediction penalty is reduced, however, because part of the correct path is already processed by the time the misprediction is detected.

Multipath execution typically requires support for quickly forking new register contexts. Simultaneous multithreaded processors (SMT) [16,21] are ideal because they provide multiple register contexts in a shared register file, managed by per-thread register map tables. The predicted path is part of the current thread and uses the current map table. A new thread is forked for the non-predicted path, using the map table of a free context. Initially, the new thread must inherit register state up to the point of the branch. With a shared register file, inheriting register state does not require copying register values, only pointers to the values. So, forking only requires allocating a new map table and copying current mappings to the new map table. Furthermore, single-cycle copying of map tables is already supported for branch checkpointing (e.g., shadow maps).

This paper proposes a method for limited multipath execution on chip multiprocessors (CMP). A CMP has multiple thread contexts, too, but they are distributed among processing elements (PEs) [10]. Each PE has a private register file and level-1 (L1) cache. Physically separated register files and caches make it difficult to quickly fork state. When an unconfident branch prediction is encountered, the contents of the active register file have to be copied to the register file of a free PE. Copying is slow, partly because of the distance between register files, but mainly because of limited bandwidth into and out of the register files. Forking memory state requires a sharing mechanism among distributed L1 caches. In any case, the overhead for copying state is so high that the non-predicted path may not even begin executing before the branch is resolved.

Fortunately, there is an alternative to copying state. The program can be executed redundantly on multiple PEs. This approach maintains redundant copies of register files and L1 caches continuously. In this paper, multipath execution is limited to two paths, so two copies of the program are executed redundantly on two PEs. In steady-state, the PEs fetch/execute nearly in lock-step and their state matches closely. Because their state matches, the PEs are prepared for multipath execution ahead of time, and forking is unnecessary.

The two PEs redundantly predict branch instructions and simultaneously estimate confidence. When an unconfident prediction is encountered, one of the PEs follows the not-taken path and the other follows the taken path. When the branch resolves in both PEs, the PE that followed the correct path continues, while the other PE squashes wrong-path instructions and re-directs instruction fetching to the correct path.

At this point, the thread that followed the incorrect path lags behind the other thread. The leading thread passes all of its control flow and data flow outcomes to the lagging thread via a communication queue. Outcomes are consumed by the lagging thread as always-correct branch and value “predictions” [13,15]. Assisting the lagging thread enables it to catch up with the leading thread, materializing some of the performance potential of multipath execution.

Communicating predictions seems equivalent to state copying. Whatever the interpretation, redundant execution assisted by predictions is more effective than on-demand copying. On-demand copying is reactive and incurs a high latency to start the non-predicted path. The proactive approach of passing predictions to a lagging thread eventually re-synchronizes the threads, often before multipath execution is needed again.

The coverage of branch mispredictions varies depending on the separation between the two threads. In the worst case, the lagging thread does not even fetch a mispredicted branch before it is resolved in the leading thread. The full misprediction penalty is exposed because no instructions from the non-predicted path are fetched/executed before the misprediction is detected. In the best case, the lagging thread fetches the mispredicted branch in the same cycle or a few cycles behind the leading thread. Little or none of the misprediction penalty is exposed in this case.

The CMP approach has advantages and disadvantages with respect to SMT approaches. A CMP guarantees bandwidth to the predicted path. On the other hand, there is potentially less misprediction coverage due to thread separation.

We do not advocate replacing SMT multipath execution. Instead, our method is a targeted enhancement for CMPs. CMP multipath execution can enhance the performance and/or fault tolerance of the following processor models.

- *Fault-tolerant CMPs*. An important class of fault-tolerant processors executes redundant threads in lock-step on separate processing elements or duplicated pipelines [14,20]. Computed results from both pipelines are sent to a checker, which compares results to detect transient faults. Two of the key elements of multipath execution are already supported, (1) redundant programs and (2) communication of outcomes, or "predictions", to a checker. So, built-in redundancy and communication mechanisms in fault-tolerant CMPs can be exploited for higher single-program performance.
- *Conventional CMPs*. Multipath execution can be used to enhance the single-thread performance of conventional CMPs, at the same time creating opportunities for fault tolerance due to redundant threads.
- *Slipstream processors*. We are currently investigating a unified architecture for multipath execution and slipstream execution [11,15]. Both use redundant threads that collaborate via prediction queues. Based on our experience with slipstream processors, we expect that slipstream execution and multipath execution complement each other. That is, one is likely to enhance performance when the other does not. A more robust architecture is possible by simultaneously applying slipstream and multipath execution.

2. Microarchitecture of a dual-path CMP

The microarchitecture of a CMP with two PEs and dual-path execution support is shown in Figure 1. Each PE is shown inside a shaded box. The conventional pipeline of each PE consists of a branch predictor, L1 instruction and data caches, and an execution core. The branch predictor supplies the program counter (PC) of the next predicted fetch block to the instruction cache. Instructions fetched from the cache are dispatched to the execution core, which renames instructions to a physical register file, issues them out-of-order to parallel function units and the data cache, and reorders them for retirement via a reorder buffer.

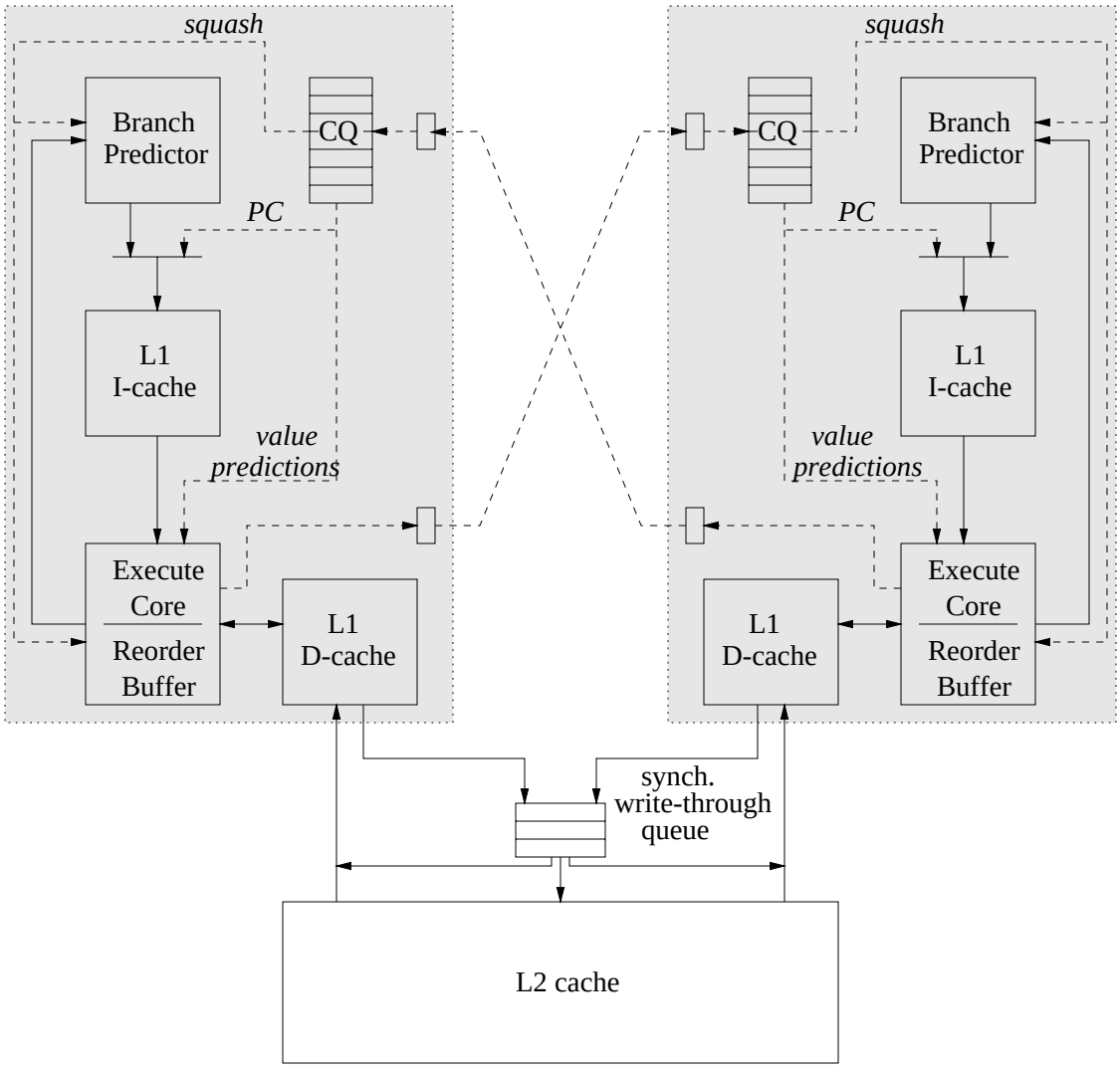

FIGURE 1. Microarchitecture of a dual-path CMP.

Dual-path execution is enabled by a new component in each PE, called the *communication queue* (CQ). New paths to and from the CQ are shown with dashed lines in Figure 1. The source operand values of completed instructions in one PE are sent to the CQ of the other PE, shown in Figure 1 with dashed arrows from each execution core to the opposite CQ. Target PCs of completed branch instructions are also sent. Communicating information between PEs may take several cycles, indicated with pipeline latches along the communication paths (small boxes in Figure 1). The contents of the CQ are used to enhance the performance of a lagging thread.

The following sub-sections describe forking (2.1), management of the communication queues (2.2), communication latency and bandwidth (2.3), methods for injecting CQ predictions into the pipeline (2.4), the memory model (2.5), and operating system support (2.6).

2.1 Forking

Forking occurs when the PEs fetch/execute opposite paths of an unconfident branch. Knowing when to fork and how to assign paths to PEs is more subtle in a distributed implementation than in a centralized one (e.g., SMT), and requires explicit coordination between the PEs (ultimately, all coordination is done through the communication queues, described in Section 2.2).

The PEs predict branches and estimate confidence redundantly, each using their own, local branch predictor. If the PEs have already forked and are pursuing different paths, then no additional forking is possible and confidence is simply disregarded. Otherwise, when a PE reaches an unconfident branch, it checks two things to decide what to do next.

1. The PE checks to see if it is the leading thread and, therefore, the first to reach (i.e., fetch) the branch instruction.
2. If it is not the first PE to reach the branch instruction, it checks to see if the branch has already resolved in the other PE.

If the PE is the first to reach the unconfident branch, then it notes the fact that it has reached a fork point but, all the same, follows the *predicted* path. That is, even though the prediction is unconfident, it is best for the leading thread to trust the branch predictor. Otherwise, we risk degrading performance with respect to a conventional single-threaded processor. The branch predictor is more often correct than incorrect when the prediction is unconfident. Also, consider that the lagging thread may be too far behind and incapable of reaching the fork point before the branch resolves. In this case, trusting the branch predictor is especially crucial.

If the PE is the second one to reach the unconfident branch, and the branch has not yet resolved in the other PE, then the PE notes the fact that it has reached a fork point and follows the *non-predicted* path (completing the forking process). If the branch has been resolved by the other PE, however, then there is no choice in the matter — a known-correct branch prediction should be available in the CQ, put there by the other PE.

The PE gets its information about which thread is ahead and which branches are resolved from its communication queue. In a completely distributed system, communication of any kind — control signals and datapaths — incurs a delay between PEs. The implications of delayed information is discussed in depth in Section 2.3, but here we point out a modification to the forking algorithm that is necessary because of delayed information.

Specifically, given the algorithm above, when the PEs are “tied” or very close in time, neither will follow the non-predicted path of an unconfident branch. It takes several cycles to communicate to the other PE that a branch has been reached. If the PEs fetch the branch in the same cycle or only a few cycles apart, they will not know about the other PE for some time and both PEs assume they are first to reach the branch. Therefore, both PEs assume they lead and that they must take the predicted path, preventing forking from ever occurring.

The solution to this problem is simple. A PE knows when the other PE is close enough to not make a precise determination of who is ahead. If the PEs are very close, and an unconfident branch is reached, one of the PEs always chooses the not-taken path and the other always chooses the taken path. Otherwise, if the PEs are not close and it is clear who leads, then the leading thread takes the predicted path and the lagging thread takes the non-predicted path (as described earlier).

2.2 Communication queue (CQ)

The communication queue (CQ) of a PE contains a recent history of the dynamic instruction stream produced by the opposite PE. Excluding instructions after a fork point, the CQs are redundant.

The CQ groups instructions together in fetch blocks (typically, basic blocks). This way, the PE can quickly read out all of the information for a block of instructions concurrently with fetching them from the instruction cache. A single CQ entry is shown in Figure 2. The entry contains a start PC and enough source operand values for the maximum number of instructions in a fetch block. Whether to use source operand values or destination operand values is implementation-specific. We chose source operand values because it works best with our approach for injecting value predictions into the pipeline, which is described in Section 2.4.

Also, as shown in Figure 2, there is a valid bit associated with the PC and each pair of source operand values. The corresponding valid bit is set when the CQ receives information for an instruction that completed in the other PE.

FIGURE 2. A CQ entry.

For convenience, we explain the operation of the CQ in terms of individual instructions, even though the CQ groups multiple instructions together in a single entry. Therefore, for the remainder of this section, a single entry in the CQ corresponds to a single instruction. This makes it easier to visualize the one-to-one mapping between the dynamic instruction stream and CQ entries.

Processing element PE_1 reads from its communication queue CQ_1. CQ_1 is written by the opposite PE, PE_2. Likewise, PE_2 reads from its communication queue CQ_2, which is written by PE_1.

CQ_1 and CQ_2 are managed by six pointers, as shown in the example in Figure 3: H_1, T_1, H_2, T_2, *head*, and *fork*. Both CQs have copies of all pointers.

CQ entries between H_1 and T_1 correspond to instructions fetched and not yet retired by PE_1. The entry pointed to by H_1 corresponds to the oldest instruction in PE_1. It is incremented when the oldest instruction is retired (like the head pointer of the reorder buffer). The entry pointed to by T_1 corresponds to the newest instruction in PE_1. It is incremented when a new instruction is fetched (like the tail pointer of the reorder buffer).

Also, T_1 is backed up when a branch misprediction is detected, to the entry containing the mispredicted branch (like squashing entries in the reorder buffer). Usually, a branch misprediction is detected when the branch executes locally. This is shown in Figure 1 as a squash signal from the execution core to the branch predictor. The misprediction may be detected earlier, however, by an incoming message to the CQ. This occurs when a branch outcome is received from the other PE before it is executed locally. Hence, there is another squash signal in Figure 1, from the CQ to both the branch predictor and the execution core.

H_2 and T_2 have the same meaning with respect to PE_2 as H_1 and T_1 have with respect to PE_1. That is, H_2 and T_2 correspond to the oldest and newest instructions in PE_2, respectively. H_2 is incremented when PE_2 retires its oldest instruction. T_2 is incremented when a new instruction is fetched by PE_2, and backed up to the offending entry when a branch misprediction is detected.

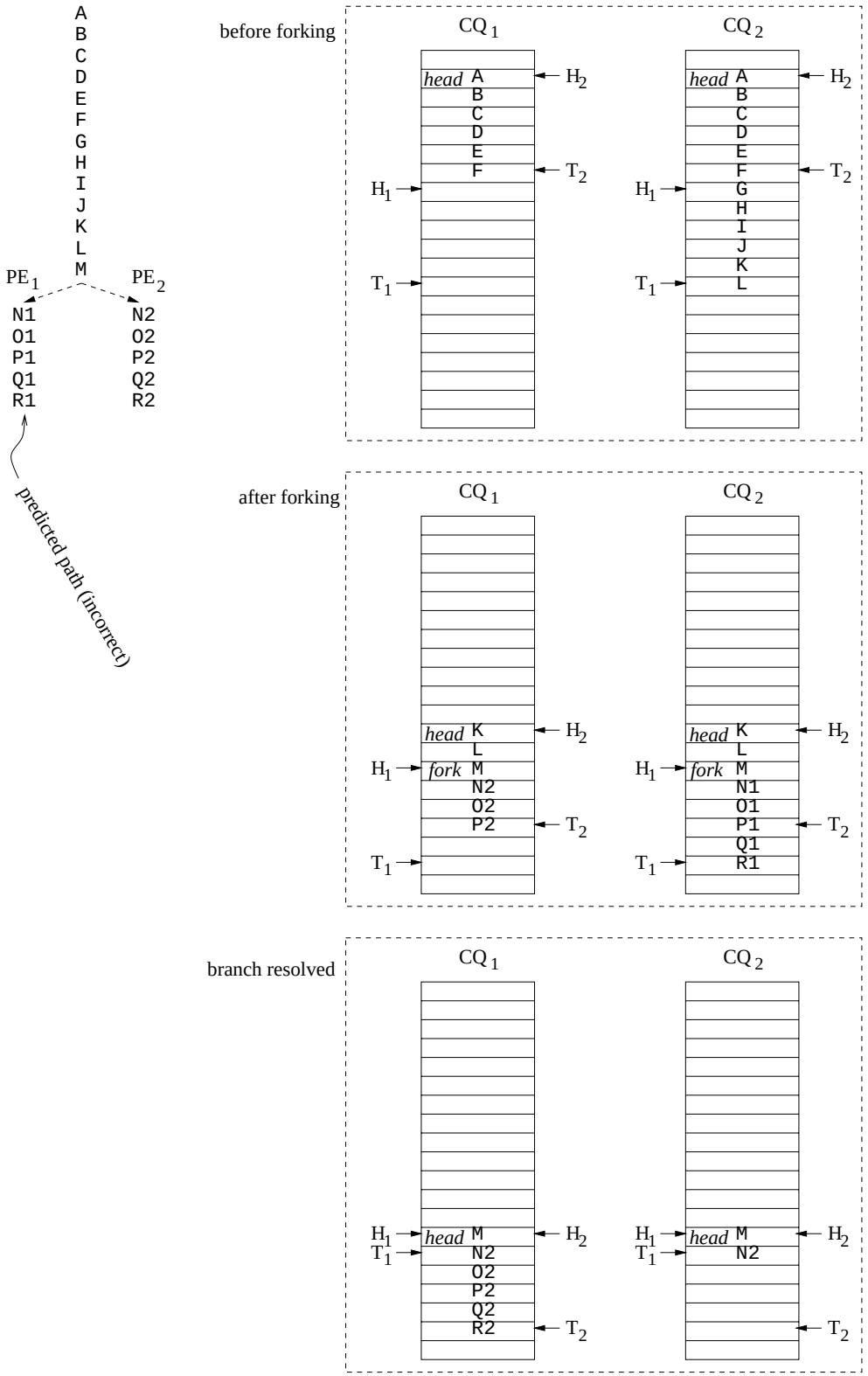

FIGURE 3. Example showing how the CQs operate.

Because CQ is a circular buffer, *H* and *T* pointers alone are not sufficient to tell which PE is ahead. The *head* pointer serves as a reference pointer. The *head* pointer points to the lagging *H* pointer, identifying the lagging thread. The *head* pointer is maintained as follows: (1) *head* always tracks the *H* pointer it is currently pointing to; (2) if *head* = H_1 = H_2, and then one of the *H* pointers advances, *head* remains behind to track whichever *H* pointer did not advance.

The example in Figure 3 shows a sequence of instructions {A-M} redundantly fetched/executed by both PEs. Instruction M, reached first by PE_1, is a mispredicted branch. It is a fork point and PE_1 fetches/executes the incorrect path (i.e., the predicted path), instructions {N1-R1}. PE_2 fetches/executes the correct path (i.e., the non-predicted path), instructions {N2-R2}. To the right of the instruction sequence are snapshots of the CQs at three points in time: before forking, after forking, and after the branch is resolved.

Before forking, *head* = H_2 and that indicates PE_2 needs to catch up to PE_1. Fortunately, CQ_2 contains predictions for six instructions {A-F} already retired by PE_1. (We know {A-F} have been retired by PE_1 because they are between *head* and H_1). CQ_2 may even have predictions for six additional instructions {G-L} (in the entries between H_1 and T_1), depending on whether they have completed yet in PE_1. In summary, entries in CQ_2 between *head* and T_1 correspond to instructions fetched and possibly executed by PE_1, and are available as predictions for PE_2 to enhance its performance.

On the other hand, before forking, CQ_1 does not provide predictions to PE_1 simply because PE_2 has not reached the same point in the program. Actually, in this example, predictions may exist in CQ_1, but only for instructions already retired by PE_1 — {A-F}, in the entries between *head* and T_2.

It is straightforward to generalize the method for reading predictions from the CQ. During the instruction fetch stage, PE_x reads the entry in CQ_x that T_x points to. A prediction is available if T_x is logically between *head* and T_y, and the valid bit of entry T_x is set.

Valid bits must be properly maintained to ensure entries are consumed only when they contain predictions, and as soon as the predictions become available. When T_x advances, indicating that a new instruction has been fetched by PE_x, PE_y resets the valid bit of the entry in CQ_y pointed to by the updated T_x pointer. It is in this entry of CQ_y that PE_x will eventually write the values for the instruction, when it completes. To ensure that the values are eventually written to the entry, the entry number (T_x) must be recorded with the instruction as it flows through the pipeline in PE_x (like a reorder buffer tag). Then, when the instruction completes, it knows to which CQ_y entry its values should be routed. When the values arrive at CQ_y, the entry's valid bit is set.

We now turn to the second snapshot in Figure 3. In the second snapshot, we can see PE_2 has nearly caught up to PE_1 (thanks in part to predictions), because H_2 is only slightly behind H_1. Also, both PEs have forked. PE_1 reached the unconfident branch first, therefore, it took the predicted path {N1-R1}, which happens to be incorrect. PE_2 reached the unconfident branch second, therefore, it took the non-predicted path {N2-R2}.

A PE knows whether it reaches a branch first (hence, which direction to fork) by comparing *T* pointers. *T* pointers reflect how far into the dynamic instruction stream the PEs have fetched. For PE_x, if T_x is between *head* and T_y, then PE_x is second to the branch. Otherwise, it is first.

Another thing, the PE which arrives second to a branch only forks if the prediction is unconfident and its CQ does not contain a prediction for the branch. If the CQ contains a prediction, it is because the other PE has already resolved the branch.

When a branch is forked, the *fork* pointer points to the corresponding entry in the CQ. For example, in Figure 3 (second snapshot), *fork* points to instruction M in both CQs. The *fork* pointer prevents the PE from reading non-redundant predictions from the CQ. If T_x is not between *head* and *fork*, then predictions should not be read from the CQ. The predictions are for instructions on the opposite path, since they were produced after the fork point by the other PE. For example, in Figure 3 (second snapshot), PE_1 fetches/executes instructions {N1-R1} but receives predictions for {N2-P2} in CQ_1. The predictions come in handy for PE_1 later, when it realizes it executed the wrong path and needs to catch up to PE_2.

The final snapshot in Figure 3 shows the state of the CQs just after the branch is resolved. PE_1 mispredicted the branch, so it re-directs instruction fetching to the correct path and backs up T_1 to just after the mispredicted branch (instruction M). Now, PE_2 has taken the lead ($T_2 > T_1$). Fortunately, PE_1 already has predictions for instructions {N2-R2} queued in CQ_1, in the entries between *head* and T_2. The predictions will assist it in catching up to PE_2, hopefully in time for the next fork point.

2.3 Inter-PE latency and bandwidth

Communication between PEs is shown in Figure 4 (dashed lines indicate inter-PE communication). Instruction values or target PCs are communicated along the indicated datapaths from one PE to the CQ of the other PE. Control signals include the *H* and *T* pointers.

The bandwidth for communicating instruction values or target PCs from one PE to the CQ of the other PE is equal to the completion bandwidth of the execution core. That is, the one-way communication bandwidth between PEs is equal to the maximum rate at which instructions complete. This is usually the superscalar issue width.

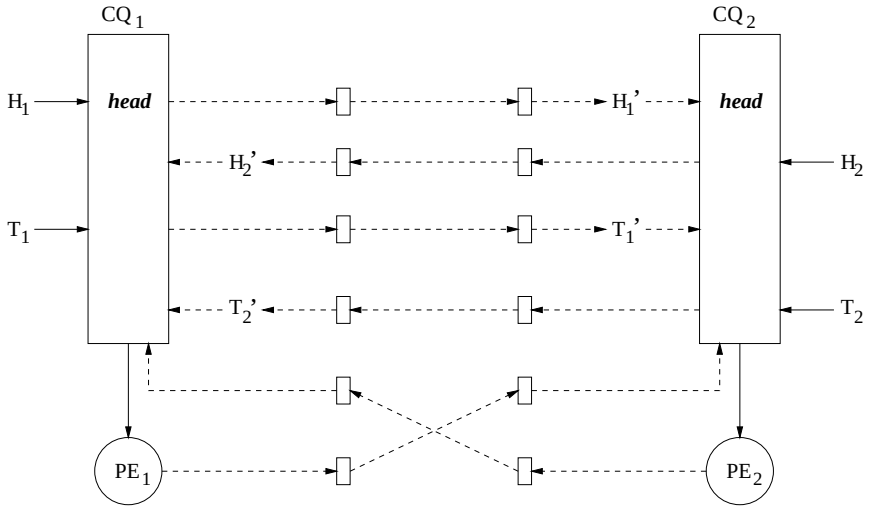

FIGURE 4. Inter-PE communication incurs a fixed delay.

Also, all communication incurs a fixed latency, indicated with pipeline latches (small boxes) in Figure 4. H_1 and T_1 are controlled by PE_1. H_2 and T_2 are controlled by PE_2. To maintain a com-

pletely distributed implementation, we do *not* assume that changes in H_1 and T_1 are immediately visible to PE_2 and, likewise, that changes in H_2 and T_2 are immediately visible to PE_1. As with communicating values and target PCs, it takes some number of cycles to propagate H_1 and T_1 from PE_1 to PE_2. This is indicated with pipeline latches between the CQs in Figure 4. So, CQ_2 uses delayed versions of H_1 and T_1 called H_1' and T_1', respectively. Likewise, CQ_1 uses delayed versions of H_2 and T_2 called H_2' and T_2', respectively.

Recall, from Section 2.1, delay impacts the forking process. The T_1 and T_2 pointers are compared to determine which PE reached an unconfident branch first. But each PE has a delayed version of the other PE's *T* pointer. Therefore, if T_1 and T_2' (or, T_1' and T_2) are within a certain number of entries of each other — equal to the inter-PE latency — both PEs assume a tie. In the case of a tie, they choose pre-determined directions, e.g., PE_1 chooses the not-taken path and PE_2 chooses the taken path. Otherwise, if the *T* pointers are sufficiently far apart and it is clear which PE leads, then the leading PE takes the predicted path and the lagging PE takes the non-predicted path.

2.4 Injecting branch and value predictions into the pipeline

This section describes how PCs and values retrieved from the CQ are injected into the pipeline. The fetch PC from the CQ simply overrides the fetch PC from the branch predictor. Values from the CQ can be injected into the execution pipeline in a number of ways, two of which are described below.

- *Reservation stations*. "Issue queues," as opposed to "reservation stations," store only instructions, and values are retrieved from the register file during the register read stage. The issue queues can be modified to also store source operand value predictions with each instruction, similar to reservation stations. Instructions whose value predictions are available from the CQ are marked as ready-to-issue before being dispatched into the scheduler. Then, instructions and their value predictions are routed to the issue queues.
- *Value prediction register file*. Values from the CQ are written to a value prediction register file that sits alongside the physical register file in the execution core. The value prediction register file does not require complex management because values are read only once. It is managed as a circular FIFO, with support for random-access reads. When an instruction is dispatched into the issue queues, its source operand value predictions are written to the tail of the FIFO. The current tail pointer locates the value prediction. Therefore, the tail pointer is substituted for the physical register file tags of source operands, for the instruction currently being dispatched. Instructions whose values are available in the FIFO are marked as ready-to-issue before being dispatched into the scheduler, and read the values during the register read stage of the pipeline using the substitute tags as indices into the FIFO. FIFO entries are marked as invalid when they are read. When the entry pointed to by the FIFO's head pointer becomes invalid, the head pointer advances, implicitly freeing the register.

Because the two threads are redundant, values from the CQ should match those produced by execution. Therefore, validation of branch and value predictions from the CQ is not strictly required. However, adding a CQ prediction validation stage to the pipeline enhances transient fault detection capability [2,12,13]. A transient hardware fault may cause errors in either or both threads, which are manifested as CQ mispredictions (either the PE or the CQ is incorrect, depending on which thread was affected by the fault). When an error is detected, various recovery meth-

ods are possible, including trapping to the operating system to restart the program from the beginning or from a checkpoint.

A key aspect of our prediction-based communication method is that the issue logic (wakeup and select logic) of the two PEs are not linked. That is, an instruction in one PE cannot wakeup an instruction in the other PE. Values can only be consumed as instructions are brought into the instruction window, during the rename/dispatch stage. If an instruction is dispatched into the window, and later its predictions arrive in the CQ, then the predictions are not forwarded to the instruction and we missed an opportunity to enhance performance. However, the core wakeup/select logic is preserved, resulting in a less complex microarchitecture. This is only possible because redundant threads are independent and can make forward progress on their own.

2.5 Memory model

From a system-level perspective, there is only one program. That is to say, physical memory pages are not duplicated. Memory is implicitly duplicated at the L1 cache level only, in that each thread independently reads and writes its private L1 data cache.

We assume the L1 data caches use a write-through policy and the L2 cache uses a write-back policy. Because state is not duplicated (or "renamed") in the L2 cache, steps must be taken to prevent the leading thread from overwriting data not yet read by the lagging thread — a WAR hazard caused by redundant threads with delay between them. The reverse situation is also possible, where the lagging thread overwrites newer data produced by the leading thread, with older data — a type of WAW hazard.

The solution is to synchronize and combine redundant writes via a *synchronizing write-through queue*, as shown in Figure 1. It is a FIFO queue with a head and tail pointer. Writes from both PEs are sent in program order to the queue (store instructions are performed when they retire, in-order). At any given time, the queue contains writes from only the leading thread. The reason is, when the lagging thread performs a write, it will match the write at the head of the queue, and that entry is popped and written into the L2 cache. Therefore, the implementation of the queue only requires: (1) a thread id (0 or 1), identifying the thread whose writes are in the queue (this happens to be the leading thread), (2) a FIFO queue, where each entry consists of a doubleword address and the corresponding doubleword of data, and (3) snooping logic (address comparators) to intercept loads from the leading thread whose data is still in the queue instead of the L2 cache (this bypass path is shown in Figure 1).

Essentially, the synchronizing write-through queue is L2 rename storage for the leading thread. The L2 cache reflects the architectural state of the lagging thread. The L2 cache and write-through queue, combined, reflect the architectural state of the leading thread, with precedence given to the write-through queue.

2.6 Operating system support for redundancy-based dual-path execution

In the new CMP microarchitecture, there is a choice between higher job throughput and improving single-program performance via dual-path execution. We advocate allowing the operating system (O/S) to flexibly choose among different operating modes, based on throughput demands, job priorities, anticipated benefit of dual-path execution, user input, etc.

The O/S initiates dual-path execution by scheduling the program for execution on both PEs. Although not absolutely required for correctness, as a practical matter, the branch predictor and confidence tables of both PEs should be flushed to ensure redundant predictions initially.

Events that require O/S intervention but do not involve a context-switch — e.g., most system calls and lightweight synchronous exceptions — do not require synchronizing the threads and terminating redundant execution. Redundant execution (hence, dual-path execution) may continue in the system code. The only requirement is that redundant memory operations with system side-effects must be merged to prevent duplicating I/O operations. This is the same requirement for writes to the memory system in general. The synchronizing write-through queue described in Section 2.5 can be augmented to merge redundant I/O operations into one operation at the system level.

Events that require O/S intervention and involve a context-switch — e.g., asynchronous interrupts and heavyweight synchronous exceptions — are handled by synchronizing the threads, terminating one of them, and swapping the program out as usual.

3. Simulation environment

We use a detailed execution-driven simulator of a chip multiprocessor. The simulator faithfully models the architecture depicted in Figure 1 and outlined in Section 2. Retired results of both threads are checked by a functional simulator run independently and in parallel with the detailed timing simulator.

Microarchitecture parameters are listed in Table 1. Each processing element is a dynamically scheduled 4-way superscalar processor with a window size of 64 instructions. Conditional branches are predicted using a 2^{16}-entry *gshare* predictor [9] with 16 bits of global history. Each entry has a 2-bit counter for the prediction and a 5-bit resetting counter [5] for confidence. The resetting counter is incremented for correct predictions and reset to 0 for incorrect predictions. The branch is unconfident if the resetting counter value is less than the threshold, 31. Indirect branches are predicted using a separate 2^{16}-entry *gshare* predictor which contains predicted targets. Return instructions are predicted using a return address stack [6] of unlimited depth.

The minimum branch misprediction penalty is 10 cycles (the Intel Itanium is 10 cycles and the Intel Pentium-4 is 20 cycles). This is the minimum number of cycles between when a branch is predicted/fetched and when it is executed. Of course, the penalty can be higher than this, depending on how long the branch waits for source operands to become available.

The size of the synchronizing write-through queue is 16 doubleword entries. The latency for retrieving data from the write-through queue is equal to the L2 cache hit latency.

Each CQ can hold predictions for up to 256 instructions. One-way communication bandwidth matches the PE completion bandwidth (4 instr./cycle). Communication latency will be varied in the experiments.

The Simplescalar [3] compiler and ISA are used. We use six benchmarks from the SPEC95 and SPEC2000 integer benchmarks (Table 2), compiled with -O3 optimization. SPEC95 benchmarks were run to completion. For the SPEC2000 benchmarks (*ref* inputs), the first billion instructions are skipped, and then 100 million instructions are simulated.

TABLE 1. Microarchitecture configuration.

conditional branch predictor & confidence mechanism	2^{16}-entry *gshare* predictor, 16 bits of global branch history
	2-bit counter for prediction, 5-bit resetting counter for confidence
	confidence threshold = 31
indirect branch predictor	identical to cond. branch predictor, but predicts indirect targets
return address predictor	return address stack (unlimited depth)
minimum misprediction penalty	10 cycles (minimum time between fetch and execution of a branch)
fetch bandwidth	fetch up to 16 sequential instructions per cycle
superscalar core	reorder buffer: 64 entries
	dispatch/issue/retire bandwidth: 4 instr/cycle
	4 fully-symmetric function units
	4 loads/stores per cycle
execution latencies	address generation = 1 cycle
	load access = 2 cycles (hit)
	integer ALU ops = 1 cycle
	complex ops = MIPS R10000 latencies
L1 instruction cache	64 KB, 4-way set-assoc., 64 B lines, LRU, 2-way interleaved
L1 data cache	64 KB, 4-way set-assoc., 64 B lines, LRU, write-through
L2 unified cache	1 MB, 8-way set-assoc., 64 B lines, LRU, write-back
memory access times	L1 instruction hit = 1 cycle
	L1 data hit = 2 cycles
	L1 miss/L2 hit = 12 cycles (minimum)
	L1 miss/L2 miss = 70 cycles (minimum)
synchronizing write-through queue	16 doubleword entries
communication queue (CQ)	256 instruction entries

TABLE 2. Benchmarks.

benchmark	suite	benchmark input	# instructions
compress	spec95	120000 e 2231	71M
go	spec95	9 9	133M
jpeg	spec95	vigo.ppm	166M
gcc	spec2K	-O3 expr.i -o expr.s	100M (skip 1st billion)
twolf	spec2K	ref	100M (skip 1st billion)
vpr	spec2K	net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2	100M (skip 1st billion)

4. Results

The dual-path processor is built on top of a CMP composed of two superscalar cores. In this paradigm, the goal is to leverage a second, otherwise unused superscalar core in a CMP to improve single-program performance. Therefore, all results are reported as the speedup of dual-path execution using two superscalar cores with respect to conventional execution on one of the cores. The core is fixed and is called the BASE configuration: BASE is a 4-way issue dynamically scheduled superscalar processor with a 64-instruction reorder buffer, as reported Section 3.

Communication latency (for passing predictions and coordinating CQs) is varied. The graph in Figure 5 shows performance improvement with respect to BASE, for communication latencies of 0 cycles, 1 cycle, 2 cycles, and 4 cycles. A 0-cycle latency means a value produced in one PE in the current cycle can be read from the other PE's CQ in the *subsequent* cycle.

The first observation from Figure 5 is that redundancy-based dual-path execution can in fact improve performance, despite not being able to copy state on-demand like an SMT-based implementation can. For the best case of 0-cycle latency, speedup ranges from 6% (*jpeg*) to 12% (*go*). *Go* is known to have especially difficult branches, so it is encouraging that the dual-path CMP can effectively improve its performance.

The second observation from Figure 5 is that communication latency has a large impact on the performance improvement of redundancy-based dual-path execution, although speedups are significant even for longer latencies. Performance improvement decreases approximately linearly with increasing latency. For a 2-cycle latency, performance improvement ranges from 3% (*jpeg*) to 9% (*go*).

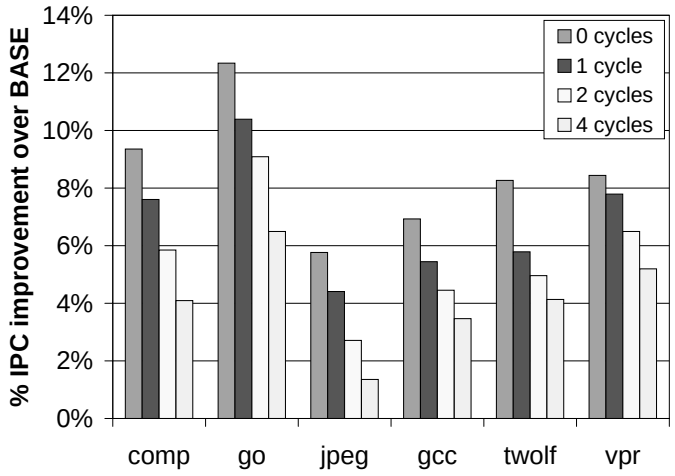

FIGURE 5. Performance of redundancy-based dual-path execution on a CMP.

Increasing latency negatively affects two aspects of the microarchitecture. First, it affects prediction passing. Second, it affects distributed mechanisms for coordinating the forking process and the CQs. For example, queue pointers are delayed more, it is increasingly difficult to determine which thread is the leading thread and anticipate ties, etc. It is unclear which aspect of the microarchitecture is more sensitive to latency and, hence, which aspect is primarily responsible for the decrease in speedup as latency increases. We plan to explore this facet in future work.

5. Related work

Uht and Sindagi proposed Disjoint Eager Execution [18], a technique in which the fetch unit follows a single flow of control for awhile, but anticipated accuracy degrades and eventually the fetch unit backs up to a prior unresolved branch and starts exploring an alternate flow of control. The term “disjoint” refers to arbitrarily jumping backwards and forwards to unresolved branches. Disjoint Eager Execution was implemented on a processor with a static instruction window.

Heil and Smith [4] and Tyson, Lick, and Farrens [17] proposed and studied selective dual-path execution, within the context of processors that quickly copy/fork state. Because the fetch unit is limited to two paths at a time, a confidence mechanism [5,17] carefully selects dynamic branches for which dual-path execution is likely to pay off.

Ahuja, Skadron, Martonosi, and Clark re-evaluated the performance potential and limits of multipath execution in light of recent advances in multithreaded processors [1]. Two detailed multipath architectures built on top of SMT processors have been proposed. These are the PolyPath Architecture by Klauser, Paithankar, and Grunwald [8] and Threaded Multiple Path Execution by Wallace, Calder, and Tullsen [19].

We are not aware of any multipath architectures built on top of CMPs, or any work that explores the implications of distributed register files and caches on implementing multipath execution.

6. Summary and future work

This paper combines redundant execution with prediction-based communication to enable dual-path execution in CMPs. Hedging an unconfident branch requires forking a new thread context for the non-predicted path. Copying state on-demand from one register file to another has too much overhead. Redundant execution maintains redundant state continuously, so that copying is unnecessary when the unconfident branch is predicted/fetched. When the unconfident branch resolves, one of the threads must re-direct fetching to the correct path and it falls behind. But, by continuously exchanging control flow and data flow outcomes between PEs via communication queues, the lagging thread can catch up, ideally before another mispredicted branch is resolved in the leading thread.

We developed a completely distributed implementation of CMP dual-path execution. Subtle aspects were highlighted. For example, coordination delays may result in both threads assuming they are in the lead when they are actually tied, preventing forking from occurring unless measures are taken to identify potential ties. As another example, it is essential that the leading thread (if there is a clear leader) select the predicted path. These and other subtle aspects are absent from centralized implementations that can rely on quick state-copying support.

For an untuned implementation, we demonstrated potential performance improvements as high as 12%, and improvements as high as 9% for anticipated inter-PE communication latencies.

The new microarchitecture is relatively unexplored. Further research is needed to attribute performance to various microarchitecture features — the confidence mechanism/threshold, communication latency, coordination latency, limited paths, no CQ-initiated instruction wakeup, etc. Future work includes experimenting with confidence thresholds, developing more precise fork-coordination methods (for example, a precise way to identify the lagging thread in close cases is to remember the thread that chose the wrong path at the last fork point), scaling the number of PEs to enable more than two paths, incorporating fault tolerance, and combining multipath and slipstream execution.

7. Acknowledgments

This research was supported by generous funding and equipment donations from Intel, Ericsson, and by NSF CAREER grant No. CCR-0092832.

8. References

[1] P. Ahuja, K. Skadron, M. Martonosi, and D. Clark. Multipath Execution: Opportunities and Limits. *12th International Conference on Supercomputing*, July 1998.

[2] T. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. *32nd Int'l Symp. on Microarchitecture*, Nov 1999.

[3] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The Simplescalar Toolset. Technical Report CS-TR-96-1308, Computer Sciences Department, University of Wisconsin - Madison, July 1996.

[4] T. Heil and J. Smith. Selective Dual Path Execution. Technical Report, Department of Electrical and Computer Engineering, University of Wisconsin - Madison, November 1996.

[5] E. Jacobsen, E. Rotenberg, and J. Smith. Assigning Confidence to Conditional Branch Predictions. *29th International Symposium on Microarchitecture*, pages 142–152, December 1996.

[6] D. Kaeli and P. Emma. Branch history table prediction of moving target branches due to subroutine returns. *18th Int'l Symp. on Computer Architecture*, May 1991.

[7] J. Kahle. Power4: A Dual-CPU Processor Chip. *Microprocessor Forum*, October 1999.

[8] A. Klauser, A. Paithankar, and D. Grunwald. Selective Eager Execution on the Polypath Architecture. *25th International Symposium on Computer Architecture*, June 1998.

[9] S. McFarling. Combining Branch Predictors. Technical Report TN-36, WRL, June 1993.

[10] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. *7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[11] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A Study of Slipstream Processors. *33rd Int'l Symposium on Microarchitecture*, Dec. 2000.

[12] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. *27th Int'l Symp. on Computer Architecture*, June 2000.

[13] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. *29th Int'l Symp. on Fault-Tolerant Computing*, June 1999.

[14] T. J. Slegel, et. al. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, March-April 1999.

[15] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. *9th Int'l Conf. on Arch. Support for Programming Languages and Operating Systems*, Nov. 2000.

[16] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *22nd Int'l Symp. on Computer Architecture*, June 1995.

[17] G. Tyson, K. Lick, and M. Farrens. Limited Dual Path Execution. Technical Report CSE-TR-346-97, Department of Electrical Engineering and Computer Science, University of Michigan - Ann Arbor, 1997.

[18] A. Uht and V. Sindagi. Disjoint Eager Execution: An Optimal Form of Speculative Execution. *28th International Symposium on Microarchitecture*, December 1995.

[19] S. Wallace, B. Calder, and D. Tullsen. Threaded Multiple Path Execution. *25th International Symposium on Computer Architecture*, June 1998.

[20] Alan Wood. Data Integrity Concepts, Features, and Technology. White paper, Tandem Division, Compaq Computer Corporation.

[21] W. Yamamoto and M. Nemirovsky. Increasing Superscalar Performance through Multistreaming. *Parallel Architectures and Compilation Techniques*, June 1995.