Multiple-Path Execution for Chip Multiprocessors

Matthew C. Chidester, Alan D. George, and Matthew A. Radlinski
HCS Research Laboratory, ECE Department, University of Florida

February 2003

Abstract—The increased dependence of clock cycle time on interconnect delay favors chip multiprocessors (CMP) for future microprocessor designs. This paper studies multiple-path execution (MPE) on a CMP to provide speedup on unmodified sequential code by exploring different paths of a conditional branch on separate processors. MPE performance due to processor complexity and count, cache and branch prediction architecture, processor-to-path allocation strategies, and limited interprocessor communication capabilities is explored. Simulation shows 12.7% speedup of SPECint95 with up to 33.5% on components with poor branch prediction accuracy using an 8-processor, 8-issue CMP with a simple mesh interconnect, realistic latencies, and limited bandwidth.

Index Terms—Chip multiprocessor, multiple-path execution, thread speculation.

1 INTRODUCTION

Advancements in integrated circuit technology over the past several decades have exponentially increased the number of transistors available on a single chip. Using these transistors, computer architects create processors with innovative features to exploit instruction-level parallelism (ILP), improving performance beyond clock-frequency scaling. However, increasing wiring delay at the silicon level and growing design and validation complexity threatens to quell this performance growth. The integration of multiple processors on a single die, known as a chip multiprocessor (CMP), has been proposed as a method to address both issues [5], [7], [12], [16], [20].

A CMP provides the opportunity to exploit parallelism beyond the ILP typically uncovered in a superscalar design and at granularities smaller than in a symmetric multiprocessor (SMP). One of the chief limitations to ILP is branch prediction accuracy. As pipeline lengths and issue widths increase, the misprediction penalty increases. In a CMP, otherwise idle CPUs can be used to explore both paths of a conditional branch, ensuring that when the branch is resolved, one CPU will have taken the correct path. This scheme is called Multiple-Path Execution (MPE). MPE

has been widely studied as a method to improve sequential program performance in superscalar and simultaneous multithreaded (SMT) designs [1], [15], [14], [27], [29].

In this paper, we propose an architecture for providing MPE on a CMP. Other MPE studies demonstrate speedups of 14% on SPECint95 with over 30% on components with hard-to-predict branches [1]. These studies conclude that MPE is feasible when the number of explored paths is limited by identifying likely mispredictions using confidence prediction [9]. Other pitfalls, such as the requirement for per-path return address stacks [1], are also well documented.

We demonstrate architectural requirements for MPE unique to CMP designs. The key limitation is found to be the on-chip interconnect. Using a simulative analysis of an idealized CMP, we explore these key architectural elements and demonstrate the bandwidth and latency requirements. Several options are proposed to reduce the interconnect dependency. Applying these techniques, we show that practical, though forward-looking, CMP designs can yield speedups similar to those found in other MPE studies.

In Section 2, we describe our proposed architecture for enabling MPE on a CMP. Section 3 describes the simulation environment used to evaluate different MPE approaches. Specific architectural tradeoffs are explored in Section 4, while the interconnect requirements are examined in Section 5. Practical limits to CMP designs are described in Section 6 and the MPE performance on several CMP implementations is compared. In Section 7, related efforts are described while Section 8 offers conclusions and opportunities for future research.

2 MULTIPLE-PATH EXECUTION ON A CMP

Traditional MPE schemes rely on idle functional units and enhanced fetch and issue techniques to execute alternate paths. A CMP can achieve similar parallelism by assigning a different path to each idle processor. Confidence prediction determines if the predicted branch direction is likely to be correct. Provided there are sufficient resources, a low-confidence branch causes a second

processor to begin executing down the not-predicted direction such that, regardless of the branch outcome, one processor follows the valid path.

A system running multiple independent tasks or programs that have been parallelized in a more conventional manner will achieve greater system-wide speedup if processors are allocated to these tasks instead of to MPE. Given the implementation advantages of a CMP, future systems will likely be comprised of multiple processors even in single-user environments for which sufficient explicit parallelism to consume all processors does not exist. MPE assumes some subset of the CMP is idle and can be allocated to explore multiple paths of one of the tasks.

In our proposed scheme, a *primary* processor executes the code exactly as a uniprocessor would, following only the predicted path. At the start of execution, all other processors on the chip are designated as *free*. When the primary processor encounters a low-confidence branch, a free processor is assigned to follow the non-predicted direction, becoming an *alternate* processor. Program order is preserved by permitting only the primary processor to commit its instructions. The committed register writeback operations are broadcast to all other processors on the chip. In this manner, alternate processors receive input dependencies for their instruction stream while free processors maintain the state necessary to become alternates.

Figure 1 shows our proposed microarchitecture for an MPE-enabled processor on a CMP. An aggressive out-of-order design with in-order front and back ends, adapted from [14] and similar to modern speculative designs, is assumed. MPE support requires four types of data to be communicated between processors: *MPE fork*, *dependency sync*, *value sync*, and *MPE resolve*.

MPE Fork. If free resources exist, the primary processor generates a path fork command in response to detecting a low-confidence, conditional branch. The command instructs a single free processor to begin executing instructions at the non-predicted path. The forked processor inserts

a specially-tagged branch instruction into its instruction stream to prevent subsequent instructions from being committed until the branch is resolved by the forking processor.

To allow the forking processor time to determine the address of the non-predicted path, path forks occur at the end of fetch; we assume a 3-cycle fetch stage. With certain interconnect topologies, alternate processors may be permitted to fork additional alternates.

<Figure 1>

Dependency Sync. Because alternate processors may fetch instructions that depend on outputs produced prior to the forked branch, the primary processor must communicate a list of register output dependencies. Dependencies are handled by inserting special instructions into the stream that await the specified values from the source processor. Renaming logic [26] enforces read-after-write (RAW) dependencies between fetched instructions and remote dependencies.

Dependency information can be transmitted continuously as instructions are decoded, or all-at-once when a fork occurs. In systems where alternate processors can fork new paths, the alternates must also communicate dependency information.

Value Sync. Maintaining synchronization between processors on the CMP comprises the largest amount of inter-processor bandwidth. This synchronization requires the primary processor to communicate any changes of architectural registers to all other processors on the chip. In the proposed design, only committed register values are synchronized. Therefore, value synchronization always occurs between the primary processor and all other processors. The free and alternate processors commit register values received from the primary processor.

Though these values may be available earlier, communication requirements are much lower if registers are synchronized in-order. Out-of-order synchronization requires a mechanism to cancel speculative data values in the case of high-confidence branch mispredictions. Also, simultaneous value communication between alternate processors and the ability for freed processors to

"resynchronize" to a path that may have proceeded along to a different state would be necessary. In-order synchronization maintains a single, global state consisting of register values produced in program order. We found the MPE performance when committing instructions in-order to be on the order of 3-5% less than that achieved when values are propagated as soon as they are known.

MPE Resolve. When the primary processor resolves a correctly-predicted branch that forked an alternate path, the alternate path must be freed. For a mispredicted branch, the primary processor is freed and the alternate becomes the new primary processor. A path resolve command matches the specially-tagged branch instruction in the alternate processor's instruction stream, notifying it of the branch outcome. If a freed processor has forked any alternate paths, they too must be freed. Freeing a processor is otherwise identical to a branch misprediction in a speculative uniprocessor: uncommitted instructions are discarded while committed values (including those received from the primary processor) are retained.

Supporting these four types of MPE data requires a dedicated interconnect between processors. The bandwidth and latency requirements of this interconnect are explored in Section 6. We assume that cache traffic occurs on a separate interconnect.

Memory dependencies are handled as a special case of register dependencies. In addition to register output dependencies, *dependency sync* data includes the full instruction (minus the opcode field) for all memory stores. In this manner, alternate and free processors insert remote store instructions into their own load/store queue. The effective address can be computed as soon as the index register is valid. Similarly, the data to be stored is inserted as soon as it is received in a subsequent *value sync*. In this manner, memory disambiguation, RAW dependencies, and speculative stores can be handled identically to the uniprocessor case with the exception that values may arrive from a remote processor rather than a local functional unit.

Alternately, memory disambiguation structures such as the Address Resolution Buffer (ARB) in [6] or Memory Disambiguation Table (MDT) in [16] could be used. In these schemes, loads are performed as early as possible and, if a dependency is later detected, execution rolls back to a point prior to the load. Our approach takes advantage of the lower-latency MPE communication to avoid costly state maintenance and rollbacks which would be required if memory dependencies were detected through the cache hierarchy.

A per-path RAS is required to prevent subroutine calls in simultaneously executing paths from polluting the stack [1]. Subroutine calls and returns are communicated in the same manner as path forks, causing free processors to push or pop the return addresses onto their RAS. Alternate processors buffer these values when executing their own instructions, instead using the RAS for subroutines in the alternate path. When the processor is freed, it recovers the RAS to the state of the primary processor as in a speculative processor [11], then applies the buffered values.

3 SIMULATION ENVIRONMENT

To evaluate the performance of MPE on a CMP, we have extended the SimpleScalar simulator [3] to produce a new timing simulator called *SimpleCMP*. SimpleCMP models a CMP with the architectural features and limitations described in the following sections to support MPE. The simulator performs trace-driven simulations of a 32-bit, MIPS-like instruction set [21], complete with register renaming, out-of-order execution, superscalar issue, and detailed cache and branch prediction logic. In addition to the CMP and MPE modifications, SimpleCMP was also modified to more closely resemble the hardware of an Alpha 21264 [13] by replacing the register update unit (RUU) with a reorder buffer and rename logic. Separate issue queues are provided for integer and floating-point instructions, and the pipeline depth matches the design of the 21264.

Table 1 shows per-processor architectural parameters. Four levels of processor complexity were compared. The 4-issue variant is modeled after the 21264 while 8-, 16-, and 32-issue

designs are extrapolated by increasing key resources. Since branch mispredictions are resolved at the end of the execute stage, the misprediction latency is a minimum of 8 cycles, but it can be much longer if the branch must wait for an input dependency before issuing.

<Table 1>

A "ones counter" branch confidence predictor [9] is employed to determine whether a branch prediction has high or low confidence. The predictor is indexed by branch address and contains a shift register representing the result of the last eight branch predictions. A set bit represents a correct decision. If two or fewer bits are set, the branch has low confidence and, if resources are available, will cause a forked path to follow both branch directions. The selected confidence predictor can be implemented simply in hardware and shows the best performance in other MPE studies [1]. Significant speedup is obtained by using ideal branch prediction that assigns low confidence to every mispredicted branch and high confidence otherwise. Since this effect is well documented [1], [15], we consider only realistic confidence prediction in this study.

We gauge performance of MPE on the simulated CMP by executing components of SPECint95 [25] with and without MPE on the processor architectures described above. All benchmarks are compiled using *gcc* 2.6.3 and the options "`-O3 -funroll-loops`". Only the integer benchmarks were simulated since the floating-point components show near-perfect branch prediction accuracies. SPEC95 was selected because simulation times become prohibitive with the larger data sets and longer runtimes of SPEC2000. Also, SPEC2000 is more memory-bound than SPEC95 [8] whereas this study is intended to focus on the effect of branch mispredictions. Table 2 summarizes the SPECint95 benchmarks and input data sets used in the simulations.

<Table 2>

To reduce simulation times, we apply the sampling scheme from [23]. A fast, functional simulation was performed for the specified number of "warmup instructions" shown in Table 2.

Only the cache and branch predictors were simulated in this stage. The following 50M instructions were simulated using a fully detailed, trace-driven pipeline simulation. Statistics were gathered only during this 50M-instruction window. The window is selected for each benchmark in such a way as to accurately represent the overall behavior of the program.

Table 2 also provides branch and confidence predictor statistics on the baseline 4-issue architecture. The branch predictor accuracy shows directional prediction accuracy of conditional branches only, and thus is somewhat lower than accuracies listed in most studies that include unconditional branches and procedure calls. Path length refers to the average number of instructions between mispredicted branches. The confidence predictor accuracy refers to the percentage of conditional branch mispredictions detected as low confidence. With unlimited resources, these branches would trigger an MPE fork and behave as a correctly predicted branch. The effective path length between mispredicted instructions is therefore increased as shown.

In most of the experimental studies that follow, only the performance results of the *gcc*, *go*, *ijpeg*, and *vortex* benchmarks are examined. The *go* and *vortex* benchmarks represent the largest and smallest path-length increases and hence the best- and worst-case MPE performance, respectively. The *gcc* benchmark represents a middle level of MPE performance while *ijpeg* is included because it has the highest uniprocessor IPC. In addition to these four, the average performance of all eight SPECint95 benchmarks is also provided.

4 ARCHITECTURAL REQUIREMENTS FOR EFFICIENT MPE

In this section, we consider the impact of several architectural elements on MPE performance. The results presented here assume an ideal interconnect with unlimited bandwidth, single-cycle latency, and crossbar connectivity. The intent of these experiments is to narrow the design space for MPE by establishing the importance of CMP size and complexity, on-chip cache hierarchy

and prediction logic, and processor-path allocation by studying idealized systems. We explore limited interconnect capacity and practical performance in subsequent sections.

4.1 Processor Count and Complexity

One of the most important design decisions regarding MPE on a CMP is determining the minimal resources required to yield effective results. In Figure 2, we show the relative improvement in instructions per cycle (IPC) for CMPs of varying sizes and issue widths when compared to a uniprocessor of the same issue width.

The first trend evidenced by Figure 2 is that *go* shows the largest increase in IPC using MPE while *vortex* shows the least (note the different y-axis scales). This result is in line with the benchmarks' branch prediction accuracy and matches results found elsewhere [1], [29].

<Figure 2>

The effect of processor count demonstrates several interesting trends. First, for 4- and 8-issue processors, most of the benchmarks yield diminished returns for more than 8 processors. At 16- and 32-issue, a 16-processor design is preferable. Due to its poor branch prediction accuracy, *go* continues to show marked improvement for up to 32 processors on all but the 4-issue design.

Perhaps the most interesting data from Figure 2 show that MPE yields a larger increase in performance on wider-issue designs, even when the processor count is held constant. This trend results from the fact that wider processors must fetch more instructions on each speculative path prior to resolving the branch. Therefore, a misprediction results in a larger performance penalty.

The results between different issue widths cannot be compared in an absolute sense because the baseline IPC is different. Also, many variables regarding issue width scaling, such as clock frequency, have not been taken into account in the configurations provided in Table 1. However, the wide-issue speedup demonstrated here is conservative. Though we assumed constant misprediction latency, the wider-issue designs would likely have a longer pipeline, hence larger

misprediction latency. Other MPE studies have demonstrated that larger misprediction latencies improve the performance of MPE [29], therefore it is likely that the IPC increase for wide-issue processors would actually be larger than shown in Figure 2.

4.2 Cache Hierarchy and Prediction Logic

When a misprediction is detected by the primary processor for a branch that forked an alternate path, the processor following the alternate path becomes the new primary processor. In this manner, the assignment of primary processor continuously migrates among all processors in the CMP. If there are a large number of processors, execution may not return to the original primary processor for many instructions. Structures that depend on locality of accesses—cache and branch prediction—will not operate effectively if they only see a subset of the instruction stream.

<Figure 3>

Figure 3 shows four alternative architectures for a CMP. The *shared-both* configuration of Figure 3d is ideal for MPE since each processor shares a common L1 cache and branch predictor. However, the *shared-none* design in Figure 3a is preferable for implementation since each processor has low-latency access to its own L1 cache and branch prediction hardware. Figures 2b and 2c represent hybrid designs of *shared-pred* and *shared-L1* with either a shared branch predictor or a shared L1 cache, respectively. In all cases, a shared L2 cache hides main memory accesses with a tolerable amount of on-chip communication latency.

4.2.1 Reflective L1 Cache

Since the physical design of *shared-none* is ideal for implementation but less suited to MPE performance, we propose a mechanism for a *reflective* L1 cache to provide the logical equivalent of *shared-L1* with the physical design of *shared-none*. To mimic the effect of a shared L1 cache, all cache lines requested by the primary processor are also loaded into the L1 cache of each free

processor. To achieve this goal, the L2 cache maintains a state table indicating whether each processor is the *primary*, an *alternate*, or *free*.

<Figure 4>

Figure 4 shows an example of such an implementation on a 6-processor CMP. The state table indicates that the primary processor is P2, P1 and P3 are alternates, while P0, P4, and P5 are free. In step 1, P2 encounters an L1 cache miss and requests the data from the L2 cache. In step 2, the L2 cache simultaneously checks its state table and fetches the requested line from the cache or main memory. Step 3 shows the fetched cache line being broadcast to the requesting processor as well as all free processors. Since a free processor does not execute instructions, there will never be a conflict requiring simultaneous transfer of two cache lines to a single processor.

4.2.2 Performance Impact

To gauge the effect of cache and branch prediction sharing, CMP architectures exhibiting the *shared-none*, *shared-pred*, *shared-L1*, and *shared-both* architectures were simulated. A fifth alternative using the *reflective-L1* design with independent branch prediction logic was also simulated. The resulting increases in IPC over a uniprocessor are shown in Figure 5. All designs used an 8-processor, 8-issue CMP, though similar trends were observed for designs with other processor counts and issue capabilities.

<Figure 5>

While some benchmarks such as *ijpeg* and *vortex* exhibit little difference among the five configurations, the general trend is that *shared-both* performs best while *shared-none* performs worst. The *shared-L1* results are closer to ideal than the *shared-pred* results, particularly for *gcc*. Therefore, the reduced cache locality has a more significant effect on performance than decreased branch prediction accuracy. This result is unsurprising since MPE is designed to reduce the number of branch mispredictions, mitigating the loss in accuracy.

Figure 5 demonstrates that a *reflective-L1* design, though not as effective as a single, shared cache, provides much of the benefit while still allowing distributed caches. On the average, the *reflective-L1* provides a 17.0% speedup compared to only 12.6% speedup with a *shared-none* design. Though *shared-both* can provide 19.0% speedup, the implementation disadvantages of such an approach make the reflective cache a clear winner.

4.3 Allocation Strategies

Allowing alternate-path processors to fork additional alternates requires a topology that support simultaneous transactions. In this section, we consider several allocation strategies that limit the degree to which alternate processors can fork new paths. We consider three types of allocation strategies: *primary only*, *fork-N*, and *eager*.

Primary Only. In this strategy, only the primary processor (following the predicted path) can fork alternates. This policy is similar to the Single Path strategy in [27]. Such a pattern is well suited to a bus topology since no communication between alternates is necessary.

Fork-N. Each processor is allowed to fork *N* alternates, regardless of whether it is the primary processor, provided there is a free processor. This approach is similar to allowing nearest-neighbor forking on a ring or torus topology without contention.

Eager. All processors can fork alternates as long as free resources remain. In [27], this approach is called Eager Execution. Supporting this policy requires crossbar connectivity on a CMP. Note that *eager* allocation is equivalent to *fork-7* in an 8-processor CMP.

<Figure 6>

Figure 6 shows the increase in IPC when MPE is applied to an 8-processor, 8-issue CMP using these allocation strategies. Ideal communication bandwidth and latency and the shared-L1 architecture are used. The results demonstrate an advantage to allowing alternate processors to fork new paths. The *primary only* scheme provides an average of 15.1% speedup while the *eager*

scheme shows a 20.9% average speedup. Allocating only one path fork per processor yields the lowest average performance increase at 13.5%. However, for all but the *go* benchmark, a *fork-2* approach is nearly as effective as *eager* but requires a much simpler topology.

One reason for the low performance of both *primary only* and *fork-1* allocation strategies is that both approaches require seven outstanding low-confidence branches along a single path to fully utilize all processors, a condition which rarely occurs. The exponential effect of *fork-2* is sufficient to occupy all processors with three outstanding branches in each path.

In other MPE studies, more sophisticated allocation techniques are explored which consider such factors as cumulative branch prediction probabilities and relative confidence levels to determine the optimum set of paths to explore with limited resources [27], [15]. Though some of these techniques provide greater performance than the *eager* approach, they rely on centralized arbitration to consider all outstanding branches. We limited our schemes to consider only those that can function with the distributed control inherent in a CMP. In practice, the allocation strategy will be determined by the CMP interconnect topology. Section 6 will consider the effect of several interconnect topologies on allocation policy.

5 COMMUNICATION REQUIREMENTS FOR MPE

Because of the increasing dominance of wiring delays in large-scale integrated circuits, the interconnect between processors of a CMP will necessarily have limited bandwidth and non-negligible latency. The ability of MPE to function on a CMP depends on the match of communication requirements to interconnect capacity. In this section, we evaluate the requirements for the on-chip interconnect to support MPE.

5.1 Bandwidth Requirements

The bandwidth required to support MPE on a CMP depends on relative amounts of the four types of synchronization data discussed in Section 2: *MPE fork*, *dependency sync*, *value sync*, and

MPE resolve. We will treat each one in turn by using statistics gathered from an 8-processor CMP with 8-issue processors.

For these experiments, per-processor bandwidth is the desired metric. Therefore, only the primary processor is permitted to fork new paths. This represents a worst-case scenario for *MPE fork* and *MPE resolve* operations because all of these commands come from a single source. The bandwidth required for *dependency sync* data was found to be similar for all topologies. *Value sync* bandwidth applies only to the primary processor and hence is independent of topology.

5.1.1 Fork and Resolve Operations

The required bandwidth for *MPE fork* commands depends upon the rate at which low-confidence branches are encountered. The effective rate is reduced when there are no free processors with which to fork. The rate of *MPE resolve* commands is identical to the fork rate.

<Table 3>

Table 3 indicates the relative frequency of low-confidence branches and *MPE fork* commands for each of the SPECint95 benchmarks. The statistics are taken from the migrating primary processor in an 8-issue, 8-processor CMP. Overall, an *MPE fork* is generated in only 10.4% of all clock cycles, though *go* generates alternate paths at over twice the average rate. The data indicates that even if resources were sufficient to allow forking on every low-confidence branch, the number of forks would not increase by a significant amount.

The results indicate that supporting a single *MPE fork* and *MPE resolve* command per cycle is sufficient for an 8-issue design. The amount of data required for an *MPE fork* is somewhat more than that of an *MPE resolve* command. Both types must identify a destination processor, requiring 3 bits for an 8-processor CMP. Each *MPE fork* command provides the 32-bit address at which to start fetching instructions. An *MPE resolve* command contains a single bit to indicate whether the destination should cancel its speculative state or become the new primary processor.

5.1.2 Dependency Information

When an alternate path is forked, the new path must have knowledge of which register values have unresolved output dependencies in the forking processor and what memory values are to be changed. Two approaches to register *dependency sync* can be taken: continuous, per-cycle dependency information and fork-time dependency communication. The number of output dependencies required on a per-cycle and fork-time basis are shown in Table 4.

<Table 4>

On average, about 4 register dependencies are generated per clock cycle while only slightly more (i.e. 4.5) are outstanding at the time of an *MPE fork*. The reason for this effect is that a relatively small set of registers accounts for the majority of all writes, resulting in a large number of write-after-write (WAW) dependencies. Figure 7 illustrates the magnitude of this effect averaged over all 8 SPEC95 integer benchmarks. A single register, R2 in this case, is the destination for 37% of all register writebacks. Together with R3, the two registers account for almost 50% of all writebacks, while over 90% are made to a set of only 12 registers.

The frequent WAW dependencies can be used to lower the rate of continuous *dependency sync* information. If a writeback occurs to a given register multiple times before a forking branch, only the latest value need be transmitted to a processor working on the alternate path. Table 4 demonstrates that only about 1.5 "new" dependencies are generated per cycle.

<Figure 7>

One significant disadvantage to the continuous approach is that free processors must receive *dependency sync* commands from any processor that may start an alternate path on it. If only the primary processor can start alternates, then continuous *dependency sync* is easily achieved.

Communicating the dependency information in either approach is accomplished by simply listing the registers that are expected to be changed. For the architecture used in this study, 7 bits

per dependency are required, though this could be lowered to as few as 5 bits if floating-point and general-purpose registers are grouped.

Memory dependencies are enforced by passing the details of each store instruction (i.e. index register, value register, and immediate offset) requiring up to 32 bits of data per store. Table 4 shows that, on average, 0.46 stores are fetched per clock cycle on an 8-issue processor. The worst case, *vortex*, still averages less than one store per cycle at 0.73. Since optimized code rarely produces temporally-adjacent stores which overlap with previous stores to the same address, all stores are communicated. If fork-time dependency communication is employed, an average of about 1.4 stores will be outstanding, though *vortex* produces over 2.1 stores per fork operation.

5.1.3 Value Synchronization

Of all the synchronization data types, the *value sync* data has the highest potential requirements. In an 8-issue processor using the 32-bit MIPS instruction set, up to 16 register values can be updated each cycle with double-word instructions (in practice, this may be limited by the number of register file input ports). Supporting this amount of interprocessor bandwidth would be prohibitive. Fortunately, the actual *value sync* bandwidth is limited by three factors:

1. average IPC
2. ratio of instructions which actually write to a register
3. percentage of WAW dependencies

The first two factors are self-explanatory. The third factor accounts for same register reuse effect that limits the number of *dependency sync* transactions. Table 5 quantifies these factors on SPECint95 using an 8-issue uniprocessor. Due to cache accesses, branch mispredictions, and true dependencies, the IPC is significantly lower than 8—about 2.6 on average. Even the highest benchmark, *ijpeg*, exhibits an IPC of only 3.8.

The ratio of instructions that write to a register is quantified by the *writes/inst* metric. This value accounts for instructions that do not write to a register at all as well as instructions that

write multiple registers. On average, there are about 0.7 register writes per instruction, though *ijpeg* has the largest ratio at almost 0.9. When this metric is combined with the IPC, the expected number of *words/cycle* is found. As Table 5 illustrates, *ijpeg* requires the most bandwidth at 3.4 words per cycle while the average is only 1.8.

<Table 5>

When per-path WAW dependencies are taken into consideration, the required bandwidth is dramatically lowered. In Table 5, the *% overlap* metric refers to the percentage of register writebacks that do not need to be transmitted due to a later write before the next path fork. Over 51% of register writes were found to overlap, though individual benchmarks varied from 30% to 71%. This effect lowers the required bandwidth to less than 1 word/cycle. While *ijpeg* shows the highest IPC and highest ratio of register writes per instruction, it also has the largest overlap factor and therefore requires only slightly above average bandwidth. With all factors considered, *vortex* requires the most bandwidth at just over 1.5 words/cycle.

The amount of data each *value sync* transaction contains is equal to the register word size plus a tag to identify the destination register. In the simulated system, the word size is 32 bits and a 7-bit tag uniquely identifies the destination.

5.1.4 MPE Performance with Limited Bandwidth

In order to allow the CMP to function in the presence of limited bandwidth, the commit phase is modified to stop committing instructions when the bandwidth is exceeded. The number of *MPE fork* and *MPE resolve* transactions is fixed to one per cycle. When more than one *MPE fork* is required per cycle, only the first is performed. If multiple *MPE resolve* commands are encountered in a single cycle, instruction commit stalls. Fork-time *dependency sync* is employed with up to 5 register dependencies and one memory dependency per cycle. Extra dependencies are queued and sent in subsequent cycles with instruction fetch stalling when the queue fills. A

queue with room for 15 register dependencies and 3 store instructions was found to limit the stall frequency to a negligible impact on performance.

We found that these minimal limitations on *MPE fork*, *MPE resolve*, and *dependency sync* did not have a significant effect on performance. Limiting the *value sync* bandwidth, however, does affect MPE performance as demonstrated in Figure 8. An 8-processor system is assumed where the *value sync* bandwidth is varied from 1 to 32 words/cycle. The bandwidth refers to the number of register values that can be transmitted and does not include extra bits to identify the destination register. Results are provided for 4-, 8-, 16-, and 32-issue processors.

<Figure 8>

The results in Figure 8 indicate that for all benchmarks, the 4- and 8-issue CMPs achieve 90% of the peak speedup with as little as two words of value sync bandwidth. For *go* and *gcc*, 93% of the potential speedup is reached with this amount of bandwidth even for 16- and 32-issue processors. Due its higher requirements, *ijpeg* achieves only 33% of its peak speedup with 16-issue processors and only two words of bandwidth; *vortex* shows a slight slowdown. However, for these two benchmarks and the SPECint95 average, a bandwidth of four words is sufficient to provide 99% of the peak performance.

Though this data shows the bandwidth requirements to be substantially lower than the theoretical maximum, the required width is still somewhat higher than the register overlapping statistics in Table 5 might indicate. We found that register writes to non-overlapping values frequently occur in bursts, such as when values are restored from the stack. Because instruction commit is stalled when the *value sync* bandwidth is exceeded, these bursts can significantly impact performance if they are not drained quickly. Maintaining a separate queue for value synchronization commands could help alleviate this problem.

5.2 Latency Requirements

In order for MPE to provide speedup, the latency of interprocessor communication must be less than the average branch misprediction latency. If the correct direction of a branch can be determined on the primary processor in less time than it would take to start an alternate, it is more efficient for the primary processor to speculate all branches in the conventional manner. Other factors, such as the in-order synchronization penalty and reduced effectiveness of the branch prediction, require that the interprocessor latency be somewhat less than the misprediction latency. In this section, we quantify the requirements for interprocessor latency.

5.2.1 MPE Performance with Increased Latency

In Figure 9, the increase in IPC when using MPE on an 8-processor, 8-issue CMP with crossbar connectivity and unlimited bandwidth is plotted for interprocessor latencies of 1 to 16 cycles. The minimum misprediction latency was also varied from 8 cycles to 24 cycles. The IPC speedup indicates the improvement over a uniprocessor with the same mispredict latency.

<Figure 9>

As expected, the combination of low interprocessor latency and high mispredict latency yields the best results while high communication latency and low mispredict latency performs disastrously. On the average, there is a significant advantage to having an interprocessor latency of 4 cycles or less with an 8-cycle mispredict latency. For mispredict latencies of 12 cycles or more, greater than 10% average speedup is attainable even with 8-cycle communication. A communication latency of 16 cycles becomes feasible with a 24-cycle mispredict latency.

In general, the data suggests that MPE will provide respectable speedup for interprocessor latencies which are between one-half and three-quarters of the mispredict penalty. At higher interprocessor communication latencies, MPE is not beneficial while at lower latencies, the performance of MPE increases drastically.

It is important to note that the actual IPC is lower for larger mispredict latencies. The purpose of these charts is to demonstrate the limits of MPE given a CMP where mispredict and interprocessor communication latencies are set by some other criteria, such as clock frequency. The IPC speedup demonstrated in Figure 9 can be achieved independently of clock frequency.

6 PRACTICAL CMP IMPLEMENTATIONS FOR MPE

In this section, all of the components from Sections 4 and 5 are combined in order to ascertain the performance of MPE on practical hardware. MPE has been shown to be communication-limited and as such, before considering the CMP as a viable architecture for MPE, communication and integration issues must be addressed. In this section, we target a theoretical 8-processor CMP featuring aggressive, 8-issue processors. First, the feasibility of such a design will be established and alternative interconnects considered. Then, simulative analysis is provided to demonstrate the performance of MPE on these realizable systems.

6.1 CMP Implementation Considerations

The exploration of MPE performance on a CMP configuration of not-yet-realized processors necessitates transistor count estimates to determine integration limits. As a baseline, we consider the Alpha 21264 microprocessor, on which the 4-issue processors used in this study are based. The 21264 consists of 15 million transistors, approximately 9 million of which comprise the core logic. The 64KB L1 instruction and data caches consume the remaining 6 million transistors [13]. Assuming quadratic scaling with issue width [4], [17], [19], an 8-issue core would require approximately 36 million transistors. The L1 cache sizes remain 64KB, and a multi-banked approach is assumed to provide increased fetch width without a significant increase in area, resulting in a total per-processor transistor count of approximately 42 million.

Throughout these experiments, the simulated L2 cache size has been held constant at 8MB regardless of CPU count or issue width. This memory requires approximately 500 million storage

transistors and an estimated additional 100 million transistors for tags and control. Together with 8 processors, a total of 946 million transistors are required.

The latency for an interprocessor communication will depend on the proximity of the source processor to the destination. A processor with 42 million transistors comprises roughly 4.2% of a billion-transistor chip. Matzke [18] argues that in a billion-transistor, 0.1 μm process chip, 16% of the chip is reachable in one clock cycle, indicating that any two adjacent processors can communicate with each other in one cycle. Similarly, the entire die may be spanned in 8 cycles.

The actual interprocessor communication latency will be dependent on topology. Figure 10 shows five potential topologies for the MPE interconnect: *bus*, *ring*, *mesh*, *torus*, and *crossbar*. The ring and torus configurations can include either uni- or bi-directional links (note that for the simple 8-node torus in Figure 10, only the horizontal links can be unidirectional). Table 6 compares the five topologies in terms of estimated latencies and restrictions on the fork tree.

It is assumed that the bus and crossbar networks would have the highest latency at 8 cycles due to the need to span a large portion of the chip. The ring and mesh networks would allow single-cycle communication with neighboring nodes. For the torus, a folded design requires longer links to accommodate the wrap-around signal, so a 2-cycle link latency is allowed.

Though all nodes must eventually receive the *value sync* data from the primary processor, a pipelined approach is employed in the ring, mesh, and torus networks wherein nodes closest to the primary processor receive the data sooner than those which are farther away. The worst-case latency is provided in Table 6 to indicate when the farthest node receives this data.

<Figure 10>

<Table 6>

Finally, the topology places restrictions on which processors can start alternate paths. Since the bus network does not support concurrent communication, only the primary processor can start

alternate paths. For the ring, mesh, and torus networks, any processor can start an alternate path on a neighboring processor, provided that it is free. The crossbar allows any processor to start an alternate path on a free processor. If multiple processors contend for a single free processor in the same cycle, the free processor selects the lowest numbered processor and notifies the others that the fork request has failed.

Table 6 shows the maximum number of new paths that the primary and alternate processors may fork provided there is no contention. Note that for a mesh, this number depends on whether the processor is located on an edge or in a corner.

6.2 Practical MPE Performance

In this section, the performance of MPE is presented when all practical limitations are provided. An 8-processor, 8-issue design is used with the reflective L1 cache architecture and two words per cycle of *value sync* bandwidth. The combined bandwidth for all MPE communications would require under 256 bits per cycle, similar to the bandwidth employed for on-die caches. The latency of the interconnect depends on the topology as shown in Table 6.

The topology also determines the allocation strategy. A bus topology results in *primary only* allocation. A unidirectional ring is equivalent to the *fork-1* strategy. The bi-directional ring is similar to *fork-1* but the primary processor can fork two paths. A unidirectional torus is similar to *fork-2* except that there is contention for free processors between different paths. The mesh and bi-directional torus also approximate *fork-2* but allow the primary processor to start three alternates. A crossbar allows *eager* path allocation.

<Figure 11>

Figure 11 compares the speedup obtained through MPE for each topology over an 8-issue uniprocessor. Results for all SPECint95 benchmarks are provided along with average speedup. The results show that the high latency of the bus and crossbar topologies negate any performance

increase from MPE despite the fact that the crossbar scheme provides the best-performing allocation scheme. The low latency afforded by a unidirectional ring topology allows a 6.7% average speedup with almost 19% on *go*. Allowing the primary processor a second path fork via a bi-directional ring increases the average speedup to 11.6%. This second fork allows the remaining processors to be occupied with half the number of low-confidence branches per path.

The highest overall speedup is provided by using a mesh interconnect—12.7% on average, 33.5% on *go*. The higher latency of the torus interconnects results in decreased speedup with the exception of the bi-directional torus on the *compress* and *go* benchmarks. Though not shown, a bi-directional torus slightly outperforms the mesh across the board if single-cycle latencies are achievable. Also, for a large number of processors, the bi-directional torus has additional advantages over a mesh due to the fact that there are no edge nodes.

The worst performing benchmark, *vortex*, shows no speedup whatsoever for the ring and torus topologies. All other topologies result in a slight slowdown as the inherently high branch prediction accuracy affords little opportunity for MPE to recoup the performance lost due to reduced cache hit rate and branch prediction accuracy.

7 BACKGROUND

Many techniques to reduce the number of conditional branches have been proposed to improve the performance of modern ILP processors in light of imperfect branch prediction accuracy. Conditional [30] and predicated [2], [10] instructions use the compiler to reduce conditional branches. Both techniques rely on special tags to allow an instruction to execute but discard the result if an associated condition is not met. An advantage to this approach is that a misprediction causes only some instructions to be discarded rather than generating a complete pipeline flush. However, compiler support is required to identify paths that are short enough to use such a

scheme. Instruction reuse [24] is another method to reduce the pipeline-flush penalty that does not require compiler support but still relies on short path lengths.

For longer path lengths, techniques to explore both the predicted and the non-predicted path have been developed. One of the first such MPE techniques is Disjoint Eager Execution (DEE) [27]. DEE employs a novel processor architecture to track the simultaneous execution of instructions from multiple conditional paths in a program. In DEE, the compiler provides static prediction probabilities to determine which branches should execute both possible paths. Selective Eager Execution (SEE) [15] extends DEE to a more traditional superscalar processor and employs confidence prediction. Special tags associate each instruction with a path, allowing mispredictions to flush only those instructions that are on an incorrect path.

Other approaches to MPE use the hardware of a simultaneous multithreaded (SMT) processor to track different paths [1], [29]. When distinct threads of execution are unable to fully use all of the functional units, one or more alternate paths are generated. Efficient partitioning of instruction fetch bandwidth on an SMT to support MPE is explored in [14]. However, these studies do not consider the effect of signaling delays on future designs.

For example, large-scale integrated circuits may not provide optimal clock frequencies for monolithic designs such as superscalar processors [18], making SEE less desirable. Similarly, architectures based on multithreaded processors do not show good signal locality in all stages. Path creation requires the working set of register values for all threads be accessible by all other threads (so that two alternate paths can read values from the parent path) or that the working set of registers is duplicated so that each path has its own copy. Either implementation may be impractical if the size of the register files is limited by the amount of chip area that can be traversed in a single clock cycle. Thus, while MPE has been shown to provide a modest

performance gain on superscalar and multithreaded hardware, the technique may not be optimal on future circuit implementations with dominant signaling delays [12].

8 CONCLUSIONS

Future processor architectures will leverage the integration of multiple CPUs on a single chip. The results in this paper demonstrate the potential for increased performance of sequential programs by using MPE on a CMP. An average speedup of 12.7% on SPECint95 was measured, with a 33.5% speedup on *go* due to its low branch prediction accuracy. This level of performance is achievable on an 8-processor, 8-issue CMP using modern branch prediction techniques and realistic branch confidence prediction.

The achieved speedup is comparable to that of other MPE studies using monolithic designs such as wide-issue superscalar and SMT processors. It is likely that a CMP will permit higher clock frequencies and shorter design time. Also, it is well established that increased ILP through issue-width scaling alone has reached the point of diminished returns [28]. MPE provides a means to increase ILP beyond issue-width scaling and independently of clock frequency.

Additionally, it has been shown that even larger speedups may be obtained on more complex systems. MPE shows increased performance with wider issue widths and longer misprediction latencies. Since the historic trend in processor design has been to increase both issue width and pipeline length, it is likely that MPE will be an even more attractive solution in the future.

A *reflective-L1* architecture was introduced to allow a CMP to maintain independent L1 caches while preserving cache locality and hit rate for effective MPE performance. It was demonstrated that the required interprocessor communication bandwidth can be manageably limited without significantly impacting performance. Furthermore, CMP-based MPE is very

conducive to topologies that limit communication to adjacent processors, allowing simpler interconnects and low latencies.

One avenue for future research would be to explore the effect of clock frequency on future designs. Since wider-issue processors would likely require a lower clock rate than the simpler processors, the performance benefit of MPE on a CMP may be even more pronounced. The performance of MPE itself can be improved in a number of ways. More sophisticated confidence prediction can be used to improve overall performance. With compiler support, instructions whose values will not be used past a branch instruction can be flagged to indicate that the value need not be communicated to alternate processors, further reducing the bandwidth requirements. The reduction in performance due to in-order value synchronization can be partially reclaimed by applying value prediction [22]. This technique would enable an alternate processor to speculatively execute instructions that depend on data produced by a remote processor.

With architectural techniques such as MPE and innovative implementations such as CMPs, it should be possible to continue the trend of simultaneous increases in processor IPC and clock frequency. In this way, the rapid advance in single-threaded processor performance can continue.

9 ACKNOWLEDGMENTS

This work was funded in part by the U.S. Department of Defense and by two NSF Graduate Fellowships (Chidester, Radlinski). Support was also provided through equipment donations from Nortel Networks and Intel Corporation.

REFERENCES

[1] P. Ahuja, K. Skadron, M. Martonosi, and D. Clark. Multipath execution: opportunities and limits. In *Proceedings of the 1998 International Conference on Supercomputing,* pp. 101-108, June 1998.

[2] D. August, D. Connors, S. Mahlke, J. Sias, K. Crozier, B. Cheng, P. Eaton, Q. Olaniran, and W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th International Symposium on Computer Architecture,* pp. 227-237, July 1998.

[3] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. Technical Report TR-1342, University of Wisconsin-Madison Computer Sciences Department, June 1997.

[4] L. Codrescu, M. Deb-Pant, T. Taha, J. Eble, S. Wills, J. Meindl. Exploring microprocessor architectures for gigascale integration. In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, pp. 242-255, 1999.

[5] L. Codrescu, D. Wills, and J. Meindl. Architecture of the Atlas chip-multiprocessor: dynamically parallelizing irregular applications. *IEEE Transactions on Computers*, 50(1), pp. 67-82, Jan. 2001.

[6] M. Franklin and G. Sohi. ARB: A hardware mechanism for dynamic memory disambiguation. *IEEE Transactions on Computers*, 45(5), pp. 51-67, May 1996.

[7] L. Hammond, B. Nayfe, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9), pp. 79-85, Sept. 1997.

[8] J. Henning. SPEC CPU2000: measuring CPU performance in the new millennium. *IEEE Computer*, 33(7), pp. 28-35, July 2000.

[9] E. Jacobsen, E. Rotenberg, and J. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 142-152, Dec. 1996.

[10] R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 100-113, Dec. 1996.

[11] S. Jourdan, J. Stark, T.-H. Hsing, and Y. Patt. Recovery requirements of branch prediction storage structures in the presence of mispredicted-path execution. *International Journal on Parallel Programming*, 25(5), pp. 363-383, Oct. 1997.

[12] S. Keckler. Fast thread communication and synchronization mechanisms for a scalable single chip multiprocessor. Ph.D. Thesis, Massachusetts Institute of Technology, June 1998.

[13] R. Kessler, E. McLellan, and D. Webb. The Alpha 21264 microprocessor architecture. In *Proceedings of the 1998 International Conference on Computer Design: VLSI in Computers and Processors*, pp. 250-259, June 1998.

[14] A. Klauser and D. Grunwald. Instruction fetch mechanisms for multipath execution processors. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp 38-47, Nov. 1999.

[15] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the PolyPath architecture. In *Proceedings of the 25th International Symposium on Computer Architecture,* pp. 250-259, July 1998.

[16] V. Krishnan and J. Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip multiprocessor. In *Proceedings of the ACM 1998 International Conference on Supercomputing*, pp. 85-92, June 1998.

[17] M. Lipasti, J. Shen. Superspeculative microarchitecture for beyond A.D. 2000. *IEEE Computer*, 30(9), pp. 59-66, Sept. 1997.

[18] D. Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, pp. 37-39, Sept. 1997.

[19] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, K. Chung. The case for a single-chip multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2-11, Oct. 1996.

[20] J. Oplinger, D. Heine, S. Liao, B. Nayfeh, M. Lam, and K. Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford University Computer Systems Lab, Feb. 1997.

[21] C. Price. *MIPS IV Instruction Set, Revision 3.1*. MIPS Technologies, Inc., Mountain View, CA, Jan. 1995.

[22] Y. Sazeides and J. Smith. The predictability of data values. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 248-258, Dec. 1997.

[23] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark. Branch prediction, instruction-window size, and cache size: performance tradeoffs and simulation techniques. *IEEE Transactions on Computers*, 48(11), pp. 1260-1281, Nov. 1999.

[24] A. Sodani and G. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 194-205, June 1997.

[25] The Standard Performance Evaluation Corporation. WWW Site. http://www.specbench.org, Dec. 1996.

[26] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1), pp. 25-33, Jan. 1967.

[27] A. Uht and V. Sindagi. Disjoint eager execution: an optimal form of speculative execution. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 313-325, Dec. 1995.

[28] D. Wall. Limits of instruction-level parallelism. Technical Report 93/9, Digital Western Research Laboratory, Nov. 1993.

[29] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. In *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 238-249, June 1998.

[30] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2), pp. 28-40, Feb. 1996.

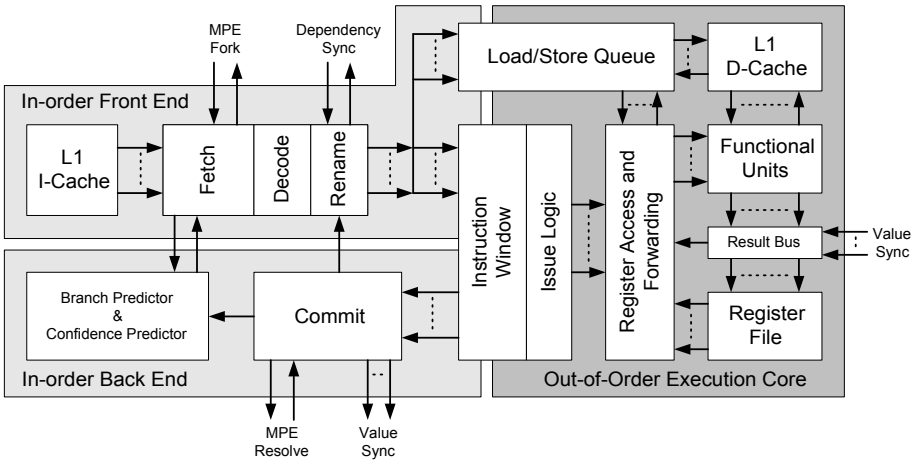


Figure 1. Processor microarchitecture with support for MPE on a CMP.

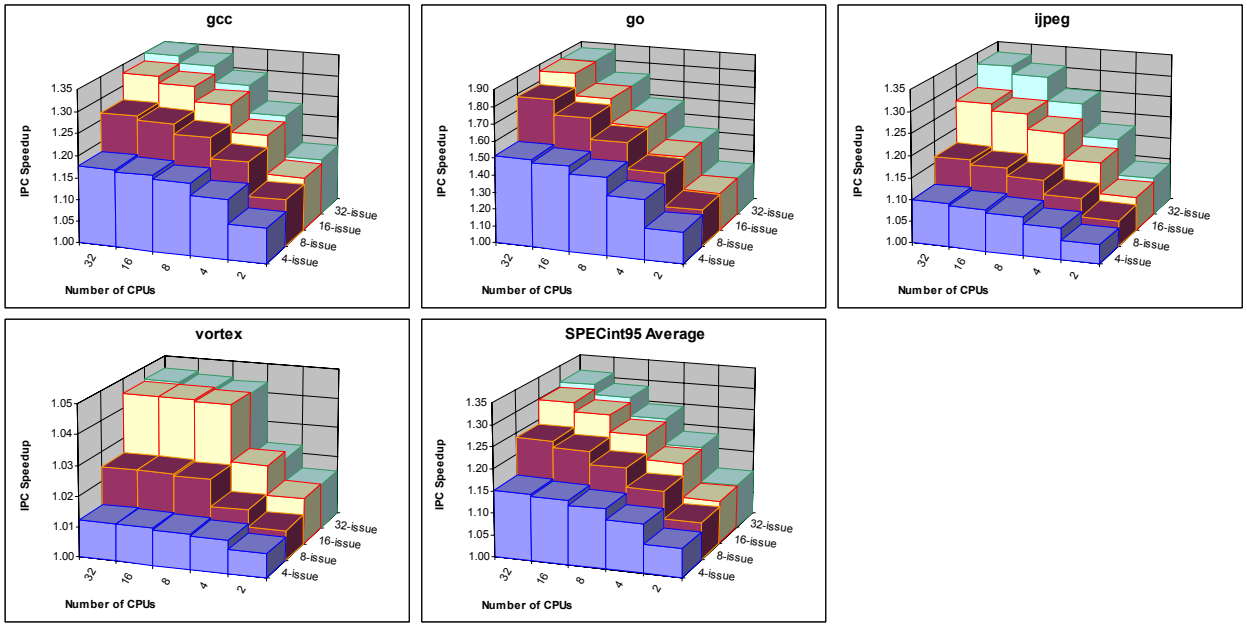


Figure 2. Effect of CPU count and complexity. Simulated system is a CMP with shared L1 cache, unlimited bandwidth, single-cycle latency, and crossbar connectivity.

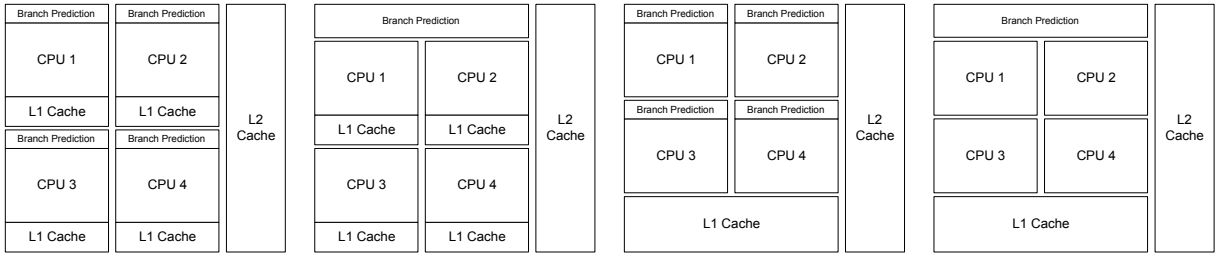


Figure 3. Alternative architectures for a CMP.

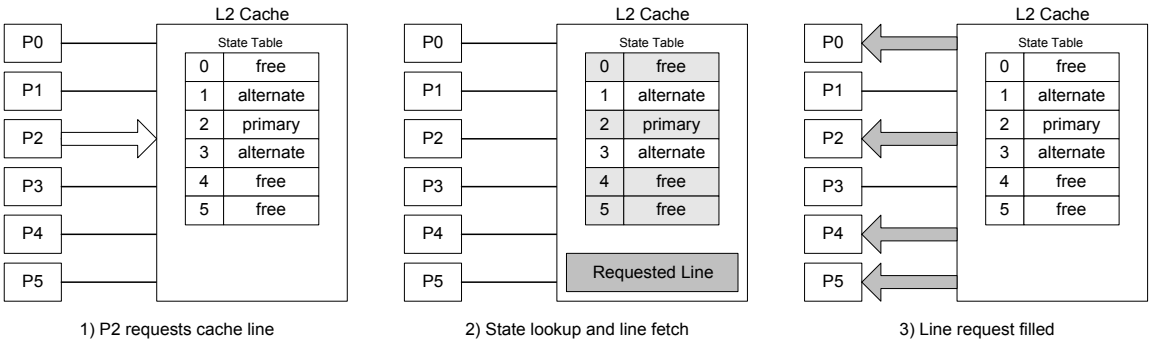


Figure 4. Example of a reflective cache architecture.

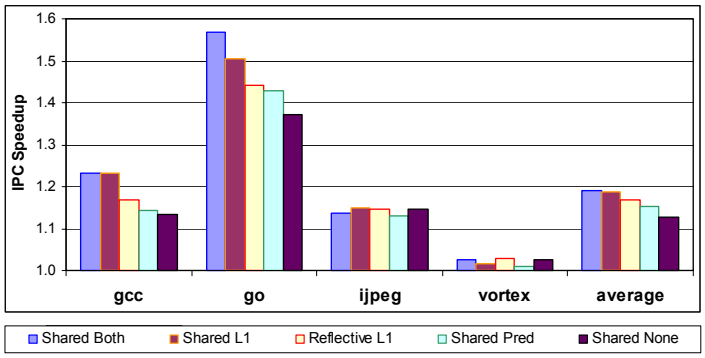


Figure 5. Effect of cache and branch prediction sharing. Simulated system is an 8-processor, 8-issue CMP with unlimited bandwidth, single-cycle latency, and crossbar connectivity.

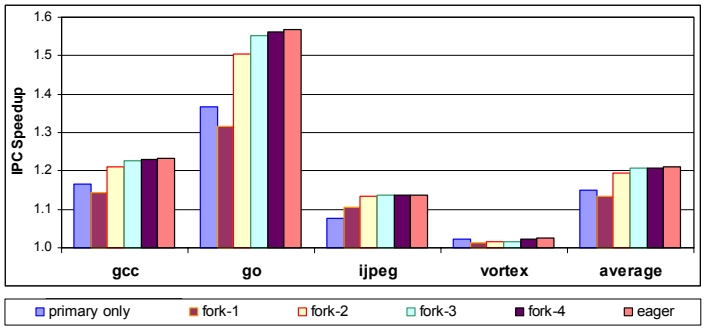


Figure 6. Effect of fork tree limitations. Simulated system is an 8-processor, 8-issue CMP with shared L1 cache, unlimited bandwidth, single-cycle latency, and crossbar connectivity.

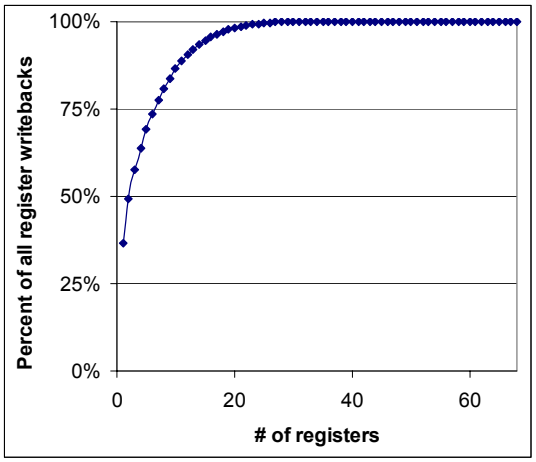


Figure 7. Register writeback distribution for SPECint95.

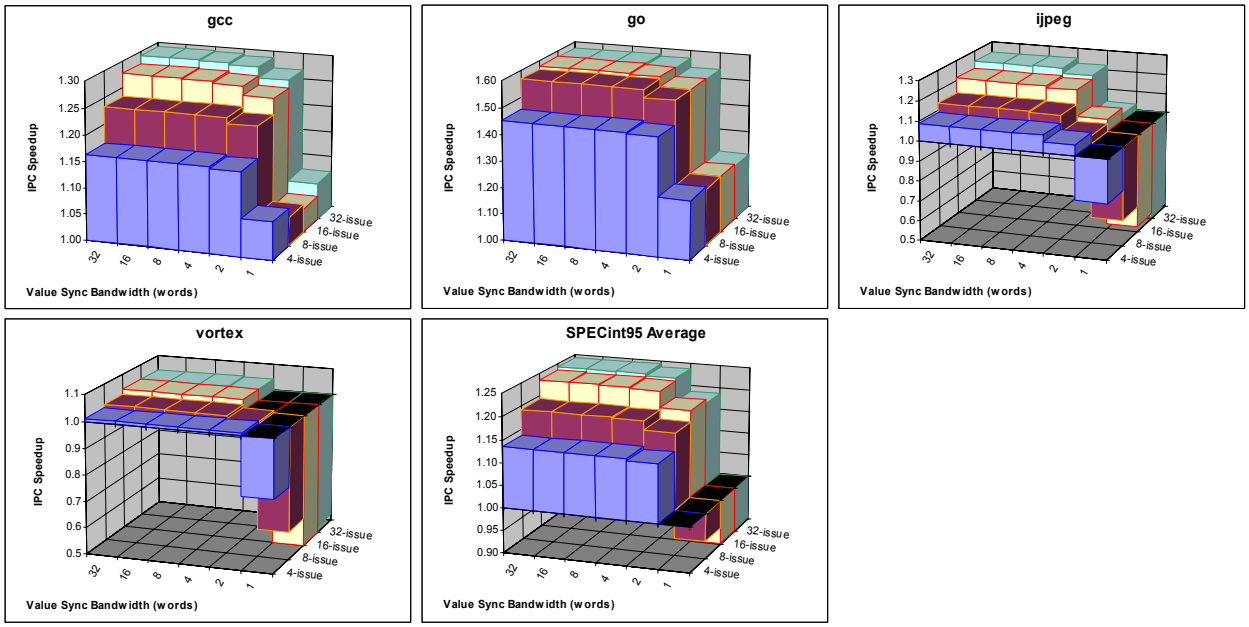


Figure 8. Effect of limited value sync capacity. Simulated system is an 8-processor CMP with shared L1 cache, single-cycle latency, and crossbar connectivity.

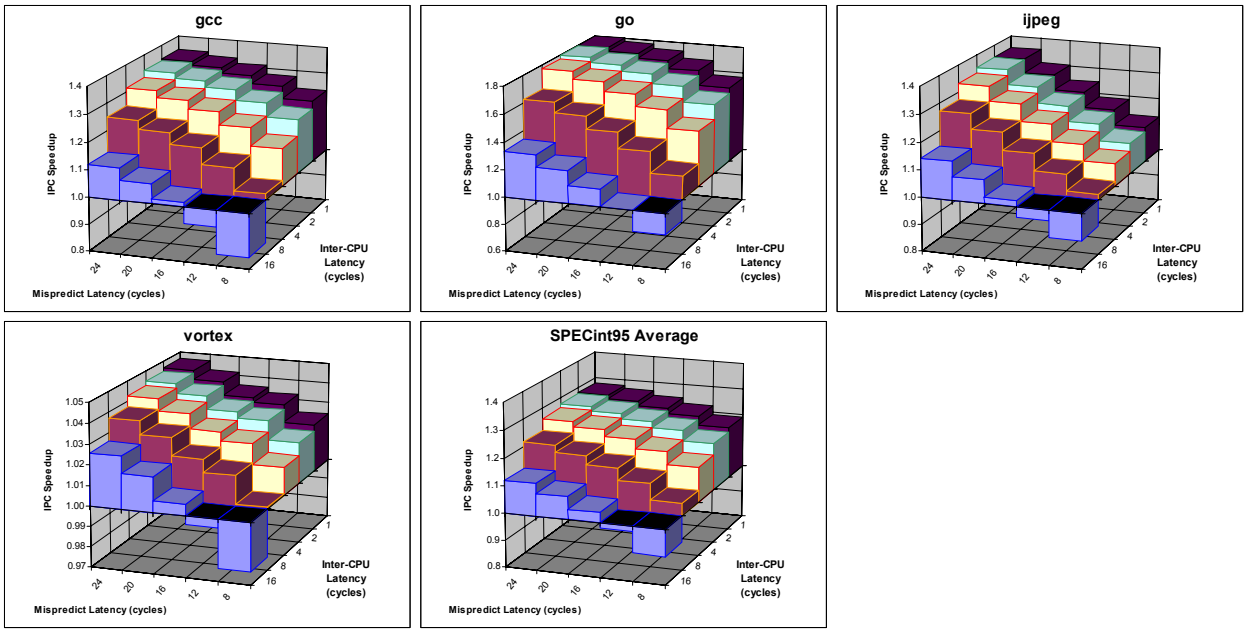
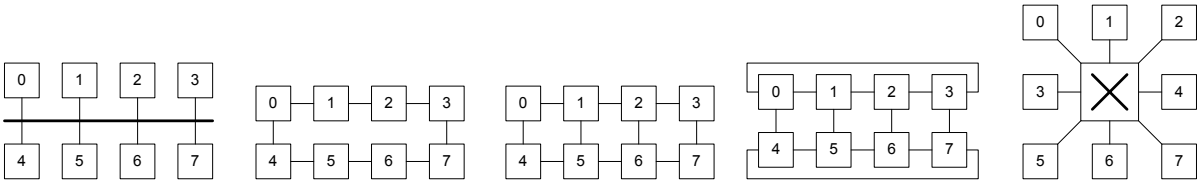


Figure 9. Effect of interprocessor and misprediction latencies. Simulated system is an 8-processor, 8-way CMP with shared L1 cache, unlimited bandwidth, and crossbar connectivity.



a) bus b) ring c) mesh d) torus e) crossbar

Figure 10. MPE network topologies for an 8-processor CMP.

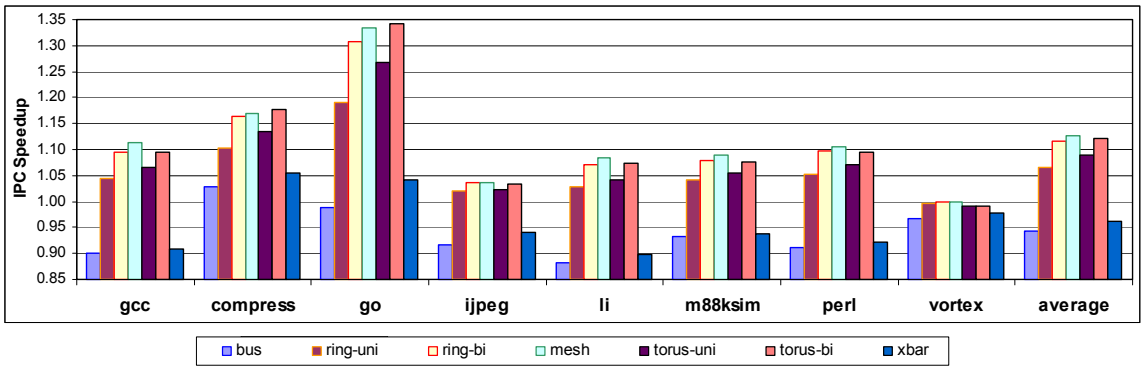


Figure 11. MPE Performance with practical topologies and latencies. Simulated system is an 8-processor, 8-way CMP with reflective L1 cache and 2 words/cycle of value sync bandwidth; latency and fork tree capability is determined by topology.

Table 1. Processor architecture parameters.

	4-issue	**8-issue**	**16-issue**	**32-issue**
Fetch/Issue/Retire Width	4 instr.	8 instr.	16 instr.	32 instr.
Rename Registers	128	256	512	1024
Integer Issue Queue	64 instr.	128 instr.	256 instr.	512 instr.
Load/Store Issue Queue	32 instr.	64 instr.	128 instr.	256 instr.
Integer ALUs	4	8	16	32
Memory Ports	2	4	8	16
Branch Predictions/cycle	1	2	4	8
Branch Predictor	*Global*: 12-bit history, 4k × 2-bit saturating counters *Local*: 1k × 10-bit history, 1k × 2-bit saturating counters *Select*: 12-bit global history, 4k × 2-bit saturating counters			
Branch Target Buffer	2048 entries, 2-way associative			
Return Address Stack	32 entries			
Minimum Misprediction Latency	8 cycles			
Branch Confidence Predictor	Ones counter, 2048 entries, 8 bits/entry, threshold 2			
L1 I-Cache	64 kB, 2-way associative, 32-byte lines, 1-cycle latency			
L1 D-Cache	64 kB, 2-way associative, 32-byte lines, 1-cycle latency			
Unified L2 Cache	8 MB, 4-way associative, 32-byte lines, 12-cycle latency			
Memory Access Latency	100 cycles, 8 bytes/cycle			

Table 2. SPECint95 benchmarks parameters.

Benchmark	Input	Warmup Instr.	Branch Predictor		Confidence Predictor		
			Accuracy	Path Length	Accuracy	Path Length	% Increase
compress	bigtest.in	2576 M	87.12 %	58	76.76 %	250	331 %
gcc	cccpi.i	221 M	86.26 %	47	61.78 %	122	160 %
go	9stone21	926 M	75.40 %	37	80.28 %	185	400 %
ijpeg	vigo.ppm	824 M	87.75 %	161	75.44 %	655	307 %
li	*.lsp	271 M	93.41 %	100	56.45 %	229	129 %
m88ksim	test	26 M	95.36 %	126	53.90 %	272	116 %
perl	scrabbl.pl	601 M	93.13 %	105	50.67 %	213	103 %
vortex	test	2451 M	98.80 %	635	46.22 %	1181	86 %

Table 3. MPE fork requirements for an 8-processor, 8-issue CMP.

Benchmark	Fraction of Cycles	
	Low-conf Branch	MPE Path Fork
compress	13.8 %	13.5 %
gcc	16.8 %	13.6 %
go	24.3 %	23.2 %
ijpeg	6.3 %	6.3 %
li	10.0 %	9.9 %
m88ksim	6.6 %	6.4 %
perl	8.6 %	8.5 %
vortex	1.9 %	1.9 %
Average	11.0 %	10.4 %

Table 4. Dependency synchronization requirements for an 8-issue processor.

Benchmark	Output Dependencies/Cycle		Dependencies at Fork Time	Stores/Cycle	Stores at Fork Time
	All	New			
compress	3.204	1.702	4.372	0.479	1.230
gcc	6.074	1.563	4.018	0.357	0.918
go	6.651	2.203	5.506	0.279	0.692
ijpeg	4.975	1.271	5.203	0.351	1.436
li	3.009	1.484	4.236	0.483	1.380
m88ksim	3.318	1.104	4.267	0.348	1.343
perl	3.021	1.433	4.162	0.663	1.924
vortex	2.474	1.628	4.759	0.734	2.146
Average	4.091	1.549	4.565	0.462	1.361

Table 5. Value synchronization requirements for an 8-issue processor.

Benchmark	IPC	All Register Writes		Latest Write Only	
		writes/inst	words/cycle	% overlap	words/cycle
compress	1.921	0.641	1.231	34.1 %	0.811
gcc	1.822	0.704	1.283	57.5 %	0.545
go	1.879	0.784	1.473	49.2 %	0.749
ijpeg	3.808	0.881	3.355	70.9 %	0.977
li	2.398	0.640	1.535	40.6 %	0.912
m88ksim	2.798	0.706	1.975	62.2 %	0.747
perl	2.633	0.645	1.698	44.6 %	0.941
vortex	3.401	0.645	2.194	30.3 %	1.530
Average	2.583	0.694	1.843	51.1 %	0.902

Table 6. Comparison of topology alternatives for MPE on an 8-processor CMP.

Topology	Links	Latency		Maximum Path Forks	
		Per Link	Maximum	Primary	Alternate
bus	bi-directional	8 cycles	8 cycles	7	0
ring	unidirectional	1 cycle	7 cycles	1	1
ring	bi-directional	1 cycle	4 cycles	2	1
mesh	bi-directional	1 cycle	4 cycles	2 or 3	1 or 2
torus	unidirectional	2 cycles	8 cycles	2	2
torus	bi-directional	2 cycles	6 cycles	3	2
crossbar	bi-directional	8 cycles	8 cycles	7	6