Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window

Huiyang Zhou
School of Computer Science, University of Central Florida
zhou@cs.ucf.edu

Abstract

Current integration trends embrace the prosperity of single-chip multi-core processors. Although multi-core processors deliver significantly improved system throughput, single-thread performance is not addressed. In this paper, we propose a new execution paradigm that utilizes multi-cores on a single chip collaboratively to achieve high performance for single-thread memory-intensive workloads while maintaining the flexibility to support multithreaded applications.

The proposed execution paradigm, dual-core execution*, consists of two superscalar cores (a front and back processor) coupled with a queue. The front processor fetches and preprocesses instruction streams and retires processed instructions into the queue for the back processor to consume. The front processor executes instructions as usual except for cache-missing loads, which produce an invalid value instead of blocking the pipeline. As a result, the front processor runs far ahead to warm up the data caches and fix branch mispredictions for the back processor. In-flight instructions are distributed in the front processor, the queue, and the back processor, forming a very large instruction window for single-thread out-of-order execution. The proposed architecture incurs only minor hardware changes and does not require* any *large centralized structures such as large register files, issue queues, load/store queues, or reorder buffers. Experimental results show remarkable latency hiding capabilities of the proposed architecture, even outperforming more complex single-thread processors with much larger instruction windows than the front or back processor.*

1. Introduction

With current integration trends, single-chip multi-core or chip multiprocessor (CMP) architectures are increasingly adopted to deliver high system throughput. As current CMP architectures only exploit explicit parallelism, however, single-thread performance is not enhanced and idle processor cores are resulting if a system lacks sufficient parallel tasks. In this paper, we propose a new execution paradigm to utilize multi-cores on a single chip collaboratively to improve the performance for single-thread workloads while maintaining the flexibility to support multithreaded applications.

One of the most significant obstacles to single-thread performance is the memory wall problem [39]: the widening speed gap between memory and processor cores considerably undermines the performance of current microprocessors even with carefully designed memory hierarchy and prefetching mechanisms. Out-of-order execution can successfully hide long latencies if there are enough independent instructions to process [1],[11],[18],[20]. With the projected memory access latency being as high as hundreds of processor clock cycles, an instruction window needs to be very large to keep track of a high number of in-flight instructions. Recently, there has been active research toward such a goal, including large issue queues [5], large register files [3],[41], scalable load/store queues [26],[31], approaches to eliminate the centralized reorder buffer (ROB) using checkpoint and recovery [1],[11],[12], and non-blocking continual flow pipelines [36]. The enlarged instruction window, on the other hand, often results in extra pipeline stages to accommodate the latency requirement to access those enlarged structures, which incur higher costs for branch mispredictions.

This paper, however, takes a fundamentally different approach. Instead of scaling current superscalar designs, we propose a novel way to utilize multi-cores on a single chip collaboratively to construct a large, distributed instruction window while eliminating the necessity for *any large centralized structures*. Figure 1 presents a high-level overview of the proposed scheme, named *dual-core execution* (DCE) as it is built upon two superscalar cores coupled with a queue.

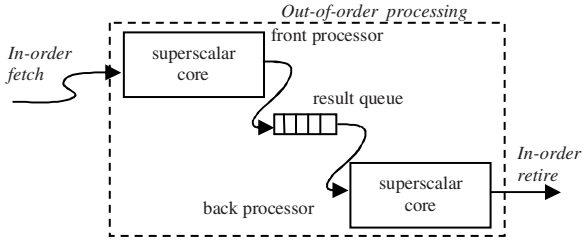

Figure 1. A high-level overview of dual-core execution (DCE): in-order fetch, in-order retire, and out-of-order processing.

The first superscalar core in Figure 1, called the *front processor*, fetches an instruction stream in order and executes instructions in its normal manner except for those load instructions resulting in a long-latency cache

Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)
1089-795X/05 $20.00 © 2005 IEEE

miss. An invalid value is used as the fetched data to avoid the cache-missing load blocking the pipeline, similar to the run-ahead execution mode in [13],[25]. When instructions retire (in order) from the front processor, they are inserted into the *result queue* and will *not* update the memory. The second superscalar core, called the *back processor*, consumes the preprocessed instruction stream from the result queue and provides the precise program state (i.e., the architectural register file, program counter, and memory state) at its retirement stage. In DCE, the front processor benefits the back processor in two major ways: (1) a highly accurate and continuous instruction stream as the front processor resolves most branch mispredictions during its preprocessing, and (2) the warmed up data caches as the cache misses initiated by the front processor become prefetches for the back processor. The front processor runs far ahead of the back processor since it is not stalled by long-latency cache misses (i.e., a virtually ideal L2 cache) and the back processor also runs faster with the assists from the front processor.

In a high-level view of DCE as shown in Figure 1, instructions are fetched and retired in-order, the same as any standard superscalar processor. The two processors and the result queue keep a large number of in-flight instructions, forming a very large *distributed* instruction window. Moreover, DCE provides an interesting *non-uniform* way to handle branches. The branches that depend on short latency operations are resolved promptly at the front processor while only the branches depending on cache misses are deferred to the back processor. Early branch resolution is also proposed in out-of-order commit processors using checkpointing and early release of ROB entries [11],[12]. As DCE is built upon relatively simple cores, such non-uniform branch handling is more efficient compared to an upsized single-thread processor with a deeper pipeline. In addition, as DCE does not need any centralized rename-map-table checkpoints, the number of outstanding branches no longer limits the instruction window size.

In large-window processors including those formed with checkpointing, aggressive memory disambiguation becomes critical as each misprediction affects a large number of in-flight instructions and potentially incurs high misprediction penalty. DCE, in contrast, has an appealing feature that it can compensate conservative disambiguation schemes to achieve similar performance to DCE with more aggressive ones (see Section 5.6).

The remainder of the paper is organized as follows. Section 2 addresses related work and highlights the differences between some related research and our work. The detailed design of DCE is described in Section 3. The simulation methodology is presented in Section 4 and the experimental results are discussed in Section 5. Section 6 concludes the paper and discusses future work.

2. Related Work

Dual-core execution (DCE) is motivated mainly from two categories of research work: run-ahead execution and leader/follower architectures.

2.1. DCE and run-ahead execution

In run-ahead execution [13],[25], when an instruction window is blocked by a long latency cache miss, the state of the processor is checkpointed and the processor enters the 'run-ahead' mode by providing an invalid result for the blocking instruction and letting it graduate from the instruction window. In this way, the processor can continue to fetch, execute, and pseudo retire instructions (i.e., retire instructions without updating the architectural state). When the blocking instruction completes, the processor returns to the 'normal' mode by restoring the checkpointed state. The instructions executed in the 'run-ahead' mode will be re-fetched and re-executed in the 'normal' mode and such re-execution is expected to be much faster as the caches are warmed up by the execution in the 'run-ahead' mode. Since each transition from the 'run-ahead' mode to the 'normal' mode involves pipeline squashing and re-fetching, it incurs similar performance cost to a branch misprediction. Although early return from the 'run-ahead' to 'normal' mode can hide such latency, it limits the distance of effective run-ahead execution [25].

The front processor in DCE processes instructions similar to the 'run-ahead' mode but it eliminates the checkpointing and mode transitions. Compared to run-ahead execution [25], DCE is developed to overcome its two important limitations and provide the flexibility for multithreaded processing. The two limitations of run-ahead execution are: (1) Speculative execution in the 'run-ahead' mode always stops once the processor returns to the 'normal' mode even such speculative execution is on right paths and generates correct prefetch addresses. (2) For miss-dependent misses (e.g., cache misses due to the pointer-chasing code: p->next->next->…), each miss will cause the processor to enter the 'run-ahead' mode. If few instructions exist between such misses, the processor will pre-execute the same set of future instructions multiple times [23]. The first limitation affects the aggressiveness of run-ahead execution while the second one wastes the processor resources. DCE eliminates all these limitations seamlessly and achieves higher performance as discussed in Section 3 and Section 5.1.

In continual flow pipelines (CFP) [36], load misses and their dependent instructions (called slice instructions) are drained out of the issue queue and register file by using invalid values as fetched data, similar to run-ahead execution. But, unlike run-ahead execution, the slice instructions are stored in a slice processing unit rather than being retired from the pipeline and the subsequent independent instructions continue their execution

speculatively. When the cache misses are repaired, the slice instructions will re-enter the execution pipeline and commit the speculative results. In this way, the work during run-ahead execution is not discarded and there is no need to re-fetch and re-execute those instructions. To maintain such speculative data, however, CFP requires coarse-grain retirement and a large centralized load/store queue (a hierarchical store queue is proposed to reduce its latency criticality [1],[36] and a new improvement is proposed in [16]). Compared to CFP, DCE eliminates such large centralized structures and builds upon much simpler processor cores (e.g., smaller register files). The fast branch resolution at the front processor (due to its simpler, shallower pipeline) reduces the cost of most branch mispredictions. Since DCE does not need any centralized rename-map-table checkpoints, it also eliminates the complexity for estimating branch prediction confidence and creating checkpoints only for low-confidence branches, as needed in CFP. Interestingly, a recent study [24] shows that the performance benefits of reusing the results in run-ahead execution are limited and may not justify the required complexity. Our results confirm such observations from a different perspective (see Section 5.3).

It has been proposed to use value prediction to further improve the effectiveness of run-ahead execution [7],[8],[19],[42]. Similarly, DCE can also benefit from such optimizations and achieve higher performance.

2.2. DCE and leader/follower architectures

Running a program on two processors, one leading and the other following, finds its roots in decoupled architectures [34], which break a program into memory accesses and subsequent computations. In DCE, the front processor not only prefetches the data but also provides a highly accurate instruction stream by fixing branch mispredictions for the back processor. Moreover, all this is accomplished without the difficult task of partitioning the program.

Slipstream processors [27],[37] are leader/follower architectures proposed to accelerate sequential programs similar to DCE and share a similar high-level architecture: two processors connected through a queue. However, DCE and slipstream processors achieve their performance improvements in quite different ways. In slipstream processors, the A-stream runs a shorter program based on the removal of ineffectual instructions while the R-stream uses the A-stream results as predictions to make faster progress. DCE, however, relies on the front processor to accurately prefetch data into caches. Conceptually, the R-stream in slipstream processors acts as a fast follower due to the near oracle predictions from the A-stream while the A-stream is a relatively slower leader since long-latency cache-misses still block its pipeline unless they are detected ineffectual and removed from the A-stream. Therefore, it is not necessary for the R-stream to take advantage of prefetching from the A-stream to make even faster progress [29]. (In [27],[37], the A-stream and R-stream own separate program context and the A-stream has *no* prefetching effect on the R-stream). In DCE, the front processor is a much faster leader as it operates on a virtually 'ideal' L2 cache while the back processor is a slower follower. A detailed performance comparison between slipstream processing and DCE is presented in Section 5.3.

Master/Slave speculative parallelization (MSSP) [44],[45] extends slipstream processing with a compiler generated master thread (or the A-stream) and the parallelization of the R-stream. Speculative parallelization [35] can also be used in DCE to improve the back processing of DCE and is left as future research work.

"Flea-Flicker" two pass pipelining [4] is proposed to handle the uncertain latency of load instructions in in-order microarchitectures and it is closest to DCE in terms of integrating run-ahead execution and leader/follower architectures. In the Flea-Flicker design, two pipelines (A-pipe and B-pipe) are introduced and coupled with a queue. The A-pipe executes all instructions without stalling. Instructions with one or more unready source operands skip the A-pipe and are stored in the coupling queue. The B-pipe executes instructions deferred in the A-pipe and incorporates the A-pipe results. Compared to this work, DCE is based on out-of-order execution, thereby having higher latency hiding. *More importantly*, flea-flicker tries to reuse the work of the A-pipe (i.e., not discarding the work in run-ahead execution, similar to CFP [36]), while in DCE the front processor preprocesses instructions and the back processor re-executes those instructions. Such re-execution relieves the front processor of *correctness* constraints, enabling it to run further ahead with much less complexity overhead (e.g., the centralized memory order bookkeeping and the coupling result store in flea-flicker or the large centralized store queue in CFP). The elimination of such centralized structures is the reason why DCE is a much more scalable and complexity-effective design.

Pre-execution/pre-computation using multithreaded architectures [2],[10],[22],[30],[38],[43] can be viewed as another type of leader/follower architecture. A pre-execution thread is constructed using either hardware or the compiler and leads the main thread to provide timely prefetches. In a multithreaded architecture, however, pre-execution threads and the main thread compete for a shared instruction window and a cache miss in any thread will block its execution and potentially affects other threads through resource competition. In future execution based on chip multiprocessors [15], an otherwise idle core pre-executes future loop iterations using value prediction to perform prefetching for the main thread.

Coupling two (or more) relatively simple processors to form a large instruction window for out-of-order

processing was originated in multiscalar processors [35] and DCE provides a complexity-effective way to construct such a window while eliminating elaborate inter-thread (or inter-task) register/ memory communication.

3. Dual-Core Execution

3.1. Detailed architectural design

In this section, we describe the design of dual-core execution based on MIPS R10000 [40] style superscalar microarchitectures, i.e., both the front and back processor is a MIPS R10000 processor. The basic pipeline of the MIPS R10000 style microarchitecture is shown in Figure 2. For memory operations, the execution stage is replaced with an address generation (AGEN) stage and two stages of memory access (MEM1 and MEM2).

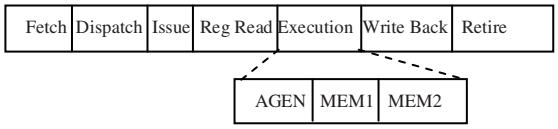

Figure 2. A MIPS R10000 style pipeline.

The proposed design of DCE is shown in Figure 3. It contains a few key components and next we discuss how they operate and what hardware changes are necessary to support the intended operations.

Front Superscalar Core

The front processor is modified so that long latency operations do not block its instruction window. Similar to run-ahead execution [25], an invalid (INV) bit is added to each physical register. When an INV bit is set, it indicates that the corresponding register value is invalid. For long latency operations such as cache-missing loads, an invalid value is used to substitute the data that are being fetched from memory by setting the INV bit of the destination register(s). Unlike run-ahead execution, which sets the INV bit when the cache-missing load reaches the head of the instruction window, the front processor sets the INV bit immediately after a load is detected to be a long latency miss so that its dependent instructions are awakened promptly. Executing instructions with an INV source register will propagate the INV bit except for branches and stores. If a branch instruction uses an INV register, its prediction will be used as the resolved branch target and the corresponding rename table checkpoint is reclaimed as if the prediction is correct. A store instruction becomes a nop if its address is invalid. If the value of a store instruction is invalid, the corresponding entry of the load/store queue (LSQ) will be marked as invalid (i.e., there is an INV bit for each LSQ entry) and the INV bit can be propagated via store-load forwarding using LSQ. Such INV forwarding is the *only* change to the LSQ in the front processor. In addition to the LSQ, an INV bit can also be forwarded using a run-ahead cache, as will be discussed later.

Instructions retire in-order as usual in the front processor (i.e., its architectural register map table is updated and the physical register in the previous map is reclaimed) except store instructions and instructions raising exceptions. When a store instruction retires, it either behaves like a nop (if there is no run-ahead cache) or updates the run-ahead cache (if it exists) but it will *not* write to the data caches or memory (only the back processor writes to the data caches). The retired instructions, at the same time, are sent to the result queue. The exception handling at the front processor is disabled since the precise state is maintained by the back processor.

As the front processor does not actually commit store instructions to update the memory, the data is lost once a

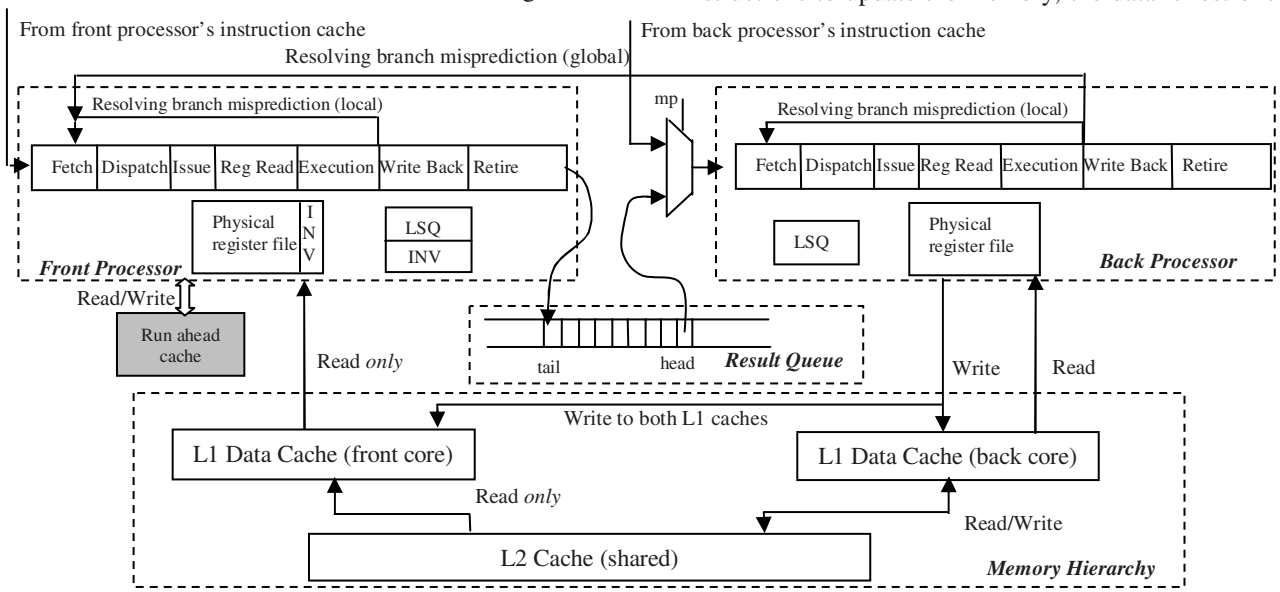

Figure 3. The design of DCE architecture.

store instruction retires since the corresponding LSQ entry is also de-allocated. As a result, subsequent load instructions in the front processor could fetch stale data from the data caches or memory. To solve such a problem, a small cache can be used as proposed for run-ahead execution [25]. Here, we use the same term 'run-ahead cache' and will investigate its performance impact in our experiments (see Section 5). The run-ahead cache is organized similar to that described in [25]. It holds both data and INV bits. When a store instruction with a valid address retires, it updates the run-ahead cache and sets the INV bit(s) (one for each byte) if the store value is invalid. If the size of the store data (e.g., a byte) is less than the block size of the run-ahead cache, the rest of the block is fetched from the data cache. When a block is replaced from the run-ahead cache, it is simply dropped and is *never* written to the data caches. When a store with an INV address retires, it acts as a nop. When a load executes, the LSQ, run-ahead cache, and L1 data cache (D-cache) are queried. The priority order is the LSQ, run-ahead cache, and L1 D-cache based on the assumption that the data in the run-ahead cache are more up-to-date than the L1 D-cache since the D-caches are only updated by the back processor.

Result Queue

The result queue is a first-in first-out structure, which keeps the retired instruction stream (both binary instructions and their PCs) from the front processor, thereby providing a continuous and highly accurate instruction stream to the back processor, substituting its I-cache. In this paper, we choose to keep instructions in their original format in the queue. Although keeping the decoded/renamed version can bypass some of the front-end processing of the back processor, the purpose of this paper is to introduce DCE and to evaluate its performance in such a design that incurs minimum hardware changes.

The instructions in the result queue are *not* associated with any centralized resource, unlike in-flight instructions in a conventional superscalar design, which reserve their allocated resources such as physical registers, load/store queue entry, rename table checkpoints, etc. Therefore, the result queue provides a much more scalable, complexity-effective way to achieve a very large instruction window.

Back Superscalar Core

A multiplexer (MUX) is added in front of the fetch unit of the back processor and its control signal (*mp*) directs whether instructions to be fetched from the result queue (single-thread mode) or from the back processor's instruction cache (multithread mode). In this way, DCE has the flexibility to serve both single-thread and multithreaded workloads.

In the single-thread mode, since the result queue provides the retired instruction stream from the front processor, the branch predictor at the back processor is not used and the branch targets computed at the front processor simply become the corresponding predictions. Once the instructions are fetched, the back processor processes them in its normal way except for mispredicted branches, as shown in Figure 3. When a branch misprediction is detected at the end of the execution stage, all the instructions in the back processor are squashed and the fetch unit is halted. At the same time, the front processor is notified to squash all its instructions and the result queue is emptied as well. As the front processor has lost the register states to recover from, the back processor's architectural state is used to synchronize the front processor. To do so, the program counter (PC) of the back processor is copied to the front processor, the back processor's architectural register values are copied over to the front processor's physical register file, the renaming table of the front processor is reset, and the run-ahead cache in the front processor is invalidated (i.e., set to be empty). Note that there is *no* need to synchronize memory states at the front processor as only the back processor writes to D-caches and all the front processor needs to do is to invalidate its run-ahead cache.

As instructions retire in-order and un-speculatively in the back processor, it provides the precise state for exception handling. Store instructions update data caches at the retire stage and there is *no* change to the LSQ.

Memory Hierarchy

In Figure 3, the back processor and the front processor use separate L1 data caches and a shared unified L2 cache. The L1 D-cache misses at one processor are used as prefetch requests for the L1 D-cache in the other processor. The stores in the back processor update *both* L1 D-caches at the retirement stage. The dirty blocks replaced from the front processor's L1 D-cache are simply dropped (in the single thread mode). The separate L1 D-caches (and the run-ahead cache) need *not* to be coherent (i.e., no added complexity to maintain such coherence) as there is *no* correctness requirement for the front processor.

Collaboration among the Front Processor, the Result Queue, and the Back Processor

In DCE, a program is fetched and preprocessed aggressively by the front processor: the long latency operations (i.e., cache misses) are initiated but not completed; and independent branch mispredictions are resolved. The result queue buffers the in-order, accurate instruction stream retired from the front processor. The back processor fetches instructions from the result queue and executes them in its normal way. Those long latency cache misses initiated long ago at the front processor become prefetches for the back processor. When the back processor is stalled due to another long latency operation, the front processor continues its preprocessing until the result queue is eventually full. At this time, long latency operations in the front processor will operate in their normal way instead of producing an invalid value. When the back processor detects that the front processor deviates

from the right path due to a branch misprediction /misresolution, the front processor is rewound to the same execution state as the back processor.

The policy of determining whether a long latency operation should use an invalid value affects the aggressiveness of the front processor. The default policy used in our design has two criteria: an L2 cache miss plus the result queue not being full. In the multithreaded mode, such invalidation is disabled at the front processor and stores update the D-caches normally when they retire. The back processor fetches instructions from its I-cache instead of the result queue, writes only to its own caches, and stops interfering with the front processor via branch misprediction handling.

Transition between Single- and Multi-thread Modes

DCE forms a very large instruction window to hide memory access latencies. For computation-intensive workloads, however, DCE is less efficient as not many cache misses can be invalidated at the front processor and the multi-thread mode should be used. Fortunately, a simple L2 miss counter can easily determine the memory intensiveness of a workload and set the single-/multi-thread mode accordingly. Other techniques proposed in [23] can also help to determine whether the single- or multi-thread mode should be used.

The transition from the single- to multi-thread mode is similar to branch misprediction recovery: the architectural state at the back processor is copied to the front processor, the invalidation is disabled at the front processor, and the back processor starts executing a new thread by fetching the instructions from its own I-cache. To transit from the multi- to single-thread mode, the architectural state at the front processor is copied to the back processor, the invalidation is enabled at the front processor, and the back processor starts fetching from the result queue.

Using the Result Queue as a Value Predictor

As discussed in Section 2, the front processor in DCE is a faster leader due to its virtually ideal L2 data cache. To speed up the back processor, the result queue can be used to carry the execution results from the front processor and provide them as value predictions [21] to the back processor. At the back processor, those value predictions are verified at the execution stage and mispredictions initiate the same recovery process as branch mispredictions. As will be seen in Section 5.3, such execution-based value prediction achieves nearly oracle prediction accuracy and introduces some performance improvements but is *not* an essential part of the DCE design.

3.2. Comparison to a single processor with very large centralized instruction windows

In DCE, all components including the front processor, result queue, and back processor keep some in-flight instructions. As discussed in Section 3.1, the result queue is latency tolerant and easily extended to a large size. In this way, DCE forms a very large instruction window using two relatively simple superscalar cores and a simple queue structure.

Compared to a single superscalar processor with a very large centralized instruction window, DCE has higher scalability, much less complexity and potentially higher clock speed (or the same clock speed with shallower pipelines). Also, its non-uniform branch resolution fits naturally with branches depending on variable latency operations: the mispredictions depending on short-latency operations are resolved more promptly at the front processor (due to its simpler, shallower pipeline) while only mispredictions dependent on long-latency cache-misses are fixed at the back processor. In addition, since the front and back processors reclaim the rename map table checkpoints in their usual way when branches are resolved, there is no increased pressure on those checkpoints, as a large centralized instruction window would normally induce [1]. Therefore, the number of outstanding branches is no longer a limit for the instruction window.

On the other hand, a single superscalar processor with a very large centralized instruction window has an advantage in ILP processing since any instruction in the window can be issued and executed once its source operands become available (although storing those speculative results is a source of the complexity, e.g., large LSQs). In DCE, instructions are only processed when they are in the back processor as the execution results in the front processor are dropped when they retire. In other words, in the instruction window formed with DCE, only the instructions at the 'head' portion can be issued and executed, thereby limiting ILP exploitation.

4. Simulation Methodology

Our simulation environment is based upon the SimpleScalar [6] toolset but our execution-driven timing simulator is completely rebuilt to model the MIPS R10000 pipeline shown in Figure 2. The functional correctness of our simulator is ensured by asserting that the source and destination values of each retired instruction match with those from the functional simulator and wrong-path events are also faithfully simulated. The cache module (including the run-ahead cache) in our simulator models both data and tag stores. The front and back processors have the same configurations (but a shared L2 cache), shown in Table 1. The correctness assertions are disabled in the front processor model but enforced in the back processor model. The default result queue has 1024 entries (the performance impact of the queue size is examined in Section 5.5) and the default run-ahead cache is configured as 4kB, 4-way associative with a block size of 8 bytes. A latency of *16* cycles is assumed for copying the architectural register values from the back processor to the

front processor (updating 4 registers per cycle for 64 architectural registers) when branch mispredictions are resolved at the back processor (a more pessimistic case with a *64*-cycle latency is also modeled in Section 5.1). Each processor has a stride-based stream buffer hardware prefetcher [14],[33], which has 8 4-entry stream buffers with a PC-based 2-way 512-entry stride prediction table. To ensure that the performance gains are from latency hiding rather than from conservative memory disambiguation, oracle disambiguation is modeled, meaning that loads will only stall when there is a prior store with the same address (i.e., perfect memory dependence prediction). In Section 5.6, we examine the impact of different memory disambiguation schemes and highlight an interesting feature of DCE to compensate conservative disambiguation schemes. The default DCE does *not* use the result queue to provide value predictions to the back processor. The performance impact of such value predictions is addressed in Section 5.3.

As DCE is proposed to mainly tolerate long memory-access latencies, we focus on memory-intensive benchmarks from the SPEC2000 benchmark suite [17] and our selection criterion is that an ideal L2 cache introduces at least 40% speedup. In addition, two computation-intensive benchmarks, *gap* and *bzip2*, are also included to illustrate interesting aspects of DCE and other competitive approaches. The reference inputs are used and single simulation points are chosen by running the Simpoint toolset [32] with our Simplescalar binaries. For those benchmarks with prohibitively long fast forward phases, we chose to skip the first 700M instructions and simulate the next 300M instructions.

The execution time of DCE is measured as the time between when the front processor starts fetching the first instruction and the back processor retires the last instruction. In our experiments, we also modeled run-ahead execution [13],[25] and slipstream processors [27],[37] to compare with DCE. Run-ahead execution is implemented according to [25] but with the processor model described in Table 1 and a 4 kB run-ahead cache. Oracle memory disambiguation is also modeled for both run-ahead execution and slipstreaming processors. For fair comparison with slipstream processors, we use the same memory hierarchy as in DCE to reflect the recent development of hardware-based memory duplication in slipstream processing [28]. The stores in A-stream are committed into the run-ahead cache rather than its L1 D-cache, simplifying the IR-misprediction recovery controller. Other slipstream parameters are based on those used in [27], including a 2^{20}-entry g-share indexed IR-predictor, a 256-instruction R-DFG, a 1024-entry delay buffer, and a 16-cycle IR-misprediction recovery latency. The fetch bypass is *not* implemented, i.e., the ineffectual instructions will still be fetched but will bypass the execution pipeline as presented in [37]. In this way, both slipstream and DCE have similar execution behavior: the front processor (or A-stream) fetches the complete program but only executes a subset of the program, while the back processor (or R-stream) re-executes the whole program. Note that slipstream and DCE have different ways to execute a subset of the program: the ineffectual instructions will bypass the processing pipeline completely in the slipstream paradigm while in DCE long-latency operations and their dependents still go through the pipeline carrying invalid values.

Table 1. Configuration of the front and back processors.

Pipeline	3-cycle fetch stage, 3-cycle dispatch stage, 1-cycle issue stage, 1-cycle register access stage, 1-cycle retire stage. Minimum branch misprediction penalty = 9 cycles
Instruction Cache	Size=32 kB; Assoc.=2-way; Replacement = LRU; Line size=16 instructions; Miss penalty=*10* cycles.
Data Cache	Size=32 kB; Assoc.=2-way; Replacement=LRU; Line size = 64 bytes; Miss penalty=*10* cycles.
Unified L2 Cache	Size=1024kB; Assoc.=8-way; Replacement = LRU; Line size=128 bytes; Miss penalty=*220* cycles.
Branch Predictor	64k-entry G-share; 32k-entry BTB
Superscalar Core	Reorder buffer: 128 entries; Dispatch/issue/retire bandwidth: 4-way superscalar; 4 fully-symmetric function units; Data cache ports: 4. Issue queue: 64 entries. LSQ: 64 entries. Rename map table checkpoints: 32
Execution Latencies	Address generation: *1* cycle; Memory access: *2* cycles (hit in data cache); Integer ALU ops = 1 cycle; Complex ops = MIPS R10000 latencies
Memory Disambiguation	Perfect memory disambiguation
Hardware prefetcher	Stride-based stream buffer prefetch

5. Experimental Results

5.1. Latency hiding using DCE

In this section, we evaluate the latency hiding effects of DCE and compare it with run-ahead execution. Figure 4 shows the normalized execution time of a single baseline processor (labeled 'base'), DCE with and without a run-ahead cache (labeled 'DCE' and 'DCE wo rc'), DCE with a 64-cycle latency for copying the architectural registers from the back to front processor (labeled 'DCE_64'), and a single baseline processor with run-ahead execution (labeled 'RA'). Each cycle is categorized as a pipeline stall with a full reorder buffer (ROB) due to cache-misses, a stall with a full ROB due to other factors such as long latency floating-point operations, a stall with an empty ROB, a cycle in un-stalled execution, or an execution cycle in the run-ahead mode (labeled 'RA mode'). In DCE, such cycle time distribution is collected from the back processor.

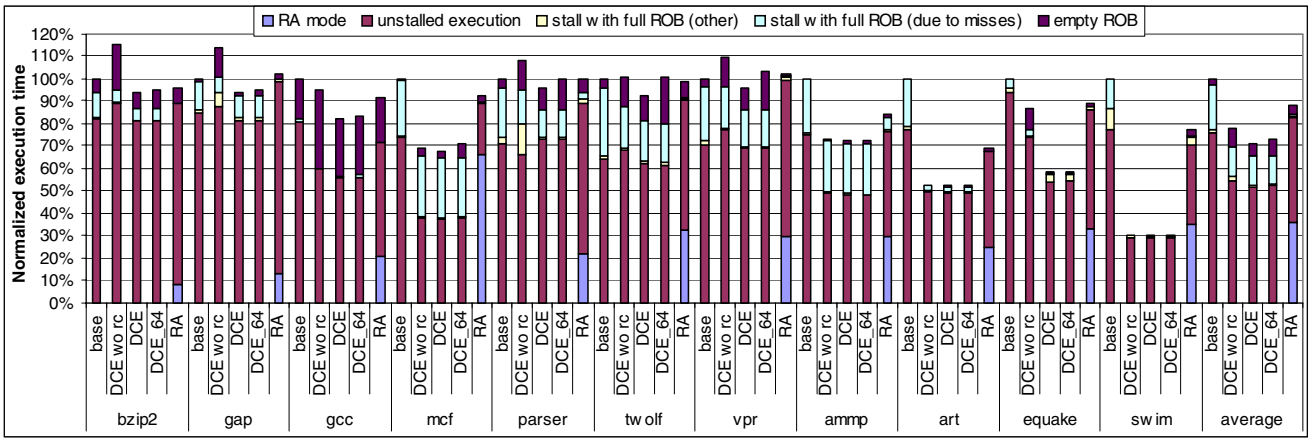

Figure 4. Normalized execution time of a baseline processor (base), DCE, DCE without run-ahead cache (DCE wo rc), DCE with 64-cycle register copy latency (DCE_64), and a run-ahead execution processor (RA).

Several important observations can be made from Figure 4. First, for computation-intensive benchmarks, both DCE and run-ahead execution have limited performance improvement. Run-ahead execution achieves a 4.2% speedup for the benchmark *bzip2* but incurs a 2.2% performance loss for the benchmark *gap*. In DCE, the run-ahead cache is crucial to avoid the negative effects and it achieves speedups of 6.7% and 6.3% for those two benchmarks respectively. Since computation-intensive benchmarks have few L2 misses, the front processor fails to run much ahead as it relies on invalidating cache-missing loads to make faster progress. Moreover, without a run-ahead cache, the front processor could load stale values after earlier store instructions are dropped. This affects performance when such stale values are used to determine branch outcomes (see Section 5.2) or to compute new load addresses. Here, we note that both DCE and run-ahead execution are *not* designed for computation-intensive workloads and a simple L2 miss counter is able to tell whether the dual cores should be used in the single-thread or multithread mode (or to allow run-ahead execution).

Secondly, for the workloads with higher memory demands, DCE significantly reduces the pipeline stalls due to cache misses, resulting in remarkable speedups, up to 232% (*swim*) and 41.3% on average (28.7% without the run-ahead cache). Such pipeline-stall reduction also leads to the reduction of both un-stalled execution time (e.g., *art*) and the stall cycles due to other factors (e.g., *swim*). The reason is that turning a cache miss into a hit not only reduces the chances to stall the pipeline but also enables more computation to be overlapped.

Thirdly, we confirm that run-ahead execution is also effective in hiding memory latencies for memory-intensive benchmarks. With run-ahead execution, the stalls due to cache misses are negligible since the processor enters the 'run-ahead' mode and continues its un-blocked execution. Compared to run-ahead execution, DCE achieves significantly higher speedups on average (41.3% vs. 13.2%). The key reason is that the front processor continues running ahead when the cache misses are repaired while the run-ahead execution processor has to stop the pre-execution and return to the normal mode. Moreover, DCE eliminates the mode transition penalties completely as discussed in Section 2.1.

Fourthly, DCE is tolerant on the increased communication latency between the front and back processors. When the latency for copying the architectural register file increasing from 16 to 64 cycles, stall cycles with an empty ROB are increased due to higher branch misprediction penalties at the back processor (e.g., *twolf* and *vpr*). However, as the front processor effectively resolves branch mispredictions for most benchmarks, the performance impact is limited and the average performance improvement is 36.2% over the baseline processor.

5.2. Non-uniform branch handling in DCE

In this section, we examine the impact of the non-uniform branch resolution in DCE. Figure 5 shows the branch misprediction rates of a single baseline processor, the front and back processor in DCE with and without a run-ahead cache.

With the large instruction window formed with DCE, the programs run along speculative paths more aggressively, resulting in more branch mispredictions than the single baseline processor. Among those branch mispredictions, most (92% on average with the run-ahead cache and 88% without) are resolved promptly at the front processor, implying that most mispredictions are independent on cache-missing loads. The mispredictions that indeed depend on cache misses are resolved at the back processor, incurring additional penalties since the architectural state needs to be copied from the back to the front processor. Fortunately, the number of such mispredictions is very small as shown in Figure 5, 0.65 (1.32 if without the run-ahead cache) mispredictions per

1000 retired instructions on average, and their performance impact is therefore limited. The run-ahead cache at the front processor helps to resolve branch predictions more accurately at the front processor and is the main reason why DCE has much fewer empty-ROB stall-cycles compared to DCE without the run-ahead cache (e.g., *gap* and *bzip2*), as shown in Figure 4.

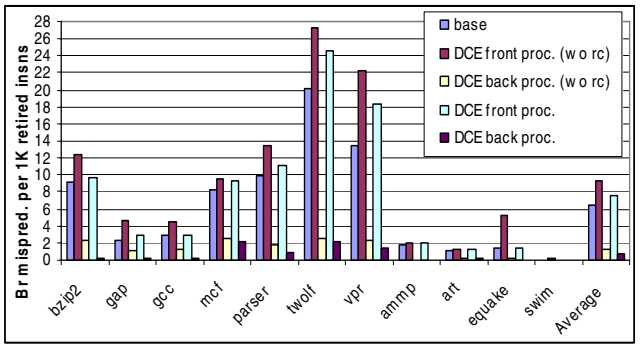

Figure 5. Branch mispredictions detected in a single baseline processor, the front and back processors in DCE with and without a run-ahead cache. (some numbers are too small to be discernible, e.g., the misprediction rate at DCE back proc. for *ammp*)

5.3. DCE vs. slipstream processing

As discussed in Section 2.2, DCE and slipstream processors share similar leader/follower architectures although they achieve the performance gains in quite different ways. Figure 6 shows the execution time of DCE and slipstream processors normalized to the execution time of the single baseline processor. The slipstreaming results are labeled 'SS' and the fractions of instruction removal are included for each benchmark. Due to the leader/follower characteristics, execution time is broken down based on the utilization of the result queue (or the delay buffer in slipstreaming processing) rather than pipeline utilization. If the result queue is full (labeled 'full'), it means that the leader retires instructions at a faster rate than the follower can consume. If the result queue is empty (labeled 'empty'), it shows that the follower runs faster than the leader, leaving the leader to be the bottleneck. If the result queue is neither full nor empty (labeled 'other'), the leader and follower run at similar speeds. In Figure 6, we also include the execution results of DCE with value prediction support, in which the result queue is used to carry the execution results from the front processor as value predictions to the back processor. Such value prediction is used in slipstream processors but not in the default DCE.

From Figure 6, we see that there is a significant amount of ineffectual dynamic instructions in each benchmark, confirming the insight from [37]. Removing them, however, does not necessarily lead to a much faster A-stream processing rate. As shown in Figure 6, the delay buffer is empty for a large portion of the execution time. The main reason is that given the ever increasing memory access latency, pipeline stalls due to cache misses dominate the execution time. Unless those cache-missing loads are removed, the A-stream can not run much faster. Taking the benchmark *mcf* as an example, over 32% of its dynamic instructions (including some cache-missing loads) are removed from the A-stream. But there are still many cache misses blocking the A-stream as we failed to detect them as ineffectual. One common case is that cache-missing loads lead to a store value, which will be referenced or overwritten after thousands of instructions. Since detecting whether such a store is ineffectual is beyond the capabilities of a 256-instruction R-DFG, those loads cannot be removed.

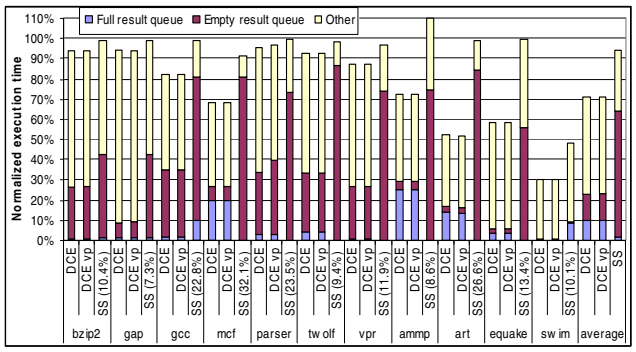

Figure 6. Normalized execution time of DCE, DCE with value prediction (DCE vp), and slipstreaming processors (SS).

In DCE, on the other hand, the front processor is a much faster leader and the result queue stays full more often for those benchmarks with higher L2 misses (e.g., *mcf* and *art*). For the benchmarks with relatively fewer L2 misses, both the L1 misses (but L2 hits) at the front processor (e.g., *gcc*) and branch mispredictions detected at the back processor (e.g., *twolf*) contribute to the execution time with an empty result queue. Compared to slipstream processors, DCE achieves much higher performance improvement with less hardware complexity (i.e., no need for IR-detectors or IR-predictors).

Another interesting observation from this experiment is that the value predictions based on the execution results from the leader processor achieve near oracle prediction accuracy (over 99.9%) in both slipstream processors and DCE with value prediction ('DCE vp'). The misprediction penalties therefore are not the performance bottleneck and even higher recovery latencies can be tolerated. The performance benefit of such value prediction in DCE, however, is quite limited since the bottleneck of the back processor is those cache-misses that were not prefetched in-time by the front processor and their dependent instructions. The front processor can not provide predictions for those instructions as they were turned invalid. A similar observation is made in [24] to explain

why reusing the results during run-ahead execution has limited benefits.

5.4. DCE vs. single processors with very large centralized instruction windows

In this experiment, we examine a single superscalar processor with different instruction window sizes, 256 and 512, and we also scale the issue queue and LSQ sizes accordingly (half of the ROB size). The issue width remains at 4 and the execution time shown in Figure 7 is normalized to the baseline 4/128 processor. From Figure 7, it can be seen that DCE using two 4/128 processors (labeled 'DCE 4/128') achieves significant speedups (up to 141% and 13% on average) over a single 4/256 processor (labeled 'base 4/256'). For the computation-intensive benchmarks *bzip2* and *gap*, limited performance improvement is observed from both DCE and large centralized window processors since cache misses are not their performance bottleneck. For the benchmarks *parser*, *twolf*, and *vpr*, DCE improves their performance but not as much as a single processor with a double-sized window. The main reason is that a superscalar with a very large instruction window not only provides latency hiding but also better ILP, as discussed in Section 3.2.

Another interesting observation from this experiment is that DCE can also benefit the superscalar processors with large windows to achieve even better results. As shown in Figure 7, DCE using two 4/256 processors (labeled 'DCE 4/256') performs much better than a single 4/256 processor (up to 148% speedup and 23% on average) and for the benchmarks including *bzip2*, *gap*, *gcc, ammp*, *equake*, and *swim*, it performs better than a single 4/512 processor (labeled 'base 4/512').

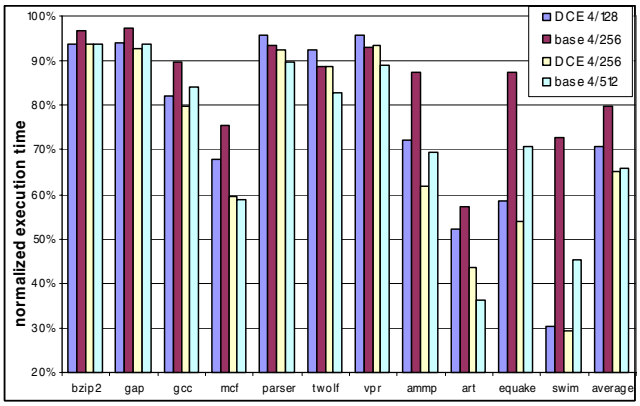

Figure 7. Performance comparison between DCE and single processors with large instruction windows.

5.5. Run ahead, how far should it go?

In DCE, the result queue size determines how far ahead the front processor can run when the back processor is stalled. As discussed in Section 3.1, the result queue is latency tolerant and can be easily scaled to keep a higher number of instructions to allow the front processor running further ahead. In this experiment, we examine the performance impact of different run-ahead distances. Both the front and back processors use a baseline 4/128 processor and the execution time shown in Figure 8 is normalized to a single baseline 4/128 processor.

From Figure 8, it can be seen that the best run-ahead distance of the front processor is benchmark dependent while longer queues result in higher speedups on average (from 32% with a 256-entry queue to 43% with a 4096-entry queue) since they enable the front processor to pre-execute more aggressively. For most integer benchmarks, including *bzip2*, *gap*, *mcf*, *parser*, *twolf*, and *vpr*, a result queue of 256 or 512 entries reaps most of the performance improvement. The benchmarks *gcc*, *ammp*, *art*, *equake*, and *swim*, on the other hand, exhibit stronger scalability with longer run-ahead distances. One main reason why a longer run-ahead distance does not help is cache pollution resulting from incorrect or untimely prefetches from the front processor. Another benchmark-dependent factor is the number of independent cache misses in the scope of a run-ahead distance.

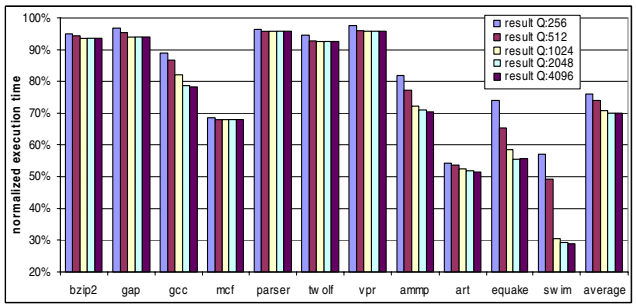

Figure 8. Normalized execution time of DCE with different result queue sizes.

5.6. Impact of memory disambiguation on DCE

In the previous experiments, we model oracle memory disambiguation to isolate the impact of latency hiding. To analyze the performance impact of different memory disambiguation schemes on DCE, we model pessimistic disambiguation, which delays the issue of load instructions until all prior store addresses are available, in this experiment. Pessimistic and oracle disambiguation represent two extremes in the spectrum of disambiguation schemes (i.e., no prediction and perfect prediction of memory dependence). An aggressive memory dependence prediction scheme, e.g., store-set based prediction [9], combined with selective reissuing is expected to achieve the performance close to oracle prediction since DCE does not incur additional penalties for memory order violations. For the violations detected at the front processor, the back processor is not affected since it processes the retired instruction stream from the front processor. For the violations detected at the back processor, it selectively re-executes the affected instructions and does not need to

disturb the result queue or the front processor. In the next experiment, however, we will show that aggressive memory dependence prediction is not essential for DCE. Furthermore, DCE can also compensate conservative disambiguation schemes to achieve similar performance to DCE with more aggressive ones.

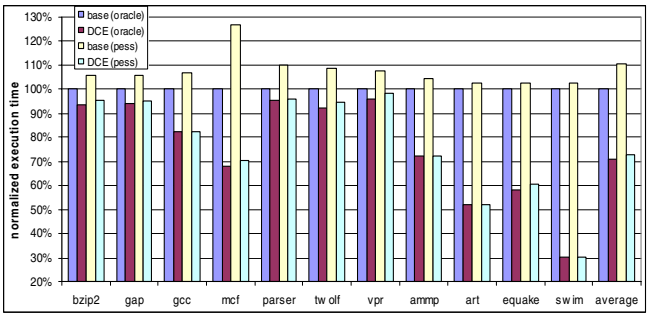

Figure 9. Normalized execution time of a baseline processor and DCE with different memory disambiguation schemes.

Figure 9 shows the execution time of a baseline processor and DCE with both pessimistic and oracle memory disambiguation. The default configuration shown in Table 1 is used for the baseline processor, the front processor, and the back processor in DCE. Two important observations can be made from Figure 9. First, better memory disambiguation benefits both the baseline processor and DCE. The benchmark, *bzip2*, for example, observes a 5.6% speedup for the baseline processor and a 2.0% speedup for DCE when pessimistic disambiguation is replaced with oracle disambiguation. Secondly, DCE achieves much higher speedups over the baseline processor with pessimistic disambiguation than with oracle disambiguation, showing that DCE is capable of compensating conservative memory disambiguation schemes. The reason is that with pessimistic disambiguation, the store instructions, whose addresses are dependent on a cache-missing load, will block the issue of later loads in the baseline processor. In DCE with the same pessimistic disambiguation, the same stores will *not* block the later loads in the front processor since the misses are converted into 'hits' with invalid values and the stores become nops with invalid addresses. In the back processor, the misses become hits due to the prefetches initiated at the front processor. So, the addresses of those stores are computed faster, allowing subsequent loads to be issued more promptly in the back processor.

6. Conclusions

In this paper, we propose a novel way to utilize multi-cores on a single chip to form a very large instruction window for single-thread applications. The proposed execution paradigm, DCE, is built upon two small CMP cores, a front and a back processor, coupled with a result queue. The front processor acts as a fast preprocessor of instruction streams. It fetches and executes instructions in its normal way except for cache-missing loads, which produce an invalid value instead of blocking the pipeline. Since it is not stalled by cache misses, the front processor runs far ahead to warm up data caches and fix branch mispredictions for the back processor. The result queue buffers retired instructions from the front processor and feeds them into the back processor. With the assists from the front processor, the back processor also makes faster progress and provides the precise program state. The proposed design incurs only minor hardware changes and achieves remarkable latency hiding for single-thread memory-intensive workloads and maintains the flexibility to support multithreaded applications. With a queue of 1024 entries, DCE outperforms a single-thread core by 41% (up to 232%) on average and achieves better performance than run-ahead execution on every benchmark we studied (24% on average).

In DCE, re-execution is used to eliminate hardware complexities needed for very large centralized instruction windows. Such re-execution, however, incurs extra power consumption. Currently, we are investigating two promising ways to improve the power efficiency of DCE by eliminating redundant re-execution. At the front processor, instructions that do not lead to cache misses or branch mispredictions can be bypassed directly to the result queue. The back processor, on the other hand, can reuse the front execution results more intelligently. The key is that for a sequence of instructions forming a data dependence chain, the back processor only needs to re-execute the instructions producing the live-in values rather than every instruction in the chain.

7. Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments and Mark Heinrich for his help in improving the paper.

8. References

[1] H. Akkary, R. Rajwar, and S. Srinivasan, "Checkpoint processing and recovery: towards scalable large instruction window processors", *Proc. of the 36th Int. Symp. on Microarch. (MICRO-36)*, 2003.

[2] R. Balasubramonian, S. Dwarkadas, and D. Albonesi, "Dynamically allocating processor resources between nearby and distant ILP", *Proc. of the 28th Int. Symp. on Comp. Arch. (ISCA-28)*, 2001.

[3] R. Balasubramonian, S. Dwarkadas, and D. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors", *Proc. of the 34th Int. Symp. on Microarch. (MICRO-34)*, 2001.

[4] R. Barnes, E. Nystrom, J. Sias, S. Patel, N. Navarro, and W. Hwu, "Beating in-order stalls with flea-flicker two pass

pipelining", *Proc. of the 36th Int. Symp. on Microarch. (MICRO-36)*, 2003.
[5] E. Brekelbaum, J. Rupley II, C. Wilkerson, and B. Black, "Hierarchical scheduling windows". *MICRO-35*, 2002.
[6] D. Burger and T. Austin, "The SimpleScalar tool set, v2.0", *Computer Architecture News*, vol. 25, June 1997.
[7] L. Ceze, K. Strauss, J. Tuck, J. Renau, J. Torrellas. "CAVA: Hiding L2 Misses with Checkpoint-Assisted Value Prediction." Comp. Arch. Letters, Volume 3, Dec. 2004.
[8] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism", *Proc. of the 31st Int. Symp. on Comp. Arch. (ISCA-31)*, 2004.
[9] G. Chrysos and J. Emer, "Memory dependence prediction using store sets", *Proc. of the 25th Int. Symp. on Comp. Arch. (ISCA-25)*, 1998.
[10] J. D. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: long-range prefetching of delinquent loads", *ISCA-28*, 2001.
[11] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-order commit processors", *Proc. of the 10th Int. Symp. on High Performance Comp. Arch. (HPCA-10)*, 2004.
[12] A. Cristal, M. Valero, A. Gonzalez, and J. Llosa, "Large virtual ROBs by processor checkpointing", *Tech. Rep. UPC-DAC-2002-39*, 2002.
[13] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss", *ICS-97*, 1997.
[14] K. Farkas, P. Chow, N . Jouppi, and Z. Vranesic, "Memory-system design considerations for dynamically scheduled processors", *Proc. of the 24th Int. Symp. on Comp. Arch. (ISCA-24)*, 1997.
[15] I. Ganusov and M. Burtscher, "Future execution: a hardware prefetching technique for chip multiprocessors", *Int'l. Conf. on Parallel Arch, and Comp. Tech.* (PACT 2005), 2005.
[16] A. Gandhi, H. Akkary, R. Rajwar, S. Srinivasan, and K. Lai, "Scalable load and store processing in latency tolerant processors", *ISCA-32*, 2005.
[17] J. Henning, "SPEC2000: measuring CPU performance in the new millennium", *IEEE Computer*, July 2000.
[18] T. Karkhanis and J. Smith, "A Day in the Life of a Cache Miss", *2nd Workshop on Memory Performance Issues*, 2002.
[19] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez, "Checkpointed Early Load Retirement", *Proc. of the 11th Int. Symp. on High Perf. Comp. Arch. (HPCA-11)*, 2005.
[20] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A large, fast instruction window for tolerating cache misses", *Proc. of the 29th Int. Symp. on Comp. Arch. (ISCA-29)*, 2002.
[21] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," *Proc. of the 29th Int. Symp. on Microarch. (MICRO-29)*, 1996.
[22] C. K. Luk, "Tolerating memory latency through soft-ware-controlled pre-execution in simultaneous multithreading processors", *Proc. of the 28th Int. Symp. on Comp. Arch. (ISCA-28)*, 2001.
[23] O. Mutlu, H. Kim, and Y. Patt, "Techniques for efficient processing in runahead execution engines", *Proc. of the 32nd Int. Symp. on Comp. Arch. (ISCA-32)*, 2005.
[24] O. Mutlu, H. Kim, J. Stark, and Y. Patt, "On reusing the results of pre-executed instructions in a runahead execution processor", *Comp. Arch. Letters*, Jan 2005.
[25] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors", *Proc. of the 9th Int. Symp. on High Perf. Comp. Arch. (HPCA-9)*, 2003.
[26] I. Park, C. Ooi, and T. Vijaykumar, "Reducing design complexity of the load/store queue", *MICRO-36*, 2003.
[27] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. "A Study of Slipstream Processors". *MICRO-33*, 2000.
[28] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "Slipstream memory hierarchies", *Tech. Report, ECE dept., NCSU*, 2002.
[29] E. Rotenberg, Personal Communication, 2003.
[30] A. Roth and G. Sohi, "Speculative data driven multithreading", *HPCA-7*, 2001.
[31] S. Sethumadhavan, R. Desikan, D. Burger, C. Moore, and S. Keckler, "Scalable hardware memory disambiguation for high ILP processors", *Proc. of the 36th Int. Symp. on Microarch. (MICRO-36)*, 2003.
[32] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior", ASPLOS-X, 2002.
[33] T. Sherwood, S. Sair, and B. Calder, "Predictor-directed stream buffers", *MICRO-33*, 2000.
[34] J. E. Smith, "Decoupled access/execute computer architectures", *Proc. of the 9th Int. Symp. on Comp. Arch. (ISCA-9)*, 1982.
[35] G. Sohi, S. E. Breach, T. N. Vijaykumar, "Multiscalar processors", *Proc. of the 22nd Int. Symp. on Comp.Arch.(ISCA-22)*, 1995.
[36] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines", *ASPLOS-11*, 2004.
[37] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: improving both performance and fault tolerance", *ASPLOS-9*, 2000.
[38] P. H. Wang, H. Wang, J. D. Collins, E. Grochowski, R. M. Kling, and J. P. Shen, "Memory latency-tolerance approaches for Itanium processors: out-of-order execution vs. speculative precomputation", *HPCA-8*, 2002.
[39] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious", ACM SIGARCH Comp. Arch. News, 1995.
[40] K. C. Yeager, "The MIPS R10000 superscalar microprocessor", *IEEE Micro*, 1996.
[41] J. Zalamea, J Llosa, E. Ayguade, and M. Valero, "Two-level hierarchical register file organization for VLIW processors", *Proc. of the 33rd Int. Symp. on Microarch. (MICRO-33)*, 2000.
[42] H. Zhou and T. Conte, ""Enhancing memory level parallelism via recovery-free value prediction", Int. Conf. on Supercomputing (ICS 2003), June 2003.
[43] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices", *the 28th Int. Symp. on Comp. Arch. (ISCA-28)*, 2001.
[44] C. Zilles and G. Sohi, "Master/Slave Speculative Parallelization", *Proc. of the 35th Int. Symp. on Microarch. (MICRO-35)*, 2002.
[45] C. Zilles, "Master/Slave Speculative Parallelization and Approximate Code", PhD Thesis, Univ. of Wisconsin, 2002.