Optimizing Dual-Core Execution for Power Efficiency and Transient-Fault Recovery

Yi Ma, Hongliang Gao, Martin Dimitrov, and Huiyang Zhou, *Member*, *IEEE Computer Society*

Abstract—Dual-core execution (DCE) is an execution paradigm proposed to utilize chip multiprocessors to improve the performance of single-threaded applications. Previous research has shown that DCE provides a complexity-effective approach to building a highly scalable instruction window and achieves significant latency-hiding capabilities. In this paper, we propose to optimize DCE for power efficiency and/or transient-fault recovery. In DCE, a program is first processed (speculatively) in the front processor and then reexecuted by the back processor. Such reexecution is the key to eliminating the centralized structures that are normally associated with very large instruction windows. In this paper, we exploit the computational redundancy in DCE to improve its reliability and its power efficiency. The main contributions include: 1) DCE-based redundancy checking for transient-fault tolerance and a complexity-effective approach to achieving full redundancy coverage and 2) novel techniques to improve the power/energy efficiency of DCE-based execution paradigms. Our experimental results demonstrate that, with the proposed simple techniques, the optimized DCE can effectively achieve transient-fault tolerance or significant performance enhancement in a power/energy-efficient way. Compared to the original DCE, the optimized DCE has similar speedups (34 percent on average) over single-core processors while reducing the energy overhead from 93 percent to 31 percent.

Index Terms—Multiple data stream architectures, fault tolerance, low-power design.

1 Introduction

The advent of the billion-transistor processor era presents new challenges in computer system design. Among them, the key challenges include 1) how the increasing number of transistors can be effectively converted into performance enhancement, 2) how a high level of reliability can be sustained, given a processor's soft error rate being proportional to its integration scale [19], and 3) how the first two goals can be achieved in a highly power/energy-efficient manner. On the first front, the industry turns to chip multiprocessor (CMP) architectures for high system throughput. The performance of single-threaded applications, however, is not addressed adequately. Furthermore, as indicated from Amdahl's law, even in parallel tasks, their sequential parts will dominate the scalability and the overall performance. On the second front, most existing designs incur some performance overhead to meet their reliability requirements. A recently proposed paradigm, the dual-core execution (DCE) [40], utilizes CMPs collaboratively to improve the performance of single-threaded applications, and it has been shown that DCE achieves substantial performance gains by forming a highly scalable instruction window to tolerate long-latency cache misses. In this paper, we propose optimizing DCE to address the remaining two challenges, and our study shows that, with the proposed simple techniques, the optimized DCE can effectively achieve high performance and/or transient-fault tolerance in a highly power/energy-efficient manner.

In DCE, a program is first processed in a very fast yet highly accurate way in the front processor and then reexecuted by the back processor. Such reexecution in the back processor is the key to eliminating centralized structures normally associated with a very large instruction window. In this paper, we show that this inherent computation redundancy can be easily exploited to provide transient-fault tolerance. To detect and recover from transient faults in DCE, the *speculative* execution results from the front processor are simply carried over to the back processor and then compared with the *nonspeculative* execution results in the back core. Any discrepancy, either due to a wrongful speculation or an actual transient fault, will then be recovered transparently with the existing wrongful speculation recovery mechanism. This way, the processor reliability can be efficiently improved at very little hardware cost. Then, we propose a new complexity-effective extension to DCE to achieve full redundancy coverage, that is, redundancy checking for all retired instructions, and we show that such redundancy checking has only limited performance impact compared to the original DCE. In other words, both transient-fault tolerance and significant performance improvement can be achieved at the same time.

Although it is the key to performance enhancement, the very large instruction window formed with DCE incurs extra energy/power overhead. The reason is due to those branch mispredictions that are dependent on long-latency cache misses. Given the large instruction window, a high number of wrong-path instructions would be fetched and executed before such a misprediction is resolved. To overcome such overhead while retaining the benefits of

• *The authors are with the School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32816-2362. E-mail: {yma, hgao, dimitrov, zhou}@cs.ucf.edu.*

Manuscript received 22 Aug. 2006; revised 5 Jan. 2007; accepted 23 Feb. 2007; published online 9 Mar. 2007.
Recommended for acceptance by R. Iyer and D.M. Tullsen.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDSSI-0242-0806.
Digital Object Identifier no. 10.1109/TPDS.2007.1080.

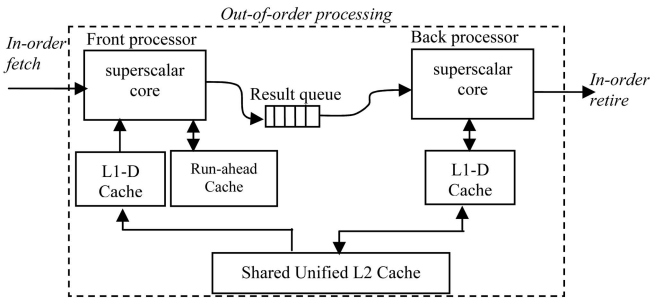


Fig. 1. DCE.

large instruction windows, we propose to adaptively adjust the window size and selectively invalidate cache-missing loads based on workload characteristics. With the proposed simple optimizations, full redundancy coverage and high performance can be achieved in a highly energy/power-efficient manner. For systems without a strong reliability requirement, more aggressive approaches can be incorporated to further reduce the energy/power consumption of DCE. First, the redundant execution in DCE can be significantly reduced while still ensuring execution correctness. Second, to avoid the energy overhead in those workloads/execution phases where DCE is not highly effective, a dynamic mechanism is devised to enable DCE only when it is beneficial. Our experimental results show that those techniques effectively reduce the energy overhead at very little performance cost, and the optimized DCE achieves even higher energy/power efficiency than a single-core processor.

The remainder of the paper is organized as follows: Section 2 presents the background on DCE and other related work. Section 3 explains the experimental methodology used in this paper. Exploiting redundant execution in DCE to achieve transient-fault tolerance is detailed in Section 4. The optimizations to improve the energy/power efficiency of DCE with and without the reliability requirement are presented in Sections 5 and 6, respectively. Section 7 concludes the paper.

2 Background and Related Work

2.1 Background on DCE

DCE [40] is an execution paradigm proposed to use CMPs to speed up single-threaded applications, and it is built upon two superscalar processor cores (called the front processor and the back processor) coupled with a hardware queue (called the result queue), as shown in Fig. 1.

In DCE, a program is first preprocessed by the front processor and then reexecuted by the back processor. The preprocessing in the front processor is the same as the normal execution except for long-latency cache-missing loads, for which an invalid (INV) value is used to substitute the data being fetched from memory, similar to the run-ahead mode in [12] and [23]. Dependent instructions of those invalidated loads are also invalidated through INV flag propagation. Invalidated branches are resolved as if their predictions are correct, and the stores with invalidated addresses simply become *nops*. When a store instruction with a valid address retires, it will not update the data cache and will only update a structure called the run-ahead cache [23] to forward the store value (either valid or invalid) to subsequent loads in the front processor. Instructions retired from the front processor are kept in the result queue and then fetched and reexecuted by the back processor. Unlike traditional superscalar designs for a large instruction window, those in-flight instructions in the result queue are not associated with any centralized resources, thereby making it highly scalable. During the reexecution in the back processor, when a branch misprediction is detected, the misprediction recovery synchronizes the front processor and the back processor by flushing all instructions in the front processor, the back processor, and the result queue and then copying the current architectural state, including the architectural register values and the program counter (PC), from the back processor to the front processor. In DCE, the front processor runs faster because of the virtually ideal level-2 (L2) cache, and the back processor makes faster progress, as the front processor effectively prefetches the data through the shared L2 cache and resolves most branch mispredictions for the back processor. Overall, DCE achieves a significant performance improvement and eliminates the need for any centralized resources.

In DCE, the execution results (if not invalidated) in the front processor, although speculative, are highly accurate, and it is reported in [40] that using the non-INV front execution results as value prediction if the accuracy is over 99.99 percent. The reasons for such a high accuracy include 1) as the front processor only invalidates long-latency cache-missing loads, the execution of independent instructions is not affected, 2) among dependent instructions of those invalidated loads, if the dependency is carried with register data flow, then such dependent instructions are invalidated as well through the INV propagation, and 3) if the dependency is carried with memory data flow, then the INV propagation through store-load forwarding in the load-store queue (LSQ) and the run-ahead cache invalidates most of those dependent instructions. Only in the very rare cases when a store with an invalidated address followed by a load accessing the same location or the replacement of an INV flag from the run-ahead cache could a stale value be fetched, resulting in a wrong execution result. The high accuracy of the speculative execution of the front processor has important implications on performance and resource utilization. First, the front processor resolves most branch mispredictions and provides a highly accurate instruction stream to the back processor. Second, the prefetches initiated by the front processor are highly accurate, and the resources are utilized very efficiently compared to other prefetching techniques. In this paper, we explore this fast, speculative, and highly accurate preprocessing to achieve transient-fault tolerance and enhance the power/energy efficiency.

2.2 Related Work

DCE is a CMP-based approach to building a highly scalable single-thread instruction window. Besides the flexibility of supporting multithreaded workloads, it eliminates the need for *any* centralized resources compared to out-of-order processors with large instruction windows [1], [11], [16], [33], two-pass/multipass in-order pipelining [4], [5], or

decoupled kilo-instruction processors [24]. Run-ahead execution [12], [23] is another alternative to large instruction windows which does not require any large centralized structures. However, in the run-ahead execution, whenever a mode-transition-triggering cache miss is repaired, the processor has to return to the normal mode, even if the preexecution is along the right paths and generates accurate prefetches. DCE effectively overcomes this fundamental limitation and achieves significantly higher performance [40]. The efficiency of the run-ahead execution is examined in [22], and it is found that one main source of inefficiency results from overlapping run-ahead periods; that is, the processor reenters the run-ahead mode due to adjacent cache misses, dependent or independent, and preexecutes the same set of dynamic instructions repeatedly. In DCE, the front processor makes exactly one pass of the instructions and does not have such a drawback. Furthermore, redundant execution in DCE enables efficient ways to achieve transient-fault tolerance, as discussed in Section 4. The energy-saving optimizations proposed in this paper, such as instruction window size adaptation (Section 5.2) and selective instruction invalidation (Section 5.3), can also be used to improve the energy efficiency of those large instruction window designs, as well as run-ahead execution.

Leader/Follower architectures provide an efficient platform for fault tolerance by exploiting the results from the leading thread. In most existing leader/follower architectures for fault tolerance, including AR-SMT [27], DIVA [2], simultaneous and redundant threading (SRT) [26], Simultaneously and Redundantly Threaded processors with Recovery (SRTR) [36], and chip-level redundant threading with recovery (CRTR) [15], both leader and follower thread/processor results are nonspeculative, that is, correct if free of hardware errors. This constraint is a main reason for their performance degradation, since the leader has to wait for the follower to check the execution results before retiring them (for example, store values and store addresses). The delayed retirement increases the pressure on critical resources such as the store queue and register file. To alleviate such an adverse impact, it is proposed in [15] that instructions except stores can commit their states before checking, whereas stores commit to an enlarged store buffer and only update memory after checking. In [38], the verification thread is paralleled to improve the power efficiency of thread-level redundancy checks. In order to overcome the difficult store-load forwarding associated with the large store buffer (named post commit buffer in [38]), it is proposed that the stores update both level-1 (L1) D-cache and the post commit buffer. The dirty data in L1 D-cache is dropped when being replaced, and the post commit buffer is only searched in the case of L1 misses. In contrast to those schemes, the preprocessing in DCE is *speculative* and relieved of the correctness requirement. To our knowledge, slipstream processing [35] is the only other thread-level redundancy scheme whose leading thread (A-stream) is speculative. Such speculative processing is the key to enabling DCE and slipstream processors to achieve both performance enhancement and fault tolerance at the same time.

Although DCE and the slipstream processing share a similar high-level architecture, they achieve performance improvements in fundamentally different ways. In slipstream processors, the A-stream runs a shorter program based on the removal of ineffectual instructions, whereas 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. Compared to slipstream processors, DCE achieves much higher performance improvement with less hardware complexity (that is, no need for IR detectors, IR predictors, and value prediction support). Both slipstream processors and DCE can exploit redundant execution to improve fault tolerance, and the detected faults can be corrected with the existing wrongful speculation recovery mechanisms. Many mechanisms proposed in this paper, such as the dual execution for full redundancy coverage (Section 4.3) and the elimination of redundant execution for power/energy efficiency (Section 6.1), can also be used to enhance the slipstream processing. In a way, DCE can be viewed as a hybrid between the slipstream and the run-ahead executions. It synergistically combines the slipstream and run-ahead executions and uses one to cancel the drawbacks of the other. The run-ahead execution cancels the drawback of slipstream that dead-instruction removal does not give the front processor sufficient lead. Slipstream cancels the drawbacks of the run-ahead execution that constant transitions into and out of the run-ahead mode have high overhead.

DCE is also influenced by other leader/follower architectures such as Master/Slave Parallelization [43] and the preexecution/precomputation paradigm using multithreaded architectures [3], [10], [17], [28], [37], [42]. The comparison between DCE and those leader/follower architectures is detailed in [40]. Also, it is worth highlighting that coupling two (or more) relatively simple processors to form a large instruction window for out-of-order processing was originated in multiscalar processors [32], and DCE provides a complexity-effective way to construct such a window while eliminating elaborate interthread (or intertask) register/memory communication. Other proposals using dual cores to enhance single-thread performance include a dual-core architecture for speculative multithreading [34] and future execution [14]. Besides homogeneous CMPs, smart memories [18] provide a platform on which DCE can be mapped.

3 Methodology

Our simulator infrastructure is built upon the SimpleScalar toolset [8], but our execution-driven timing simulator is completely rebuilt to model MIPS-R10000-style superscalar processors. In our simulator, both data and tag stores are modeled in the cache modules (including the run-ahead cache), and wrong-path events are also faithfully simulated. 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. Our baseline processor, shown in Table 1, is a MIPS-R10000-style out-of-order superscalar processor, which is used for both front and back processors in DCE. The front and back processors share a unified L2 cache. When simulating DCE, the correctness assertions are disabled in the front processor model but enforced in the back processor model. The default DCE setup includes a 4-Kbyte four-way associative run-ahead cache with a block size of 8 bytes, a 1,024-entry result queue with a 16-cycle delay to account for interprocessor communication, separate 32-Kbyte two-way

TABLE 1
Configuration of the Baseline Processor

Pipeline	3-cycle fetch stage, 3-cycle dispatch (decode and 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=1024 kB; Assoc.=8-way; Replacement=LRU; Line size=128 bytes; Miss penalty=*220* cycles. 64 MSHRs
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

L1 caches, and a shared unified 1-Mbyte eight-way L2 cache. A latency of 64 cycles is assumed for copying the architectural register values from the back to the front processors in the case of a branch misprediction. A stride-based stream buffer hardware prefetcher, which has eight four-entry stream buffers with a PC-based two-way 512-entry stride prediction table, is also included in the baseline processor model. All our experiments are performed using SPEC CPU 2000 benchmarks with the same selection criterion and simulation points as described in [40], that is, memory-intensive benchmarks for which an ideal L2 cache introduces at least 40 percent of speedup and two computation-intensive benchmarks, *gap* and *bzip2*, to illustrate interesting aspects of DCE and other competitive approaches.

4 Optimizing DCE for Transient-Fault Tolerance

4.1 Sphere of Replication and Error Protection

In DCE, instructions are fetched from the L1 I-cache of the front processor, retired from the front processor into the result queue, and then forwarded to the back processor. To protect the instruction stream, we propose to include *parity bits* in both the L1 I-cache and the result queue. If a faulty instruction is accessed in the L1 I-cache, then the failed parity check will nullify the cache block and the read access becomes an L1 I-cache miss. If an instruction is corrupted in the result queue, then the back processor simply treats it as a branch misprediction and rewinds the front processor to refetch and reexecute the faulty instruction. For instruction execution, the front and back processors provide spatial redundancy, as discussed in Section 4.2. To protect from transient faults that could result in a deadlock (for example, a ready flag is incidentally flipped so that an instruction can never move forward), a watchdog timer similar to that used in DIVA is added in the back processor to restart the execution from the current architectural state whenever the timer expires. The architectural state of the back processor, including an architectural register file, the PC, and the memory state, needs to be protected with integrity coding schemes such as error correction coding (ECC).

As part of the memory state, the L1 D-cache of the back processor and the shared L2 cache may contain dirty data if the write-back policy is employed, and transient faults to such dirty data could be irrecoverable. Therefore, they also need to be protected with ECC bits.

4.2 Reliability Enhancement Using DCE

DCE enables efficient ways to detect and recover from transient faults. As highlighted in Section 2.1, the execution results from the front processor in DCE, if not invalid, are highly accurate. Therefore, we can simply carry them in the result queue and use them to perform redundancy checking with the back processor results. Any discrepancy, either due to a transient fault during execution or a wrongful speculation by the front processor, will incur a misprediction recovery using the existing branch misprediction mechanism in the back processor. We call DCE with this redundancy check as DCE_R. Note that, although it is possible in DCE_R that a dynamic load may return different values when executed in the front and back processors, such a discrepancy will simply be treated as a speculation error, and the same load will be reexecuted by both the front and back processors. In a sense, the loads executed in the front processor can be viewed as an aggressive load speculation in an out-of-order processor, and the reexecution of the same loads in the back processor is simply a value-based approach to ensure the correct memory ordering [9]. For those nonrepeatable loads such as I/O accesses, the front processor can simply invalidate those instructions and rely on the back processor to process them properly.

Although utilizing the speculative execution results from the front processor conveniently enables redundancy checking in DCE_R, the wrongful speculations in the front processor will incur additional recoveries even in transient-fault-free execution. Fortunately, as discussed in Section 2.1, such wrongful speculations are extremely rare events, and our experimental results confirm the observation (0.02 recoveries per 1,000 retired instructions on the average) and show a negligible performance impact compared to the original DCE (see Fig. 4). On the other hand, as DCE_R exploits the execution results in the front processor to provide redundancy checking, the instructions that are invalidated by the front processor are only executed by the back processor and still susceptible to transient faults. In Fig. 2, we report the redundancy coverage (that is, the percentage of retired

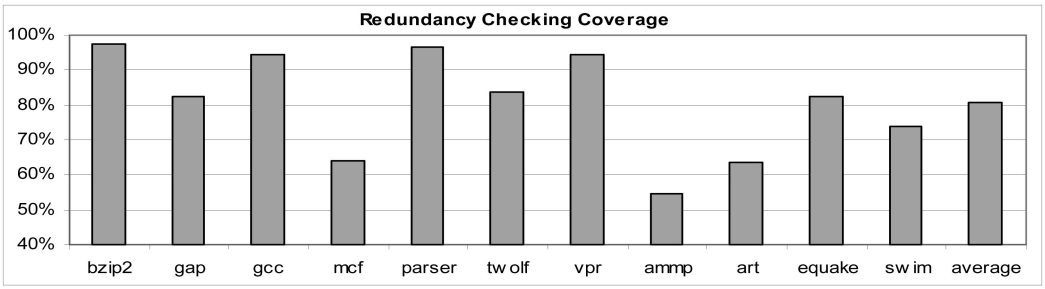


Fig. 2. The percentage of retired instructions with redundancy checking.

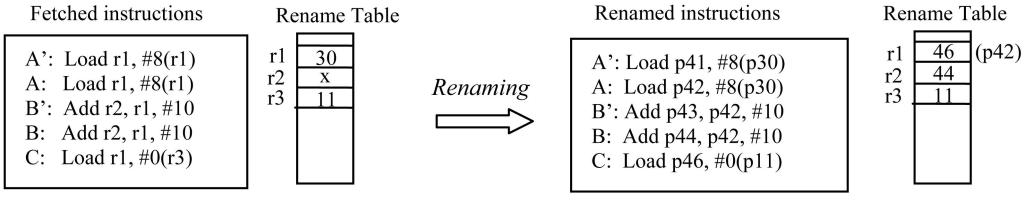


Fig. 3. An example to illustrate the renaming process to achieve full redundancy coverage.

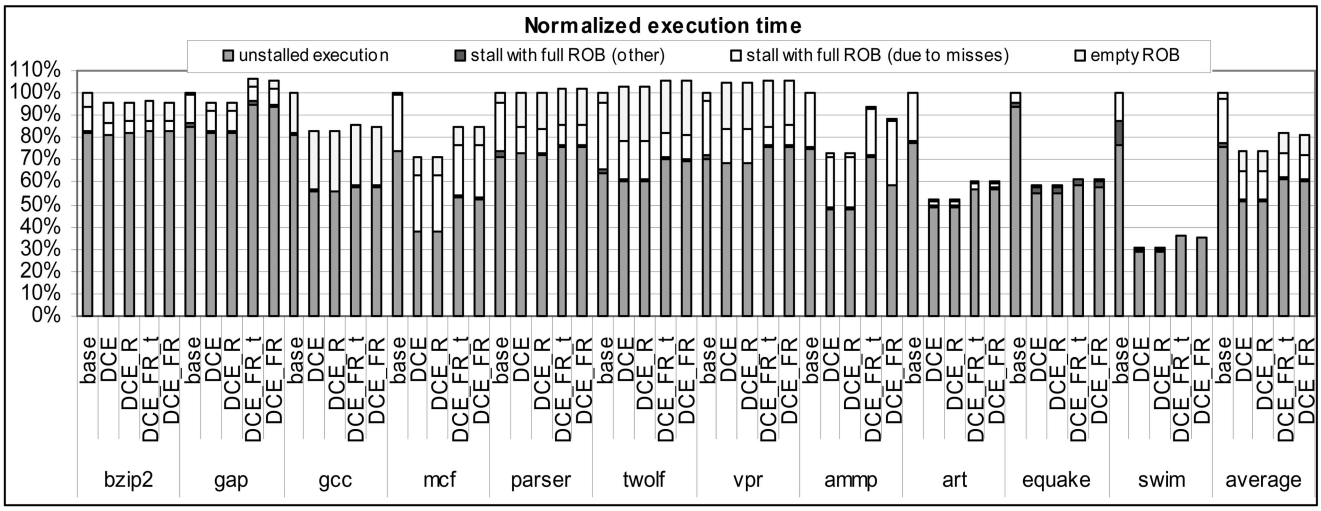


Fig. 4. The performance impact of fault-tolerant DCE.

instructions with redundancy check) for each benchmark by using DCE_R, and it can be seen that DCE_R protects an average of 81 percent of all retired instructions with redundancy checking.

4.3 Achieving Full Redundancy Coverage

In order to achieve redundancy coverage for all instructions, we propose a novel complexity-effective way to extend the back processor in DCE_R so that it dual-executes the instructions that are invalidated by the front processor, and such an extension to DCE for full redundancy is named DCE_FR. The main idea is similar to previous work on the dual use of data path for fault tolerance [25]. The difference is that only a small subset of instructions needs to be executed twice in DCE_FR, and the pipeline replication sphere is extended by replicating instructions as early as in the fetch stage.

In DCE_FR, the result queue appends a flag (F_INV) to each instruction to indicate whether it is invalidated by the front processor. When the back processor fetches an instruction with a true F_INV (meaning invalidated by the front processor), the same instruction will be fetched twice: the first used for redundancy checking and the second for normal execution. The *order* here is important, as it significantly simplifies the required changes in the renaming logic of the back processor to support such redundant execution. For the redundant copy of an instruction with a true F_INV, its source operands access the rename table as usual to get their physical register mappings. Its destination register(s) obtain(s) a new physical register from the free list, but does *not* update the rename table. The renaming process for the original instruction stream, including the original copy of those with a true F_INV and the instructions with a false F_INV, remains unchanged. At the retire stage, the redundant copy frees its destination register(s) right after its results are used to compare with the results of the original instruction. Fig. 3 illustrates this process with an example.

In Fig. 3, instructions A and B are invalidated by the front processor, so they are replicated (A' and B') in the back processor when fetched from the result queue. Instruction C is not invalidated by the front processor and carries a valid result in the result queue for redundancy

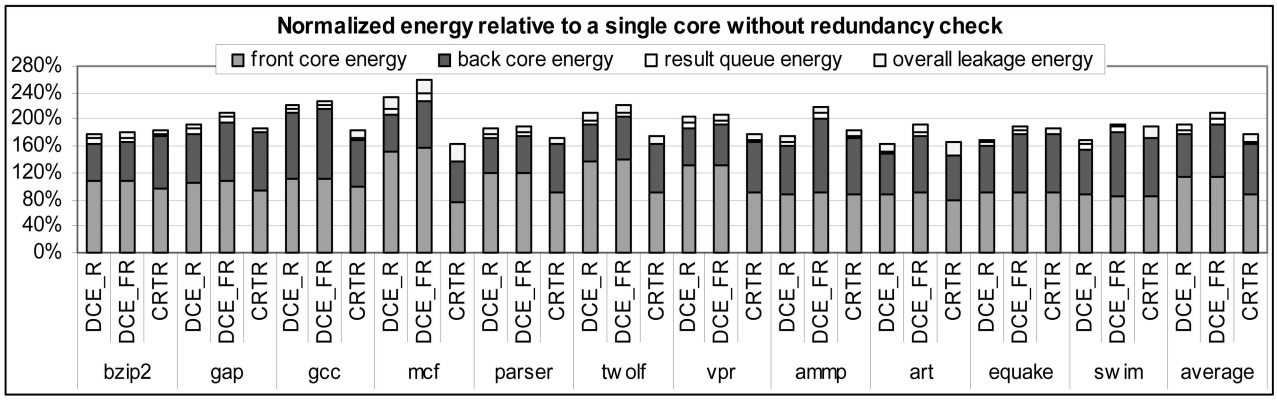


Fig. 5. Energy consumption of DCE_R, DCE_FR, and CRTR normalized to a single core without redundancy checking.

checking. The redundant instructions A' and B' access the rename table for their source operands and then obtain free physical registers for their destination registers. However, those new names will *not* update the rename table. The instructions A, B, and C repeat the same process, except that their destination operands will update the rename table. This example also shows why it is important that the redundant copy needs to be processed before the original instruction. If we exchange the positions of A' and A, then the source operand $(r1)$ of A' would be incorrect, as it would have been already renamed by the instruction A.

After renaming, the redundant instructions are processed in the same way as any other instructions. At the retire stage, they provide a redundancy check for their original correspondents (the instruction immediately following it in the reorder buffer (ROB)). Compared to the renaming scheme proposed in [41], which treats the redundant copies as a separate thread and requires an additional rename table to maintain the correct dependency information, our scheme needs just one rename table, simplifies the register deallocation mechanism, and frees the destination registers of the redundant instructions more promptly (that is, no waiting for redefinition). Dual execution in the back-core lead to the extra pressure on the physical register file, the ROB, and the issue queue. Therefore, such an early release of physical registers is beneficial for register-hungry applications, as it enables younger instructions to be dispatched more promptly and results in higher performance, as seen in Fig. 4.

Fig. 4 shows the normalized execution time of a single baseline processor, DCE, DCE_R, DCE_FR, and DCE_FR with the renaming scheme proposed in [41] (labeled as "DCE_FR_t"). Each cycle is categorized as a pipeline stall with an empty ROB, a stall with a full ROB due to cache misses, a stall with a full ROB due to other factors such as long-latency floating-point operations, or a cycle in unstalled execution. For DCE-based schemes, such cycle time distribution is collected from the back processor. Since DCE_FR provides full coverage of the instruction stream, we do not inject errors in our experiments. Considering the rare occurrence of transient errors, their performance impacts are negligible.

Several important observations can be made based on Fig. 4. First, both DCE_R and DCE_FR achieve a significant performance improvement, 35.7 percent and 23.5 percent on average, respectively, over the single baseline processor because of the large instruction window formed with DCE. Slight performance degradations observed in *parser*, *twolf*, and *vpr* are due to their relatively high number of branch mispredictions dependent on cache-missing loads. Second, compared to DCE, DCE_R has a negligible performance impact, since the number of additional recoveries due to redundancy checking is extremely small. Third, for *gap*, *mcf*, *ammp*, *art*, and *swim*, DCE_FR results in many more nonstall execution cycles than DCE_R, since many instructions are invalidated by the front processor for those benchmarks, slowing down the progress of the back processor. Fourth, the new renaming scheme proposed in this paper performs slightly better than the one in [41], with an improvement of up to 6 percent in the benchmark *ammp* and 1 percent on average, due to its more efficient register deallocation mechanism. Overall, DCE_FR achieves a 23.5 percent performance improvement with redundancy checking for all retired instructions, which is a significant improvement over the prior work on thread-level redundancy. Even excluding the outlier benchmark, *swim*, which features high memory-level parallelism that can be exploited with a large instruction window, DCE_FR improves the performance by 17.3 percent on the average over the single core.

5 Power-Efficient Fault-Tolerant DCE

5.1 Energy Consumption of DCE-Based Paradigms

To analyze the power/energy efficiency of the DCE-based schemes, both WATTCH [6] and HotLeakage [39] are ported into our simulator to account for dynamic and static energy consumption. In our experiments, we use the 70-nm technology with a clock frequency of 5.6 GHz and assume linear clock gating [6]. The normalized energy consumption of DCE_R and DCE_FR relative to a single baseline processor without redundancy checking is shown in Fig. 5. For comparison, a CMP-based thread-level redundancy scheme CRTR [15] is also included. In the CRTR model, a 32-entry store buffer is included in the leading processor, and the interprocessor communication latency is assumed as 16 cycles, same as in DCE. The execution time of CRTR is very close to the single baseline processor, except the benchmark *gcc*, for which it incurs a 43 percent

slowdown even with the 32-entry store buffer (it is fully 54 percent of time for *gcc*, and a further increase to the 64-entry effectively recovers the lost performance). The energy consumption in Fig. 5 is broken down to the dynamic energy consumed by the front/leading processor (including the dynamic energy of the shared L2 cache), the back/trailing processor, the result queue, and the leakage for all components.

In Fig. 5, it can be seen that both DCE_R and DCE_FR incur significant energy overhead. The back core in DCE_FR needs to dual-execute the invalidated instructions to provide full redundancy coverage, thereby consuming more energy than DCE_R. Compared to CRTR, DCE_R has more energy overhead for all integer benchmarks except *bzip2* and less energy overhead for floating-point benchmarks. DCE_FR, on the other hand, consumes more energy than CRTR: up to 98 percent in *mcf* and 33 percent on average.

The main sources of the energy overhead associated with the DCE-based schemes are identified as follows:

1. Wrong-path instructions executed in the front processor following the branch mispredictions are resolved in the front processor, which also account for the wasted energy in the baseline single core and the CRTR scheme.
2. Wrong-path instructions executed in both front and back processors following the branch mispredictions are resolved in the back processor. Given the large instruction window formed with DCE, one such branch misprediction can potentially result in thousands of wrong-path instructions being fetched and executed before the misprediction is detected, and this is the main reason that DCE_R and DCE_FR incur much more energy overhead for the benchmarks *mcf*, *parser*, *twolf*, and *vpr*. Here, note that, with the nonuniform branch handling in DCE, that is, mispredictions dependent on short-latency operations are resolved promptly in the front core and only those dependent on long-latency cache misses are resolved in the back processor, the branch misprediction rate at the back processor is actually quite low: 0.65 per 1,000 retired instructions on average. However, with the very large instruction window, such mispredictions still present a major source of power/energy inefficiency. CRTR, in contrast, does not have such a drawback, as all the mispredictions are resolved in the leading processor.
3. The instructions with invalidated source operands, although producing no useful results, still need to access structures including the issue queue, ROB, register file, rename table, and so forth for INV propagation.
4. As a result of such invalidation, the back processor in DCE_FR has to dual-execute those invalidated instructions to achieve full redundancy coverage, thereby consuming more energy.

The advantages or potential energy savings of the DCE-based schemes are through the reduced execution time, especially when the leakage energy becomes dominant in deep-submicron technologies. Here, note that the inclusion of the back core does not increase static energy too much, since the 1-Mbyte L2 cache is the main source of leakage energy consumption. Based on the energy overhead analysis, we next propose simple optimizations to improve the power/energy efficiency of DCE-based schemes in Sections 5.2 to 5.4 and examine their results in Section 5.5. Although the proposed optimizations are applicable to both DCE_R and DCE_FR, we focus our discussion on DCE_FR in this section due to the space limitation.

```
In the back processor, for every 1M retired instructions,
    if (the branch misprediction rate in the back processor > 0.6 per 1K insn)
        current queue size = 128;
    else if (the branch misprediction rate in the back processor > 0.3 per 1K insn)
        current queue size = 256;
    else if(the branch misprediction rate in the back processor > 0.15 per 1K insn)
        current queue size = 512;
    else
        current queue size = 1024;
```

Fig. 6. The algorithm to adaptively adjust the instruction window size.

5.2 Adapting Instruction Window Sizes

As discussed in Section 5.1, the energy overhead of DCE_FR mainly comes from the high cost of branch mispredictions that are dependent on long-latency cache misses. For brevity, we refer to those branch mispredictions as "*important mispredictions*" in the rest of the paper. To reduce the energy overhead associated with important mispredictions, we can either improve the branch prediction accuracy or reduce the impact of each misprediction. The first option is out of the scope of this paper and is left as future work. For the second option, since the cost of each important misprediction is directly related to the instruction window size, if the window size can be adaptively adjusted based on workload characteristics, then we can improve the power/energy efficiency while retaining the performance benefits of large instruction windows.

Unlike other large instruction window designs, the instruction window in the DCE-based schemes is mainly formed with the result queue, which enables very simple ways to change the window size (that is, no involvement of other structures). As the result queue is a circular first-in, first-out (FIFO) structure, in order to change the logical queue size, we only need to change the hardware to determine how its head pointer and tail pointer will be updated. In other words, rather than the pointer advancement logic $\mathrm{head/tail} = (\mathrm{head/tail} + 1)$ mod queue_size, we add a register to maintain the current queue size (cur_queue_size) and change the pointer advancement logic to $\mathrm{head/tail} = (\mathrm{head/tail} + 1)$ mod cur_queue_size. As the current queue size will always be configured as a 2's power, the logic is simply an addition followed by a shift operation. With the hardware support, we propose an algorithm to periodically adjust the instruction window size, as shown in Fig. 6.

The key idea behind the algorithm in Fig. 6 is to reduce the window size for those workloads/execution phases that feature a high important-misprediction rate and to fully exploit large-window benefits for others. Once a new queue size is determined, the actual change takes effect when the queue is squashed at the next important-misprediction recovery. The parameters in the algorithm, that is, those

1. For every 1M retired instructions, the following statistics are collected:
   Number of L2 cache misses (L2 miss rate).
   Number of branch mispredictions dependent on L2 cache misses (I_br misp rate).
2. Calculate the following conditions,
   Condition A: the L2 miss rate > 50 per 1K insns AND the I_br misp rate < 2.5 per 1K insns.
   Condition B: the L2 miss rate > 25 per 1K insns AND the I_br misp rate < 0.25 per 1K insns.
   Condition C: the L2 miss rate > 2.5 per 1K insns AND the I_br misp rate < 0.02 per 1K insns.
3. If the invalidation is disallowed and (Condition A OR Condition B OR Condition C), enable the invalidation.
4. If the invalidation is allowed and !(Condition A OR Condition B OR Condition C), disable the invalidation.

Fig. 7. The algorithm to adaptively enable/disable the invalidation in the front processor.

threshold misprediction rates, are determined empirically, and our experiments indicate that the algorithm is quite robust for changes in those thresholds.

5.3 Selective Invalidation of Cache-Missing Loads

Using the terminology defined in [21], a load instruction can be distinguished between an address load and a data load. An address load can be further classified as a traversal address load, that is, a static load that produces an address to be consumed by itself or another address load, or a leaf address load, that is, a static load that produces an address to be consumed by a data load. In DCE-based schemes, invalidating a traversal address load can lead to a large portion of instructions being invalidated through the INV propagation if subsequent iterations depend on its loaded result, which, in turn, increases the chances of important mispredictions and the number of instructions to be dual-executed in the back processor. Therefore, we propose to *not* invalidate such traversal address loads, even if they miss in the L2 cache. Besides potential energy savings from a more accurate instruction stream and fewer invalidated instructions, such uninvalidation can also improve the effectiveness of DCE when there are other cache-missing loads dependent on a traversal address load. Not invalidating this particular load enables the front processor to execute the dependent loads and prefetch the data for the back processor.

Accurate identification of all traversal address loads requires code analysis and potential compiler support. Therefore, in this paper, we only focus on a special type of traversal address loads, that is, "load ra, x(ra)," that can be easily detected at the decode stage. Then, those detected traversal address loads are prevented from being invalidated, even when they miss in the L2 cache.

5.4 Adaptively Enable/Disable the Invalidation in the Front Processor

As pointed out in [40], DCE is proposed to hide memory-access latencies for memory-intensive workloads and it is not suitable for computation-intensive applications as there are not many cache-missing loads to be invalidated for the front processor to make faster progress. In addition, even if an application is relatively memory intensive, DCE can still hurt the performance if the workload has a high important-misprediction rate due to the high cost of misprediction recovery. Furthermore, in DCE_FR, the invalidation also increases burden to the back processor as it needs to dual-execute those invalidated instructions. To address this problem, we propose an algorithm to adaptively enable/disable the invalidation of cache-missing loads based on a workload's dynamic behavior, as shown in Fig. 7.

The algorithm in Fig. 7 lists three scenarios where invalidating cache-missing loads is beneficial. The first (that is, condition A) is for heavily memory-intensive workloads/phases with a moderate important-misprediction rate. As we still want to take advantage of the fast preprocessing of the front core, we choose to allow the invalidation and resort to the window-size adaptation to control the energy overhead. The second (that is, condition B) is memory-intensive applications/phases with a low important-misprediction rate, which is the typical case for DCE to fully utilize its large instruction window. The third one (that is, condition C) represents programs/phases with moderate memory intensiveness and extremely low important-misprediction rate for which the large instruction window also helps to improve the performance. If any of these three conditions is true, then the invalidation should be enabled. Otherwise, it will be disabled. Here, note that even if the invalidation is disabled, the front processor results are still *speculative*, as stores commit to the run-ahead cache instead of the L1 D-cache, and instructions retire without waiting for the redundancy checking from the back processor. When the invalidation is enabled, it does not necessarily mean that all cache-missing loads will be invalidated, since some of them may be detected as traversal address loads and are not allowed to be invalidated using the criterion in Section 5.3.

The important-misprediction rate is easy to obtain if the invalidation is enabled, as it is simply the misprediction rate in the back processor. If the invalidation is disabled, however, then all mispredictions are resolved in the front processor. To differentiate important mispredictions from the rest, we add a time stamp to each shadow map and set it when the shadow map is allocated for a branch. When a misprediction is resolved, the difference between the current time stamp and the one in its shadow map tells the latency to resolve this branch. If such latency is greater than 100 cycles, then an important misprediction is detected.

5.5 Experimental Results

We first examine the performance impact of the optimizations proposed in Sections 5.2 to 5.4. The normalized execution time of these schemes relative to a single baseline processor is shown in Fig. 8, where "FR_rs" represents DCE_FR with an adaptive-sized result queue, "FR_rs_tl" is "FR_rs" augmented with the noninvalidation of traversal address loads, and "FR_rs_tl_in" is "FR_rs_al" combined with the adaptive control of invalidation in the front processor. In Fig. 8, it can be seen that the overall performance impacts of the proposed optimizations are limited compared to the original DCE_FR. Using an adaptive-sized instruction window introduces minor performance degradation for the benchmarks *bzip2*, *mcf*, *parser*, *twolf*, and *vpr*, whereas it benefits *art* slightly. Noninvalidation of traversal address loads improves the performance

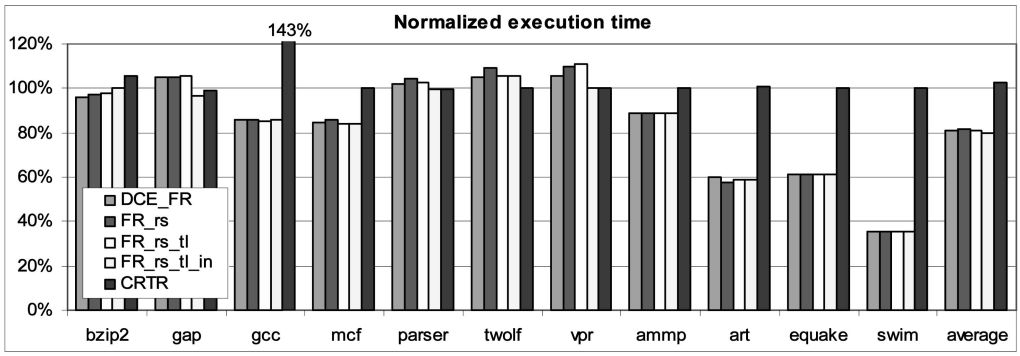


Fig. 8. The normalized execution time of DCE_FR, optimized DCE_FR, and CRTR.

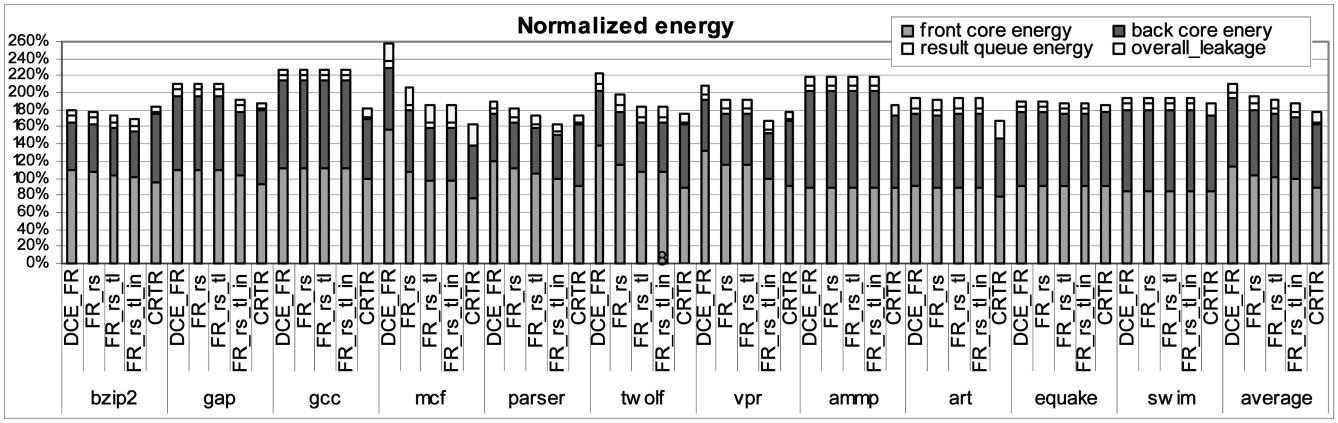


Fig. 9. The normalized energy consumption of DCE_FR, optimized DCE_FR, and CRTR.

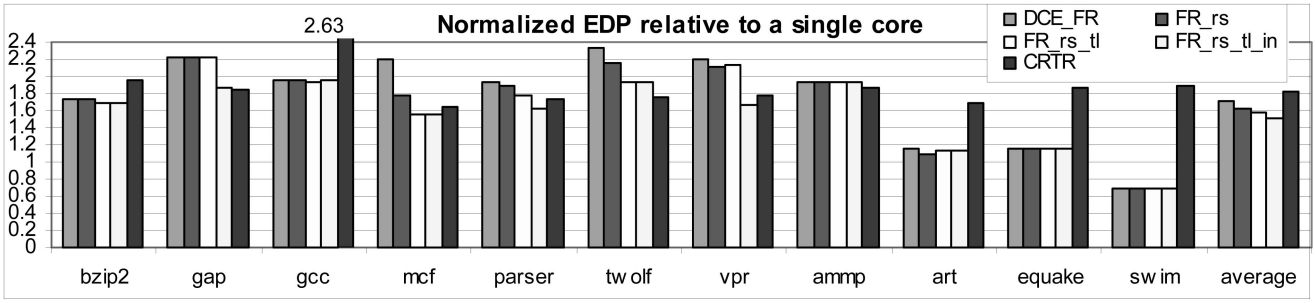


Fig. 10. The normalized energy-delay products (EDPs) of DCE_FR, optimized DCE_FR, and CRTR.

for *mcf*, *parser*, and *twolf*, as many important mispredictions are eliminated. The additional capability to dynamically control the invalidation at the front processor improves the performance for *parser*, *twolf*, and *vpr*, as the invalidation is always disabled for those benchmarks due to their high important-misprediction rates. For *gcc*, *mcf*, *ammp*, *art*, *equake*, and *swim*, in contrast, the invalidation is enabled most of the time, as they are either heavily memory intensive (for example, *mcf*) or memory intensive with low important-misprediction rates (for example, *equake*). For computation-intensive workloads *bzip2* and *gap*, the invalidation is disabled most of the time and only enabled for few phases with relatively more memory operations, resulting in slight performance improvement for *gap*.

The impacts of the proposed optimizations on energy consumption and power/energy efficiency, both normalized to the single-core processor, are shown in Figs. 9 and 10, respectively. In Fig. 9, it can be seen that the proposed optimizations significantly reduce the energy overhead associated with DCE_FR: from 110 percent over a single core down to 87 percent on the average. The energy savings mainly come from the front processor, as it executes much fewer instructions when the window size is reduced (for example, *mcf*), or the number of important mispredictions is reduced (for example, *parser*, *twolf*, and *vpr*). Compared to CRTR, the optimized DCE_FR only incurs a small energy overhead (87 percent versus 77 percent over a single core) while achieving significant performance improvements (a 24.9 percent speedup versus a 2.7 percent slowdown on the average).

Using normalized EDP [13] relative to a single core processor as a power/energy-efficiency metric, Fig. 10 shows that the proposed optimizations effectively improve the power/energy efficiency for DCE_FR (from 1.71 to 1.51). Compared to CRTR, the optimized DCE_FR achieves better power/energy efficiency on average (1.51 versus 1.83) and for all individual benchmarks except *gap*, *twolf*, and *ammp*. With a metric emphasizing more on performance, for

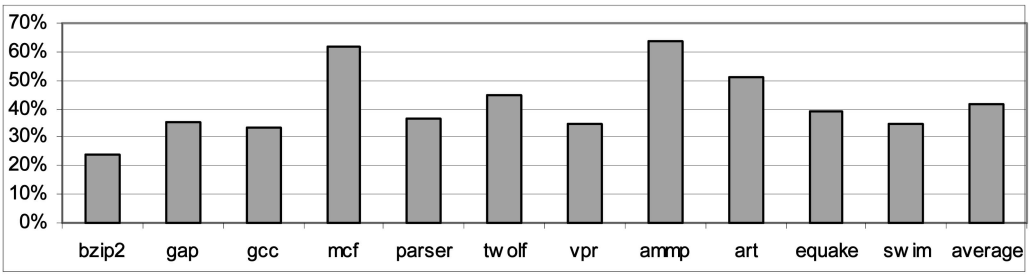


Fig. 11. The ratio of the number of executed instructions over the number of retired instructions in the back processor.

example, $\text{energy} - \text{delay}^2$ product (ED^2P) [7], *gap* and *ammp* also report higher efficiency than CRTR, given their performance advantage. Eliminating the outlier benchmark *gcc* improves the average EDP of CRTR from 1.83 to 1.78, but it is still not as efficient as the optimized DCE_FR. Note that, in Fig. 10, the average EDP is computed using $(\Sigma\ \text{Delay})^*(\Sigma\ \text{Energy})$, as suggested in [29].

6 Power/Energy-Efficient DCE

As addressed in Section 5.1, the energy overhead of DCE is mainly due to two sources: wrong-path instructions and redundant execution. The optimizations proposed in Sections 5.2 and 5.3 aim to reduce the energy overhead associated with wrong-path instructions. In this section, we focus on reducing unnecessary redundant execution for systems without a strong reliability requirement.

6.1 Reducing Redundant Execution in DCE

In DCE, instructions are reexecuted in the back processor after having been preprocessed by the front processor. Such reexecution serves two purposes: 1) ensuring execution correctness as the front processor executes instructions in a speculative way and 2) providing redundancy checking, if necessary. With the relief of the reliability requirement, the only objective left is to ensure the correctness of program execution, as the reexecution corrects any wrongful speculation made by the front processor.

As highlighted in Section 2.1, apart from the invalidated instructions, the execution results from the front processor are actually highly accurate, and the only reason for a wrong result is due to incorrect memory processing. For example, a store with an INV address will make all the subsequent loads speculative, as a stale value may be loaded if the same "unknown" address is accessed. The replacement of an INV value from the run-ahead cache is another reason for a load to fetch a stale value. Branch mispredictions due to an invalidated operand, however, do not present a correctness issue, as all the invalidated instructions will be reexecuted in the back processor anyway. Since load instructions are the only source to produce potentially incorrect but still valid results in the front processor, the back processor does not need to reexecute every instruction to ensure the correctness. Instead, it only needs to reexecute all the loads and all the invalidated instructions. For other valid instruction results, the back processor can simply incorporate them into the register file and let those instructions bypass the execution engine. For valid load instructions, the back processor checks whether the reloaded values match with the previously loaded data. If not, the back processor simply incurs a branch misprediction recovery to restart from the current architectural state. This way, power/energy consumption can be saved in the back processor if a significant number of instruction reexecutions can be avoided.

6.2 Switching between DCE and Single-Core Execution

As addressed in Section 5.4, DCE is not an efficient paradigm for certain types of workloads/execution phases such as computation-intensive applications or those with high important-misprediction rates. For those workloads/phases, it would be more energy efficient to completely disable DCE and use a single core to execute them.

The switch from the dual-core mode (or called the single-threaded mode in [40]) to the singe-core mode (called the multithreaded mode in [40]) is similar to a branch misprediction recovery in the back processor: The architectural state at the back processor is copied to the front processor, the front processor fetches, processes, and retires instructions in its normal way with its invalidation and the run-ahead cache being disabled, and the back processor either becomes idle or starts executing a new thread by fetching instructions from its own I-cache. To switch from the single-core mode back to the dual-core 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 instructions from the result queue.

The algorithm to determine whether/when the single-core or dual-core mode should be used is similar to the one to control invalidation in Fig. 7, as they are essentially exploiting the same workload behavior. The only difference is replacing "enabling invalidation" with "switching to the dual-core mode" and "disabling invalidation" with "switching to the single-core mode."

6.3 Experimental Results

The effectiveness of the approach proposed in Section 6.1 is shown in Fig. 11, which reports the ratio of the number of executed instructions over the number of retired instructions in the back processor for each benchmark. As Fig. 11 shows, the back processor only needs to reexecute a small portion of all instructions: 41 percent on average, with a minimum of 24 percent for *bzip2* and a maximum of 63 percent for *ammp*, to ensure the correctness. The performance cost due to load value mismatch in the back processor is very limited, as there are only 0.12 mismatches per 1,000 retired instructions on average.

The impacts of the proposed optimizations on performance and energy consumption are shown in Figs. 12 and

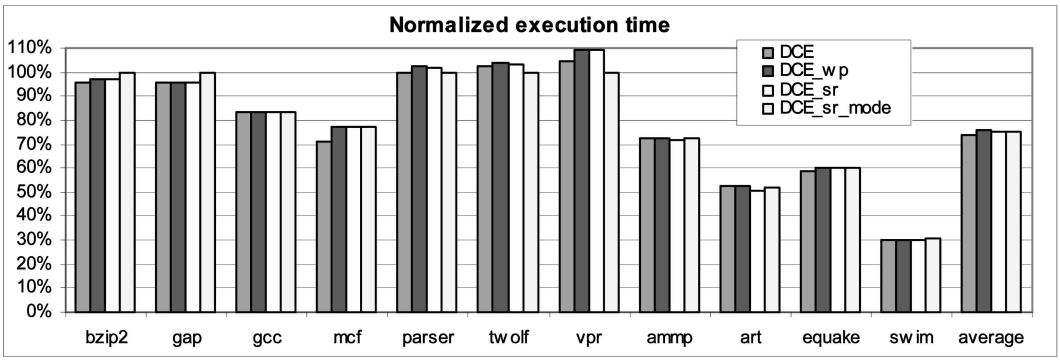


Fig. 12. The normalized execution time of DCE and optimized DCE.

13, respectively. Among the results, those labeled as "DCE_wp" are based on the optimizations proposed in Sections 5.2 and 5.3, since those optimizations target at wrong-path instructions and are also effective in improving the power/energy efficiency for DCE without the reliability requirement. As seen in Figs. 12 and 13, with the optimizations proposed in Section 5.2 and 5.3, the energy overhead of DCE is reduced from 93 percent to 77 percent on the average, and the execution time is increased up to 5 percent for *mcf* and 2 percent on the average. Therefore, in the remaining experiments, we always include those optimizations and evaluate additional energy savings achieved by reducing redundant execution. To highlight the energy overhead of redundant execution, we use an ideal branch predictor to model the impact of accurate control flow. The results show that, by eliminating wrong-path instructions, the average energy overhead can be further reduced to 61 percent, and the average execution time can be reduced by 42 percent, which indicates that a better branch predictor will be an efficient way to improve the energy efficiency of DCE.

With the significant reduction of instruction reexecution, the selective reexecution optimization proposed in Section 6.1 reduces the energy overhead from 77 percent to 59 percent on average at very little performance cost, as seen in Figs. 12 and 13 (labeled "DCE_sr"). The additional capability to adaptively switch between the single-core and dual-core modes enables DCE to be used more efficiently. For those benchmarks that are either computation intensive (for example, *gap*) or feature a high important-misprediction rate (for example, *parser*, *twolf*, and *vpr*), the front core operates in the single-core mode most of the time, and the back core simply remains idle. This way, the adverse impacts of DCE on both performance and energy consumption are eliminated, as seen from the results labeled "DCE_sr_mode" in Figs. 12 and 13. For the remaining benchmarks, for example, *equake* and *swim*, their phase behavior is exploited for more aggressive energy reduction. Overall, the mode switching reduces the energy overhead from 59 percent to 31 percent on average and still achieves 34 percent of performance improvement over a singe core.

The power/energy efficiency results measured in EDP, as reported in Fig. 14, show that the proposed optimizations significantly improve the power/energy efficiency of DCE. Compared to the single-core baseline processor, the optimized DCE achieves much better power/energy efficiency for *art*, *equake*, and *swim* and equivalent efficiency for *bzip2*, *gap*, *parser*, *twolf*, and *vpr*. For the remaining benchmarks *gcc*, *mcf*, and *ammp*, their substantial performance gains (20 percent, 30 percent, and 38 percent speedups, respectively) can be desirable for systems emphasizing more on performance. In addition, the mode switching algorithm in Section 6.2 offers the flexibility to explore different performance-energy trade-offs by adjusting the parameters in the algorithm. For example, by simply increasing the required L2 miss rate in "condition B" from 25 to 30 per 1,000 instructions in Fig. 7, the single-core mode will be used most of the time for *gcc*.

When using ED^2P as the energy-efficiency metric, the optimized DCE reports an average of 26.7 percent of improvement over the single-core processor. The implication of such improvement is that, if the system is equipped with dynamic voltage and frequency scaling capabilities, then additional opportunities exist for energy-efficiency enhancement by exploiting the superlinear effect of the operating voltage on dynamic power/energy consumption [7]. For example, based on the same assumptions as in [30] that the

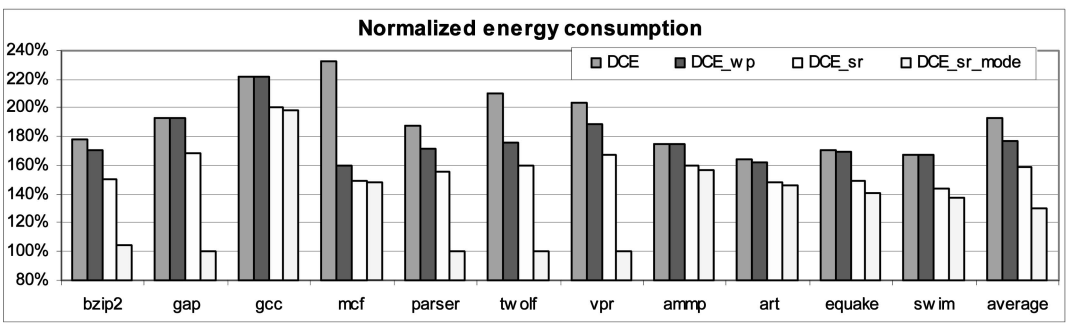


Fig. 13. The normalized energy consumption of DCE and optimized DCE.

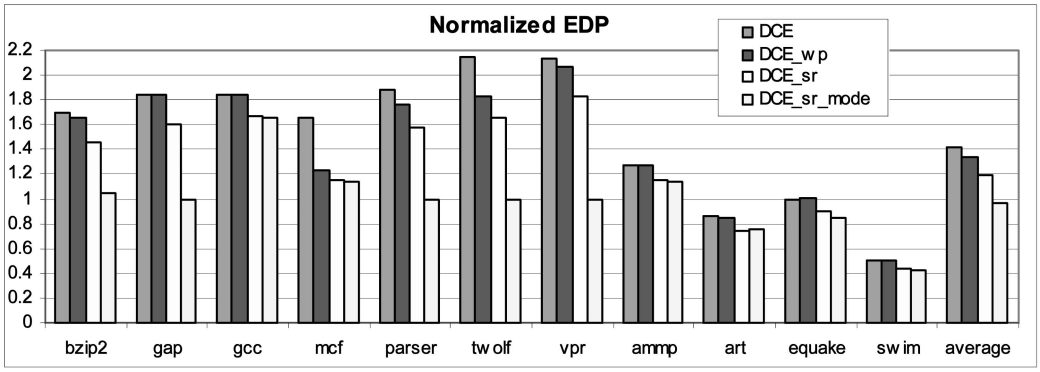


Fig. 14. The normalized EDPs of DCE and optimized DCE.

full frequency range is twice the full voltage range, and there is a linear relationship between frequency and voltage, the frequency-voltage pair is scaled down from (5.6 GHz, 1.0 v) to (4.48 GHz, 0.9 v), that is, a 20 percent frequency reduction along with a 10 percent voltage drop. Then, our simulation results show that the optimized DCE achieves an average of 16.5 percent of performance improvement with a 9.3 percent energy overhead compared to the single-core processor. If the voltage can be scaled down more aggressively for the same frequency change, for example, from (5.6 GHz, 1.0 v) to (4.48 GHz, 0.85 v), then a 3.6 percent energy *reduction* can be achieved, along with the 16.5 percent performance improvement. The voltage and frequency scaling results above are obtained using the same scaling for both the front processor and the back processor. The decoupled structure in DCE also enables opportunities for more aggressive energy savings by applying frequency-voltage scaling to the front processor and the back processor separately. Since the front processor generally can reach a high IPC due to its virtually ideal L2 cache, its operating voltage and frequency can be further reduced compared to the back processor. The detailed evaluation is left for future work.

7 Conclusions

DCE is an approach recently proposed to improve the performance of single-threaded applications using CMPs. In this paper, we propose simple extensions to DCE to exploit its inherent redundant execution, and the extended DCE achieves both performance enhancement and transient-fault tolerance simultaneously while incurring only minor hardware cost.

Then, we perform a detailed analysis of energy overhead associated with DCE-based paradigms, and we identify that a main cause for their energy inefficiency is the high number of wrong-path instructions executed in the large instruction window formed with DCE. To negate such an adverse impact while retaining the benefits of large instruction windows, we propose simple schemes to adaptively adjust the instruction window size and selectively invalidate cache-missing loads based on the runtime behavior of workloads. For systems without a strong reliability requirement, an interesting way is proposed to further improve their power/energy efficiency by reducing the redundant execution in DCE. As shared by other large instruction window designs, DCE works best with memory-intensive workloads with a low important-misprediction (that is, branch mispredictions dependent on long-latency cache-missing loads) rate. For other applications/execution phases, the limited performance gains usually do not justify the energy overhead associated with DCE. To solve this problem, we develop a dynamic scheme which enables DCE only when it is beneficial and falls back to the single-core execution otherwise.

Our experimental results show that the proposed simple optimizations effectively improve the power/energy efficiency. The optimized DCE with full redundancy checking achieves a 24.9 percent speedup over a single core without fault tolerance at a cost of 87 percent of energy overhead over the single core and is a much more power/energy-efficient scheme than a previously proposed transient-fault tolerance approach for CMPs [15]. Without the reliability requirement, the optimized DCE further improves the performance to a 34 percent speedup and lowers the energy overhead to 31 percent and, therefore, becomes even more power/energy efficient than the single-core processor.

Acknowledgments

The authors thank the anonymous reviewers for their insightful and valuable comments.

References

[1] H. Akkary, R. Rajwar, and S. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," *Proc. 36th Int'l Symp. Microarchitecture (MICRO-36)*, 2003.

[2] T. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *Proc. 32nd Int'l Symp. Microarchitecture (MICRO-32)*, 1999.

[3] R. Balasubramonian, S. Dwarkadas, and D. Albonesi, "Dynamically Allocating Processor Resources between Nearby and Distant ILP," *Proc. 28th Int'l Symp. Computer Architecture (ISCA-28)*, 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. 36th Int'l Symp. Microarchitecture (MICRO-36)*, 2003.

[5] R. Barnes, S. Ryoo, and W. Hwu, "Flea-Ficker Multipass Pipelining: An Alternative to the High-Power Out-of-Order Offense," *Proc. 38th Int'l Symp. Microarchitecture (MICRO-38)*, 2005.

[6] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimization," *Proc. 27th Int'l Symp. Computer Architecture (ISCA '00)*, 2000.

[7] D. Brooks, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P. Cook, "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors," *IEEE Micro*, vol. 20, no. 6, 2000.

[8] D. Burger and T. Austin, "The SimpleScalar Tool Set v2.0," *Computer Architecture News*, vol. 25, June 1997.

[9] H. Cain and M. Lipasti, "Memory Ordering: A Value-Based Approach," *Proc. 31st Int'l Symp. Computer Architecture (ISCA '04),* 2004.

[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," *Proc. 28th Int'l Symp. Computer Architecture (ISCA '01),* 2001.

[11] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-Order Commit Processors," *Proc. 10th Int'l Symp. High Performance Computer Architecture (HPCA-10),* 2004.

[12] J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-Executing Instructions under a Cache Miss," *Proc. ACM Int'l Conf. Supercomputing (ICS '97),* 1997.

[13] M. Horowitz, T. Indermaur, and R. Gonzalez, "Low Power Digital Design," *Proc. Int'l Symp. Low Power Electronics,* 1994.

[14] I. Ganusov and M. Burtscher, "Future Execution: A Hardware Prefetching Technique for Chip Multiprocessors," *Proc. Int'l Conf. Parallel Architecture and Compilation Techniques (PACT '05),* 2005.

[15] M. Gomma, C. Scarbrough, T. Vijaykumar, and I. Pomeranz, "Transient-Fault Recovery for Chip Multiprocessors," *Proc. 30th Int'l Symp. Computer Architecture (ISCA '03),* 2003.

[16] A.R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A Large, Fast Instruction Window for Tolerating Cache Misses," *Proc. 29th Int'l Symp. Computer Architecture (ISCA '02),* 2002.

[17] C.K. Luk, "Tolerating Memory Latency through Soft-Ware-Controlled Pre-Execution in Simultaneous Multithreading Processors," *Proc. 28th Int'l Symp. Computer Architecture (ISCA '01),* 2001.

[18] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz, "Smart Memories: A Modular Reconfigurable Architecture," *Proc. 27th Int'l Symp. Computer Architecture (ISCA '00),* 2000.

[19] S. Mukherjee, J. Emer, and S. Reinhardt, "The Soft Error Problem, An Architectural Perspective," *Proc. 11th Int'l Symp. High Performance Computer Architecture (HPCA-11),* 2005.

[20] S. Mukherjee, M. Kontz, and S. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," *Proc. 29th Int'l Symp. Computer Architecture (ISCA '02),* 2002.

[21] O. Mutlu, H. Kim, and Y. Patt, "Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns," *Proc. 38th Int'l Symp. Microarchitecture (MICRO-38),* 2005.

[22] O. Mutlu, H. Kim, and Y. Patt, "Techniques for Efficient Processing in Runahead Execution Engines," *Proc. 32nd Int'l Symp. Computer Architecture (ISCA '05),* 2005.

[23] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," *Proc. Ninth Int'l Symp. High Performance Computer Architecture (HPCA-9),* 2003.

[24] M. Pericas, A. Cristal, R. Gonzalez, D. Jimenez, and M. Valero, "A Decoupled KILO-Instruction Processor," *Proc. 12th Int'l Symp. High Performance Computer Architecture (HPCA-12),* 2006.

[25] T. Ray, J. Hoe, and B. Falsafi, "Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery," *Proc. 34th Int'l Symp. Microarchitecture (MICRO-34),* 2001.

[26] S. Reinhardt and S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," *Proc. 27th Int'l Symp. Computer Architecture (ISCA '00),* 2000.

[27] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," *Proc. 29th Int'l Fault-Tolerant Computing Symp. (FTCS-29),* 1999.

[28] A. Roth and G. Sohi, "Speculative Data Driven Multithreading," *Proc. Seventh Int'l Symp. High Performance Computer Architecture (HPCA-7),* 2001.

[29] Y. Sazeides, R. Kumar, D. Tullsen, and T. Constantinou, "The Danger of Interval-Based Power Efficiency Metrics: When Worst Is Best," *Computer Architecture Letters,* Jan. 2005.

[30] G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. Scott, "Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling," *Proc. Eighth Int'l Symp. High Performance Computer Architecture (HPCA-8),* 2002.

[31] J. Smolens, J. Kim, J. Hoe, and B. Falsafi, "Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitecture," *Proc. 37th Int'l Symp. Microarchitecture (MICRO-37),* 2004.

[32] G. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA '95),* 1995.

[33] S.T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual Flow Pipelines," *Proc. 11th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-11),* 2004.

[34] S.T. Srinivasan, H. Akkary, T. Holman, and K. Lai, "A Minimum Dual-Core Speculative Multi-Threading Architecture," *Proc. 22nd IEEE Int'l Conf. Computer Design (ICCD),* 2004.

[35] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream Processors: Improving Both Performance and Fault Tolerance," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-9),* 2000.

[36] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-Fault Recovery Using Simultaneous Multithreading," *Proc. 29th Int'l Symp. Computer Architecture (ISCA '02),* 2002.

[37] 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," *Proc. Eighth Int'l Symp. High-Performance Computer Architecture (HPCA-8),* 2002.

[38] M. Wasiur-Rashid, E. Tan, M. Huang, and D. Albonesi, "Exploiting Coarse-Grain Verification Parallelism for Power-Efficient Fault Tolerance," *Proc. 14th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '05),* 2005.

[39] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotleakage: A Temperature-Aware Model of Sub-Threshold and Gate Leakage for Architects," Technical Report CS-2003-05, Dept. of Computer Science, Univ. of Virginia, 2003.

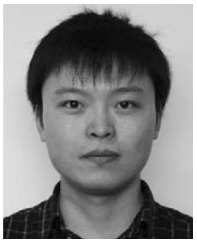
[40] H. Zhou, "Dual-Core Execution: Building a Highly Scalable Single-Thread Instruction Window," *Proc. 14th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '05),* 2005.

[41] H. Zhou, "A Case for Fault-Tolerance and Performance Enhancement Using Chip Multiprocessors," *Computer Architecture Letters,* Sept. 2005.

[42] C. Zilles and G. Sohi, "Execution-Based Prediction Using Speculative Slices," *Proc. 28th Int'l Symp. Computer Architecture (ISCA '01),* 2001.

[43] C. Zilles and G. Sohi, "Master/Slave Speculative Parallelization," *Proc. 35th Int'l Symp. Microarchitecture (MICRO-35),* 2002.

Yi Ma received the BE degree in computer science and technology from Zhejiang University, Hangzhou, China. She is currently working toward the PhD degree in the Department of Computer Science, University of Central Florida. Her current research focuses on high-performance, low-power multithreaded, and multicore architectures.



Hongliang Gao received the BE degree in computer science and technology from Beihang University, Beijing, in 2001. He is currently a PhD student in the School of Electrical Engineering and Computer Science, University of Central Florida. His research interests include novel techniques for control flow speculation, fault tolerance, and power efficiency in modern processors.

Martin Dimitrov received the BS degree from Bethune-Cookman University and the MS degree from University of Central Florida, both in computer science. He is a doctoral student at the University of Central Florida. His research interests include high-performance and reliable microarchitectures.

Huiyang Zhou (S '98, M '03) received the bachelor's degree in electrical engineering from Xian Jiaotong University, China, in 1992 and the PhD degree in computer engineering from North Carolina State University in 2003. He is currently an assistant professor in the School of Electrical Engineering and Computer Science, University of Central Florida. His research focuses on high-performance microarchitecture, low-power design, architecture support for system reliability, and backend compiler optimization. He is a member of the IEEE Computer Society.