ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/285186100

A small and power efficient checkpoint core architecture for manycore processors

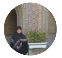
Article · January 2015

DOI: 10.1504/IJHPSA.2015.072852

CITATION
1

READS
89

2 authors:



Mageda Sharafeddin
Lebanese University

17 PUBLICATIONS **105** CITATIONS

SEE PROFILE

Haitham Akkary
American University of Beirut

63 PUBLICATIONS **1,357** CITATIONS

SEE PROFILE

All content following this page was uploaded by Mageda Sharafeddin on 14 December 2016.

The user has requested enhancement of the downloaded file.

A Small and Power Efficient Checkpoint Core Architecture for Manycore Processors

Abstract: This article describes and evaluates a small, out-of-order, simultaneous multithreaded (SMT) core architecture suitable for power constrained microprocessors, such as manycore microprocessors for high performance computing. The architecture does not require a reorder buffer (ROB) or physical registers for register renaming and instruction retirement. Instead, it uses a large number of virtual register IDs for register renaming, and a logical register file with multiple contexts. The architecture improves total thread execution throughput using two register contexts to support SMT execution of parallel workloads. Moreover, the architecture improves instruction level parallelism (ILP) and execution performance when running single-thread applications. In addition to eliminating the reorder buffer and the physical renaming register file, the architecture minimizes the logical register file hardware by using the two SMT register contexts and in-cell register file context fusion mechanism for recovering from branch mispredictions. We present results from Spec2006 benchmarks running on a SimpleScalar performance simulator of our architecture. Our simulation measurements show 5% single-thread performance improvement and 9.6% 2-thread SMT performance improvement over a conventional SMT core architecture with reorder buffer.

Keywords: Checkpoint core architectures; Out-of-Order processors, Virtual register renaming; Simultaneous multithreading

1 Introduction

Manycore microprocessors that target high performance computing pose new challenges as well as create new innovation opportunities for microprocessor architects. The energy constraints of these devices require architects to think differently about performance/power optimizations than on large conventional superscalar processor cores, including those that feature simultaneous multithreading (SMT), an architectural technique proposed by Tullsen (1995). New ideas that are unsuitable for large, superscalar cores may be appropriate and useful for this new, power constrained microprocessor market segment. This work focuses on evaluating new out-of-order (OOO) execution algorithm specifically targeting small, power constrained cores, such as those used in Intel Xeon manycore microprocessors as discussed in the work of Sodani (2015).

In conventional superscalar processors, a large physical register file is necessary for exposing large amount of instruction level parallelism as explained by Smith (1995). By allocating a separate physical register for each instruction that writes a logical register, a feature known as register renaming, write-after-write and write-after-read register dependences are eliminated. This allows the processor to execute instructions in any order, limited only by true data dependences and readiness of input operands. Physical registers are either organized as data fields within an instructions reorder buffer (ROB), or as a separate physical register file of size larger than the number of logical registers, as is the case on the Pentium 4 introduced by Hinton (2001).

Regardless of whether the physical registers used for register renaming are implemented as data registers within the reorder buffer (ROB) or as separate physical register file, the ROB in conventional superscalar processors provides key mechanisms for maintaining correct sequential program execution as explained by Smith (1985). Even though register renaming allows the processor core to schedule instructions for execution out of program order, the ROB reorders the execution results, thus updating (i.e. committing or retiring) register and memory state in the original program order, as needed in case of an interrupt or mispredicted branch event. In addition to the in-order retirement mechanism, the ROB provides the mechanisms necessary to reclaim dead physical registers that have been read by all instructions that need their values, and for restoring the correct non-speculative register renaming map table to resume execution properly upon recovery from a branch misprediction event. Although the ROB has become a central structure in superscalar processors, Akkary (2003, 2004) have shown that in-order retirement of instructions and the associated ROB mechanisms limit performance, especially as the size of the ROB increases to exploit more instruction level parallelism. As a higher performance alternative to ROB, Checkpoint Processing and Recovery architectures (CPR) were proposed for building scalable large instruction window processors. CPR replaces in-order retirement and its associated branch recovery mechanisms with checkpoint recovery and bulk retirement of instructions. CPR also decouples physical register reclamation from retirement by using read counters and remapped flags to identify and reclaim unneeded physical registers Moudgill (1993) .

Despite the advantages that CPR provides by eliminating the ROB and its sequential mechanisms, it introduces its own unique complexities. The CPR architecture studied in Akkary (2003, 2004) requires too

many checkpoints and too many counters. Specifically, CPR used eight mapping table checkpoints with flash copy support, eight instruction execution counters to manage bulk commit and checkpoint allocation and reclamation, register read counters and remapped flags, one per physical register, augmented with a free list array to track physical registers usage and reclamation. The complexity of CPR was justified as a more efficient alternative to ROB for scaling the instruction window on high performance superscalar cores. However, its complexity makes CPR unsuitable for energy-efficient small core architectures that target manycore microprocessors.

In this work, we evaluate a checkpoint architecture that uses virtual register renaming (VRR) introduced by Sharafeddine (2013). VRR targets small, low power cores similar in configuration to cores currently used in power constrained computing devices or in many core microprocessors (e.g. 2-wide superscalar, dual thread SMT, small ROB, load and store buffers, etc....). The virtual registers in VRR are not associated with any hardware storage locations such as physical registers or ROB entries. Instead, they are associated with unique names or IDs supplied by a hardware counter. Therefore, the number of virtual registers could be as large as needed for high performance without incurring a large hardware or energy cost.

The VRR architecture we evaluate in this work features a specialized logical register file, with two contexts. The two register contexts support dual thread execution in SMT mode. When executing a single thread application, the two register contexts are used as checkpoints for recovering from branch mispredictions and the core works as a CPR core. VRR not only eliminates the ROB of conventional OOO cores, it also eliminates CPR physical register file with its complex free list, register reclamation counters and remapped flags circuits. Moreover, VRR uses a flash context fusion operation in the specialized logical register file to reclaim checkpoints out of order. This increases the checkpoint hardware utilization significantly and reduces the number of checkpoints needed for performance in comparison to CPR.

This article makes the following contributions to improve instruction level parallelism (ILP) and thread level parallelism (TLP) on small out-of-order superscalar cores:

- It describes how the VRR architecture can be enhanced to support SMT execution without increasing the number of register contexts or the size of the architected register file, thus maximizing hardware utilization and consequently energy efficiency. The VRR based SMT architecture uses two register contexts as checkpoints when running in single thread mode to minimize the impact of branch mispredictions on performance. On the other hand, it uses the same two register contexts to improve parallel applications throughput in SMT mode by running two threads simultaneously.
- It present results on dynamic logic activity in key blocks of the VRR core and compare to a conventional ROB core. These results show that the VRR architecture to be very effective in minimizing dynamic energy consumed in the core, thus allowing the architecture to improve core ILP and TLP performance without increase dynamic energy demands of the core. This makes our VRR core an interesting option for future power constrained many core microprocessors.

The rest of the article is organized as follows. Section 2 describes the baseline virtual register renaming core architecture (VRR) when configured in single thread execution mode. Section 3 describes the SMT extensions to the baseline VRR core. In section 4, we discuss performance and energy results for SMT VRR execution. Section 5 examines related work. The article finally concludes in section 6.

2 Background

2.1 Single Thread Execution on the Virtual Register Renaming Core

Figure 1 shows a block diagram of the VRR core architecture presented by Sharafeddine (2013). An instruction in VRR goes through the typical processing steps of a conventional superscalar processor. These instruction processing steps include fetch, decode, rename, register operands read, schedule, execute, writeback and retire. VRR uses algorithm by Tomasulo (1967) and performs equivalent processing steps to Intel P6 architecture presented by Papworth (1996) but has key distinguishing features and differences, as shown in Figure 1. First, VRR performs register renaming using virtual register IDs generated by a counter. These IDs are not mapped to any fixed storage locations in the core. Second, VRR does not have a ROB or physical register file. Instead, it uses a logical register file cell with two fully ported register contexts. Third, VRR, like other CPR architectures, performs bulk commit of instructions using register checkpoints and checkpoint counters, and handles mispredicted branches and exceptions by rolling back execution to the last safe checkpoint.

Like in conventional superscalar architectures, the mapping table has the same number of entries as the number of logical registers defined by the instruction set architecture (ISA). Therefore, each entry in the table contains at any given time the VID of the last renamed instruction in the program that has as register destination the entry's corresponding logical register. Since the virtual ID counter is finite in size and cannot be allowed to overflow for correctness reason, VRR opportunistically resets the VID counter whenever it

can, e.g. when the pipeline is flushed to recover from a mispredicted branch. When such opportunity does not arise before the counter reaches its maximum value, VRR forces a pipeline flush. We have used a 10 bit VID counter in our simulations without seeing noticeable performance degradation from these forced pipeline flushes. Any buffer entries needed by an instruction (e.g., reservation station, load queue, store queue) are allocated in the rename stage. If any needed buffer is full, the pipeline stalls until an entry becomes available.

A group of contiguous instructions are committed during bulk commit stage. VRR tracks execution of instructions within a group using counters. When every instruction in a group executes (possibly out of order) without encountering an exception or mispredicted branch, VRR uses a flash copy (shift operation within the register file) to commit all the result registers instantaneously.

2.2 VRR Register File

The logical register file is the central component in the VRR architecture. It replaces the ROB and the physical register file of conventional superscalar cores.

VRR register file cell has two register contexts and in-cell flash copy logic to support taking a checkpoint, restoring a checkpoint, and fusing the two contexts. Figure 2 shows the block diagram of the register file. The shaded checkpoint box marks the unused in-cell flash copy logic in single thread mode. In single thread mode, both contexts and one checkpoint are used, while in SMT mode which we discuss in section 3, two threads are supported and each one uses one context and its corresponding checkpoint. The two context bits are fully connected to the register file read and write ports, while the checkpoint latches are not connected to the external register file read and write ports. The read and write ports are common to the two context bits. An instruction can read or write one of the two context bits based on operand context ID assigned at rename. Finally, there are two copies of the logical register mapping table in the renaming block, one for each context. In one cycle, Context0 (C0) register mappings can be flash copied into Context1 (C1) mapping table.

Table 1 shows a state machine that represents the state and state transitions of VRR register file and the actions taken on these transitions. We next discuss these events and the corresponding actions that are taken.

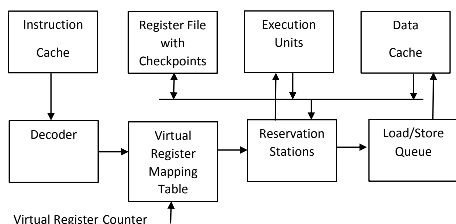


Figure 1 Block Diagram of Virtual Register Renaming Core Arcitecture.

2.3 Creating and Committing Register Checkpoints

Program execution starts in C0. Both C0 and its corresponding checkpoint initially contain the startup state of the program, as defined by the ISA. A checkpoint of the register state by definition corresponds to the precise execution state at some point in the program. The checkpoint bit (C0 Chkpt latch in Figure 2) always contains the last committed, precise register C0 state. VRR uses this checkpoint for handling exceptions and interrupts.

In our performance simulation model, VRR attempts to create a checkpoint every 16 instructions, if possible. When both contexts are busy, VRR keeps executing instructions beyond the checkpoint distance of 16 and then places the next checkpoint as soon as a context becomes free.

Notice that between the time C1 is spawned and the time C0 is committed, both contexts are active with their instructions executing concurrently in the pipeline. VRR identifies to which context an instruction belongs using a context ID assigned at the rename stage and carried with every instruction. Finally, stores from a context are issued to the data cache only after the context is committed.

2.4 Recovering from Mispredicted Branches

VRR uses the two contexts for recovering from branch mispredictions by simply discarding the context (or contexts) associated with the mispredicted branch and all instructions after the misprediction. In other words, if a mispredicted branch is in C0, all instructions belonging to both contexts are flushed and execution restarts from

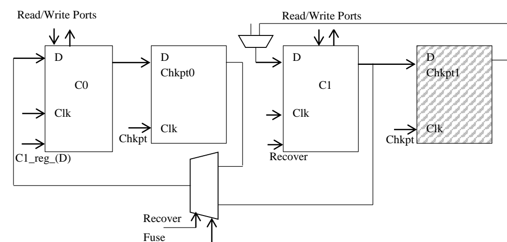


Figure 2 RF with Dedicated Backup Checkpoint

From State	To State	Input	Action
State 0	State 0	C0 mispredict	Restore
State 0	State 0	C0_inst cnt = 0	take checkpoint
State 0	State 1	Distance Reach low_conf_br	Copy Map Table C0 ->C1
State 1	State 0	C0 mispredict	restore
State 1	State 0	C0_inst _cntr = 0	take checkpoint and fuse
State 1	State 0	C1_inst _cntr = 0	Fuse only Not-Pending
State 1	State 1	C1 mispredict C1 mispredict	Copy Map Table C0 ->C1

Table 1 State Machine for VRR Microarchitecture .

the checkpoint. If a mispredicted branch is in C1, C1 instructions are flushed and execution restarts from the beginning of C1, after flash copying, again, C0 register mappings to C1 in the rename table.

Since execution restarts after a branch misprediction from the first instruction in the context to which the mispredicted branch belongs, some correctly executed instructions from before the branch are fetched and executed again. We call this rollback penalty. This rollback penalty is an additional overhead of branch mispredictions in checkpoint architectures that don't occur in conventional ROB superscalar cores. Finally, to ensure forward progress, a new checkpoint is taken immediately at the second instruction after execution restarts from a checkpoint. This ensures that at least the first instruction after rollback always executes and commits.

2.5 Recovering from Exceptions

After an exception, execution rolls back to the last committed checkpoint. VRR then switches to a special non-speculative in-order execution mode until the exception instruction is reached, leaving in the register file the precise state needed to take the exception. This mode can be implemented by allowing only one instruction at a time into the RS buffer using the allocate stage pipeline stall mechanism.

2.6 Context Fusion

It was necessary for CPR performance by Akkary (2003) to have checkpoints created as close as possible to mispredicted branches in order to reduce the amount of rollback execution on branch recovery. Previous CPR architectures used eight checkpoints, low confidence branch estimation and limited maximum distance between checkpoints to minimize performance loss from rollbacks. CPR reclaims checkpoints in program order through bulk commit of all instructions in the oldest checkpoint when they complete execution. Therefore, a long latency instruction such as cache load miss that goes to DRAM can stall checkpoint reclamation in CPR, thus creating the need for larger number of checkpoints.

In contrast, VRR uses a new effective mechanism to improve checkpoint utilization and reduce rollback overhead after branch mispredictions, thus minimizing the number of needed contexts without hurting performance. Like CPR, VRR bulk commits C0 (which is always the oldest context) into the checkpoint storage once all C0 instructions execute. Moreover, VRR reclaims C1 once it detects using a branch counter that all branches in C1 are correctly predicted, even if other instructions in C0 or C1 have not completed. We call this mechanism context fusion. VRR literally fuses C0 and C1 together into a new C0 with a larger number of instructions, leaving C1 available for a new checkpoint.

In section 2.2 Figure 2 we showed the context fusion logic within the register file cell. Similar logic also exists in the rename map table for the same purpose. C0 bit can be updated with an input from a 2-to-1 multiplexer. The multiplexer selects between the checkpoint storage bit in case of execution roll back to the checkpoint (e.g. for handling interrupts or exceptions), or C1 bit when fusing the contexts. However, there is an important difference between the two cases. The difference is in how the clock enable signal to C0 bit latch, labeled Ctxt1_reg_(D) where D stands for Dirty in Figure 2, is generated. When restoring the checkpoint, the clock enable signal Ctxt1_reg_(D) goes active for all registers resulting in copying all the checkpoint state into C0. When fusing the contexts, only registers that are destinations to instructions in C1 are copied into C0. In this case, the clock signal Ctxt1_reg_(D) goes active only for registers in C1 that become dirty, which is the case only for registers that are outputs of renamed instructions in C1. These instructions are the last writers of the register. On fusing the two contexts, VRR globally clears all context IDs in the RS, load queue and store queue entries, thus moving all these entries to C0.

3 Simultaneous Multithreading using Virtual Register Renaming

Simultaneous Multithreading (SMT) is an architectural feature used in many modern processors to improve execution throughput on superscalar cores. SMT introduced by Tullsen (1995) allows multiple threads to issue instructions in a single cycle. This increases the utilization of functional units in the superscalar processor. Additionally, running multiple threads can potentially hide stall latency of a thread by allowing the other threads to proceed, thus improving throughput. Most current superscalar processor cores support dual thread SMT.

A dual thread SMT makes a physical processor core appears to software as if it is two logical processors, capable of running concurrently two threads from a parallel application, or two different single thread applications. In order to allow two threads to run concurrently on the same physical core, the architecture register state and register renaming tables are duplicated. The rest of the hardware resources, such as caches, TLBs, execution units, etc are shared between the two threads, thus keeping the cost and energy per instruction of SMT minimal.

SMT increases the execution throughput of a superscalar core, usually represented as average number of instructions executed per clock cycle (IPC). Superscalar cores typically achieve only a fraction of their peak execution throughput when executing a single thread. This comes from limited instruction level parallelism due to data dependences, pipeline stalls resulting from cache misses, and pipeline flushes due to branch mispredictions. SMT threads increase the amount of parallelism available to the hardware, since instructions from different SMT threads are independent

and can execute concurrently. Moreover, when one thread stalls for any reason, instructions from other threads can still execute, keeping hardware resource utilization and execution throughput high.

We follow the full simultaneous issue model presented by Tullsen (1995). In our dual thread system this means that both threads compete for the issue slots each cycle. Priority to threads is assigned in a round robin fashion.

Since our VRR architecture uses two register contexts in single thread execution mode for checkpointing, implementing dual thread SMT on the VRR core is possible with a small modification in hardware as shown in Figure 2. Each of the two SMT threads could execute using one of the map table and register file contexts. We don't need to worry about branch mispredictions since we run non-speculatively in SMT. When the machine detects that two threads are running, it assigns one SMT register file context per thread. The execute stage of the pipeline detects a branch instruction and will block rename and issue of further thread instructions until the branch executes. In the rare cases of exceptions however, the machine rolls back to the checkpoint taken per thread to restore precise state of the machine. The checkpoint cell is a local bidirectional shift cell, marked as Chkpt in Figure 2. Notice that the checkpoint bit does not add capacitive load on the external register file bit-lines. By avoiding speculation completely, SMT on VRR achieves better throughput and consumes less power.

4 Results and Analysis

4.1 Simulation Methodology

We used Spec 2006 benchmarks and a detailed Simplescalar based performance simulator provided by Austin (simplescalar.com) to evaluate VRR relative to a conventional 2-wide, OOO baseline core that uses a ROB. We created two Simplescalar models: one that models the baseline SMT machine using ROB and another that models our proposed VRR-based SMT machine. Both models can load and run multiple threads simultaneously. Thread scheduling and resources are identical in both models. In terms of hardware the only difference is that ROB based SMT has a ROB and physical register file while the VRR based SMT has a logical register file. Table 2 shows the baseline core configuration that we have used, which represents a small core that is appropriate for power constrained computing devices. The VRR core model uses a logical register file with two contexts and two checkpoints as described earlier. Other than not having a ROB, the VRR core branch predictor, caches, pipeline and buffer configurations are identical to the baseline core configurations.

One has to be careful when comparing SMT to single thread throughput performance. This is because in SMT, threads with different execution characteristics could get different shares of execution resources. For example, a thread that misses the cache frequently will be completing less instructions than a thread that hits the cache most of the time. Execution cycles therefore are not distributed equally or fairly among SMT threads. As a result, a simulation sample that completes 100 million instructions on 2-thread SMT is not equivalent to a simulation workload of 50 million instructions from each thread.

In order to equalize the simulation workloads when comparing SMT throughput to single-thread execution throughput, we used the following methodology. In our experiments, we collect statistics from running the 2-thread SMT model for 100M cycles after skipping 100M instructions for cache and branch prediction warmup. We then extract from the SMT simulation the number of instructions from each thread completed during the initial warmup as well as during the next 100 million simulated instructions. These completed warmup and simulated instructions from each thread in SMT mode are what we use as warmup and simulation samples to measure performance in single-thread execution mode, except that we run them as two back to back simulations in single-thread mode versus concurrent simulation in SMT mode. Since we don't add any hardware features or resources to optimize SMT mode execution, the cycle time of the SMT architecture matches the cycle time of single thread execution mode. Comparing execution times of our equivalent simulation workloads in SMT and single-thread execution mode is therefore a fair measure of throughput performance. Moreover, since the cycle time and the simulated instructions are exactly the same in both SMT and single thread execution mode, average IPC is proportional to execution time and can be correctly used to compare throughput performance.

Table 3 shows an example of how we collected the performance metric presented in 4.2 using a combination of gobmk, thread 1, and perl, thread 2. In our diagrams in section 4.2 we refer to this as gobmk+perl. We set the simulation time to 100M cycles in SMT and record the number of useful instructions executed in each thread. We use these numbers to set the maximum useful instructions to be simulated in single thread mode. Notice that, since the cycle time and the simulated instructions are exactly the same in both SMT and single-thread execution mode, average IPC is proportional to execution. We collected performance data using representative simulation samples from Spec 2006 benchmarks, after skipping an initial execution phase to warm-up the branch predictor and the instruction and data caches. We used various two-thread

	Base Model	**Checkpoint with VRR**
Pipeline	2 wide, 13 Stages	2-wide 13 Stages
Reorder Buffer	80 Entries	None
Register File	80 physical registers for renaming 2 Retirement Register Contexts	2 Register Contexts + 2 Checkpoint Latches
Reservation Stations	32 Entries	32 Entries
Load/Store Queue	24 Entries	24 Entries
L1 Icache, L1 Dcache	16 KB, 4 way	16 KB, 4 way
L2 Cache	256KB, Unified	256KB Unified
Branch Prediction	Combined Bimod-gshare	Combined Bimod-gshare

Table 2 Simulated Machine Configuration.

combinations from 12 of SPEC 2006 integer and floating point benchmarks. The total hardware resources of the VRR and ROB SMT models are the same as the single thread baseline configuration shown in Table 2. However, these resources are shared between the two running SMT threads.

4.2 Performance Results

The earlier work by Sharafeddine (2013) showed that VRR in single thread execution mode using all 26 Spec2006 benchmarks achieves on average 4.8% improvement in performance over a conventional ROB-based baseline. In this article, we focus our performance analysis on the SMT execution mode.

A key issue that we have faced in our SMT VRR architecture is that in order to keep the register file as small as possible, we would like to exploit during single thread execution the free SMT register context and use it for checkpointing and branch recovery, thus increasing performance without increasing the size of the register file and its energy consumption. An important question then arises: how do we manage branch prediction and recovery from mispredicted branches in SMT mode with smaller number of checkpoints per thread and without significantly increasing rollback execution and consequently taking away useful execution cycles needed to exploit SMT thread level parallelism. Notice that this issue is exacerbated by the use of a small low power 2-wide core that has limited hardware resources and high utilization of these resources compared to typical large superscalar cores.

While inspecting execution traces in SMT mode, we noticed an interesting property of branch mispredictions and used it to reduce the risk of excessive rollbacks. We observed that on our 2 wide and 2 thread SMT core, most branches end up executing very shortly after entering the RS. Having two threads running simultaneously, it would make sense to simply disable speculative execution of instructions after a branch without hurting too much SMT performance, by simply stalling instructions at the rename stage until all branches ahead of them belonging to the same thread have been executed. Per thread branch up-down counter that is incremented when a branch enters the RS and decremented when a branch leaves can be used to generate the rename stage stall signal. Recovery from a mispredicted branch becomes a matter of detecting that a mispredicted branch has executed, allowing all instructions in the RS and load and store queues to drain out, and then flushing and restarting the front end of the pipeline from the corrected path.

SMT Mode	**Thread1 gobmk, Thread2 perl (gobmk+perl)**	**Single Thread Mode**
Retired Instructions in First Benchmark	gobmk_num_inst	gobmk_sim_cycle = Runs till gobmk_num_inst Retire
Retired Instructions in Second Benchmark	perl_num_inst	perl_sim_cycle = Runs till perl_num_inst Retire
Total IPC	(gobmk_num_inst + perl_num_inst)/100M	(gobmk_IPC/gobmk_sim_cycle) + (perl_IPC/perl_sim_cycle)

Table 3 Performance Metric Calculation for Single Thread and SMT.

Our conjecture that SMT performance would not be impacted significantly by this policy was based on the fact that since branches typically execute very quickly after entering the RS, actual execution stalls might not happen that frequently because of the availability of instructions from the other thread in the RS to keep the functional units busy. Moreover, since our core is a 2-wide superscalar core, there is a good probability of having enough thread level parallelism to keep functional units busy, even when one of the threads stalls for a few cycles to allow previous branches to execute and resolve. To test our conjecture, we compared the performance of two dual-thread SMT VRR machines: One that disables speculative execution beyond branches and uses one context per thread, and one that performs speculative execution and uses two contexts per thread. Figure 3 shows the results from this experiment.

Interestingly, there is actually 4% degradation in performance on the speculative SMT VRR machine compared to the non-speculative one, even though the speculative machine has a larger RF with 2 contexts per thread. This indicates that the rollback execution has a big impact on total throughput, as it takes away precious execution resources that could have been exploited by the thread level parallelism.

When repeating the same experiment for a ROB SMT machine of the baseline configuration, we observed that the ROB machine did not suffer as much performance degradation from the speculative execution (only about 0.5% on average). The reason is that the ROB SMT machine suffers from wasted execution cycles due to wrong path instructions after mispredicted branches, but does not suffer from rollback execution like VRR.

Using non-speculative execution in SMT mode as a baseline, we now show the performance benefits of SMT VRR. First, due to the large number of threads tested we only show a representative subset of results in Figure 4. The first bar is a measure of speedup in 2T non-speculative VRR compared to 2T non-speculative ROB, the second bar is a measure of relative speedup for thread 1 when in the 2T VRR machine compared to that for the same thread in the 2T ROB machine, and the third bar is a measure of the relative speedup of thread 2 in the 2T VRR machine compared to that in the 2T ROB machine. We chose to show all this data together to capture the fairness of VRR as well as its performance benefits. 2T SMT VRR core outperforms the 2T SMT ROB core by an average of 9.6%. This is a larger increase in performance than the 4.8% average speedup VRR gives over a ROB baseline when running in single thread execution mode as discussed by Sharafeddine (2013). The explanation is the following. Since the 2 SMT threads share the core hardware resources, it puts more pressure on the ROB. Consequently, eliminating the ROB as a bottleneck helps the VRR core achieve a larger improvement in performance over the baseline core when both run in SMT mode. Additionally, Figure 4

demonstrates the fairness of the VRR method compared to ROB in handling the individual threads. All thread1-%-speedup and thread2-%-speedup bars are above 1. This is an indication that the 2T VRR machine does not speedup one thread at the expense of the other thread. The only exception is when bzip2 runs with libquantum, it suffers degradation in performance when run in 2T VRR. This degradation is due to unfairness in the 2T ROB model. While our 2T VRR machine runs the two mentioned set of threads by maintaining close IPC of the two run threads, the 2T ROB machine is biased against libquantum. In the 2T ROB machine IPC for bzip2 is three times that of libquantum.

In Figure 5 we show results from running all 2-thread combinations on the VRR core in SMT mode, versus running the overall workload serially and back to back on the VRR core in single thread mode. Notice that we follow the methodology presented in Table 3. The improvement in total throughput performance is 27% in SMT mode.

A final closing note on the performance section is that the combination of threads presented here is representative of the results when running all possible combinations of benchmarks.

4.3 Logic Activity Analysis of VRR SMT Execution

We used the SimpleScalar performance model to compare logic activities of blocks that differ between the VRR and the ROB baseline machine. The logic activities are useful as indirect measure of the dynamic energy consumed by the two models.

First, we compare VRR SMT to single-thread SMT execution. In addition to having a performance advantage over single-thread VRR on threaded workloads, VRR SMT also has an advantage in dynamic energy required per executed instruction. As described earlier and due to the non-speculative execution in SMT mode, VRR SMT eliminates the rollback execution penalty due to mispredicted branches from which single thread VRR execution mode suffers.

Table 4 lists the branch misprediction rate and cost of rollback execution due to branch misprediction recovery for each Spec2006 benchmark. On average, there is about 5% increase in number of executed instructions due to rollback. Some of these benchmarks, such as Gobmk and Astar, suffer from high rollback penalty of 23% and 15% respectively. Since in our study we did not use branch confidence to improve checkpoint placement as suggested in previous CPR works, e.g. study by Jothi (2013), it may be possible in the future to reduce rollback by using a better branch predictor and low confidence branch estimation to select the checkpoints placement.

Second, we compare VRR SMT execution to ROB SMT execution. We focus on the blocks that are different in the two cores. These blocks consist of the following:

- The ROB core has a physical register file (PRF) to hold speculative as well as retired register values. In our simulated core model, this PRF consists of a total of 144 register entries, 80 entries for register renaming in addition to 64 retired registers entries that correspond to two register contexts for the two SMT threads. The VRR core does not have a physical register file.
- The ROB core has a reorder buffer that contains an entry for each renamed instruction that has not been committed/retired. Each entry contains a pointer to the physical register destination allocated in the PRF rename space for the instruction corresponding to this reorder buffer entry. The VRR core does not have a reorder buffer.

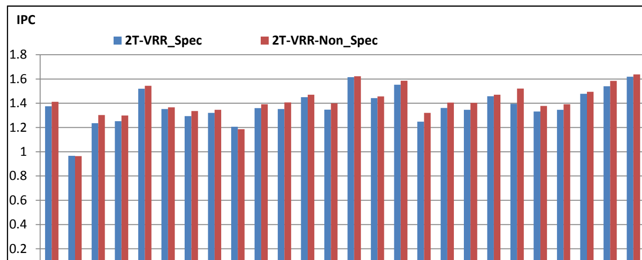


Figure 3 Non-Speculative 2T VRR with 1 Context per thread Vs. Speculative 2T VRR with 2 Contexts per Thread.%

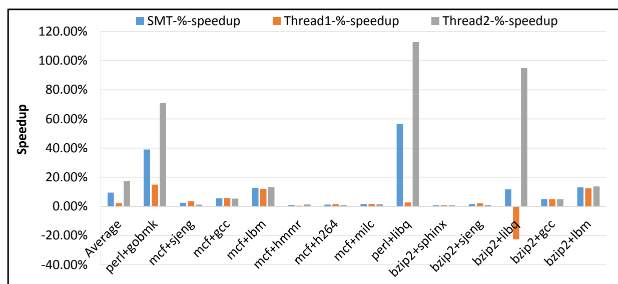


Figure 4 2T VRR Core Outperforms 2T Reorder Buffer Core by an Average of 9.6%

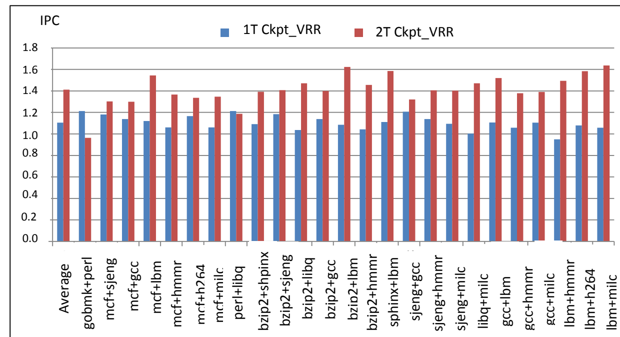


Figure 5 2T SMT Chkpt_VRR Improves Throughput by 27% over Single Thread Chkpt_VRR

- The 32-entry RS block in our simulated VRR core contains two source register operands in each entry. The ROB core does not have these register operands in the RS, since all register operands are kept in the PRF, from which they are read and written when needed. The ROB core, however, needs to store in each RS entry two pointers to the instruction source operands in the PRF.
- The VRR core has a retirement logical register file (RRF) with storage for 64 register operands corresponding to two register contexts for two SMT threads. The ROB core uses space in the PRF for the two retired register contexts and tracks the location of these retired registers with two sets of PRF pointers, one set for each register context. Clearly, the VRR does need to maintain such pointers since it does not use a PRF.

We ignore in our comparison the additional energy cost of pointers to register operands in the PRF that the ROB core needs (i.e. pointers in the ROB, RS and retired registers pointers). This clearly favors the ROB core. Nevertheless, our results show that in spite of this, the VRR core seems to have a slight advantage in energy consumption over the ROB core, as we discuss next.

Table 5 shows the number of reads and writes of register operands per instruction from the register files and from the RS array in the ROB SMT and VRR SMT core for various simulated workloads. Notice that in the ROB SMT core, every instruction reads two register operands from the PRF when dispatched to execution (Column 2), but does not read any register operands from the RS array (Column 6). On the other hand, every instruction reads two register operands from the RS in the VRR SMT machine when dispatched to execution (Column 7) and a fraction of instructions read register operands from the RRF when they enter the RS after renaming (Column 3). Even though there are more register operands per instruction that are read from the RS and RRF arrays in the VRR SMT core than the register operands per instruction that are read from the PRF in the ROB SMT machine, the PRF array in the ROB SMT machine is more than twice as large as either the RS or the RRF arrays in the VRR SMT machine (144 vs. 64). Since the increase in the capacitive load in the PRF is larger than the increase in the register reads activity in the VRR machine, the overall energy cost of reading register operands in the VRR SMT core is less than that in the ROB SMT core. A similar analysis of the overall energy cost of writing register operands leads to a similar conclusion that the VRR energy cost per instruction for writes into the register arrays is lower on the VRR machine. We therefore conclude our analysis by observing that the VRR SMT core not only performs better on multithreading workloads but also achieves this improvement in performance with a lower dynamic energy per instruction cost. We argue that this makes VRR an interesting core architecture option for power-constrained manycore microprocessors targeting high performance computing.

We have a bigger advantage in energy per instruction from VRR in SMT mode compared to single thread execution. This is because of the non-speculative execution of VRR SMT mode, which eliminates the rollback execution penalty due to mispredicted branches from which single thread VRR execution mode suffers.

Benchmark	# Br. Mispr/1000 inst	% Rollback Execution
Mcf	0.05	0.05%
Gobmk	38	22.73%
Milc	0.05	0.06%
Sjeng	11.6	8.67%
Perl	5	4.85%
Libquantum	2.7	10.08%
Bwaves	0.05	0.81%
Lbm	3.2	9.2%
Bzip2	0.03	0.3%
Games	4.8	8.89%
Zeusmp	0.3	1.61%
Gromacs	0.12	1.31%
Leslie	0.18	0.53%
Dealii	1	1.82%
Soplex	0.05	0.17%
Povary	5.3	7.61%
Calculix	0.07	0.14%
Hmmr	0.4	2.02%
H264	2.9	2.69%
Tonto	0.9	1.93%
Astar	1.25	14.53%
Wrf	4.3	11.28%
Sphinx	3.3	9.44%
Gcc	0.12	0.09%
Xalanc	2.3	7.08%
Namd	2.3	2.97%

Table 4 Speculative Execution Stats for Single Thread VRR Machine

Benchmark	RF_Read		RF_Write		RS_Read		RS_Write	
	ROB	VRR	ROB	VRR	ROB	VRR	ROB	VRR
Average	2.00	0.4	0.85	0.22	0.00	2.00	0.00	1.39
gobmk+perl	2.00	0.33	0.78	0.12	0.00	2.00	0.00	1.35
mcf+sjeng	2.00	0.33	0.82	0.25	0.00	2.00	0.00	1.40
mcf+gcc	2.00	0.31	0.80	0.26	0.00	2.00	0.00	1.34
mcf+lbm	2.00	0.44	0.84	0.22	0.00	2.00	0.00	1.35
mcf+hmmr	2.00	0.36	0.86	0.18	0.00	2.00	0.00	1.41
mcf+h264	2.00	0.33	0.83	0.23	0.00	2.00	0.00	1.40
mcf+milc	2.00	0.38	0.85	0.26	0.00	2.00	0.00	1.38
perl+libq	2.00	0.40	0.77	0.09	0.00	2.00	0.00	1.58
bzip2+sphinx	2.00	0.41	0.84	0.22	0.00	2.00	0.00	1.41
bzip2+sjeng	2.00	0.44	0.86	0.24	0.00	2.00	0.00	1.45
bzip2+libq	2.00	0.53	0.85	0.17	0.00	2.00	0.00	1.60
bzip2+gcc	2.00	0.41	0.84	0.24	0.00	2.00	0.00	1.40
bzip2+lbm	2.00	0.48	0.88	0.21	0.00	2.00	0.00	1.41
bzip2+hmmr	2.00	0.44	0.90	0.18	0.00	2.00	0.00	1.46
sphinx+lbm	2.00	0.42	0.84	0.24	0.00	2.00	0.00	1.29
sjeng+gcc	2.00	0.34	0.82	0.30	0.00	2.00	0.00	1.33
sjeng+hmmr	2.00	0.39	0.87	0.21	0.00	2.00	0.00	1.40
sjeng+milc	2.00	0.39	0.87	0.28	0.00	2.00	0.00	1.37
libq+milc	2.00	0.48	0.86	0.21	0.00	2.00	0.00	1.03
gcc+lbm	2.00	0.41	0.84	0.23	0.00	2.00	0.00	1.29
gcc+hmmr	2.00	0.36	0.85	0.22	0.00	2.00	0.00	1.34
gcc+milc	2.00	0.37	0.90	0.29	0.00	2.00	0.00	1.31
lbm+hmmr	2.00	0.43	0.89	0.15	0.00	2.00	0.00	1.36
lbm+h264	2.00	0.46	0.86	0.23	0.00	2.00	0.00	1.35
lbm+milc	2.00	0.43	0.89	0.19	0.00	2.00	0.00	1.33

Table 5 Reads and Writes per Instruction of Various Blocks in the ROB SMT Core

5 Related Work

A study by Tomasulo (1967) proposed an algorithm for eliminating write-after-write and write-after-read register dependences and executing instructions out of order using reservation stations. Tomasulos algorithm did not provide precise state for handling exceptions. In addition, it did not perform speculative execution and therefore did not require precise state for recovering from branch mispredictions.

Checkpoints in processors are used to repair architecture state to a known, precise previous state. The use of checkpoints for recovering from branch mispredictions and exceptions in OOO processors was first proposed by Hwu (1987). The Pentium 4 by Hinton (2001) used a retirement register alias table to track register mappings, while the MIPS R10000 introduced by Yeager (1996) and Alpha 21264 by Leibholz (1997) used check-points to recover register rename mappings. All the above designs used physical registers for register renaming. Architectures that use checkpoints with physical register files for recovery and for scaling the instruction execution window to help tolerate cache misses include Virtual ROBs introduced by Cristal (2002), Cherry introduced by Martinez (2002), Checkpoint Processing and Recovery introduced by Akkary (2003, 2004), Continual Flow Pipelines by Hilton (2010); Jothi (2011, 2013); Akkary (2004), Out of Order Commit Processors by Cristal (2004), and further examined by Safi (2007).

Gonzalez (1998) proposed using virtual registers to shorten the lifetime of physical registers. Virtual-physical registers are used to delay the allocation of physical registers. Instead of allocating a physical register when instructions are renamed, they are allocated when instructions execute and produce results that need physical storage. Hence, virtual registers are used for register renaming. Kilo instruction processors by Cristal (2004) also used virtual renaming and ephemeral registers to do late allocation of physical registers. In contrast to virtual-physical registers and ephemeral registers, VRR does not require physical registers for allocation of execution results to physical registers.

Safi (2007) show that in a 130nm technology with a checkpoint architecture, the latency of register alias table with an organization similar to our in-cell shift checkpoint cell have little impact on latency, energy, and area of an architecture with less than four checkpoints. Their study shows a linear increase in energy with checkpoints. With 16 checkpoints latency increases by 8.2%, energy by 11% for read and 31% for write, and area increases by 31%. Simultaneous multithreading (SMT) was first proposed in Tullsen (1995), which showed better resource utilization and improved throughput with SMT. Jeng (2000) examined the effect of speculation on power efficiency and proposed a power threshold driven execution to switch speculation off. They show improved performance and power savings even without speculation. The study by Hily (1999) argues that OOO execution may not be cost effective in SMT machines.

Researchers have been looking for microarchitectures capable of maintaining performance of single threads compared to that of OOO machines and increasing throughput when more than one thread is running. Proposals such as Core Fusion by Ipek (2007) and Federation by Boyer (2010) were introduced to address both ILP and TLP. Core Fusion is a symmetric Chip Multi Processor (CMP) which consists of small OOO cores that can be grouped to handle single threads. Federation is a CMP consisting of small in-order cores that can handle multiple threads and improve throughput. The in-order cores can be grouped to become an OOO core for better single thread performance. Morphcore by Khubaib (2012) examines a hybrid architecture that can improve single thread performance and multithread workloads. It shows improved performance compared to OOO machines using medium and small size cores. MorphCore is a flexible substrate for better performance in single thread and multithread scenarios. VRR is orthogonal to MorphCore and can be used in such systems.

6 Conclusion

This article describes and evaluates a small, checkpoint, out-of-order SMT core architecture suitable for power constrained manycore microprocessors used for high performance computing. The architecture does not require a ROB or physical registers for register renaming and instruction retirement. Instead, it uses a large number of virtual register IDs for register renaming, and a logical register file with multiple contexts. The architecture uses the register contexts to support SMT execution for improved throughput performance of parallel applications. When running single thread applications, the architecture uses the register contexts and in-cell register file context fusion mechanism for recovering from branch mispredictions. The architecture improves single thread performance by 5% and 2-thread SMT performance by 9.6% over a conventional SMT core architecture with ROB. We believe that VRR energy efficiency in single thread mode and SMT mode added to the improved performance over the baseline ROB, make VRR an interesting architecture for many core microprocessors.

References

Akkary H., Rajwar, R. and Srinivasan, S.T. (2003), 'Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors', *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*, pp. 423–434.

Akkary, H., Rajwar, R. and Srinivasan, S.T. (2003), 'Checkpoint Processing and Recovery: an Efficient, Scalable Alternative to Reorder Buffers', *MICRO 36 Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, Vol. 23, No. 6, pp. 11–19.

Akkary, H., Rajwar, R. and Srinivasan, S.T. (2004), 'An Analysis of a Resource Efficient Checkpoint Architecture', *ACM Transactions on Architecture and Code Optimization*, pp. 418–444.

Austin, T. (2011), 'SimpleScalar', http://www.simplescalar.com.

Boyer, M., Tarjan, D. and Skadron, K. (2010), 'Federation: Boosting Per-thread Performance of Throughput-oriented Manycore Architectures', *Transactions on Architecture and Code Optimization (TACO)*, Vol. 7, No. 4.

Cristal, A., Valero, M., Llosa, J. and Gonzalez, A. (2002), 'Large Virtual ROBs by Processor Checkpointing', *Technical Report, UPC-DAC-2002-39 Department of Computer Science, Barcelona, Spain.*

Cristal, A., Ortega, D., Llosa, J. and Valero, M. (2004), 'Out-of-order Commit Processors', *Processdings of High Performance Computer Architecture*, pp. 175–184.

Cristal, A., Santana, O.J., Valero, M. and Martinez, J.F. (2004), 'Toward Kilo-instruction Processors', *ACM Transactions on Architecture and Code Optimization (TACO)*, Vol. 1, No. 4, pp. 389–417.

Gonzalez, A., Gonzalez, J., and Valero, M. (1998), 'Virtual-physical Registers', *High Performance Computer Architecture*, pp. 175–184.

Gunther, S.H., Binns, F., Carmean, D.M. and Hall, J.C. (2001), 'Managing the impact of increasing microprocessor power consumption', *Intel Technology Journal*, Vol. 5, No. 1.

Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A. and Roussel, P. (2001), 'The Microarchitecture of the Pentium 4 Processor', *Intel Technology Journal*, Vol. 5, No. 4.

Hilton, A. and Roth, A. (2010), 'BOLT: Energy-efficient Out-of-order Latency Tolerant Execution', *Proceedings of High Performance Computer Architecture*, pp. 1–12.

Hily, S. and Seznec, A. (1999), 'Out-of-order Execution May not be Cost Effective on Processors Featuring Simultaneous Multithreading', *Fifth International Symposium on High Performance Computer Architecture.*

Hwu, W. and Patt, Y.N. (1987), 'Checkpoint Repair for Out-of-order Execution Machines', *14th Annual International Symposium on Computer Architecture*, pp. 18–26.

Ipek, E., Kirman, M., Kirman, N., and Martinez, J. (2007), 'Core Fusion: Accommodating Software Diversity in Chip Multiprocessors', *Proceedings of the 34th Annual International Symposium on Computer Architecture*, Vol. 82, No. 2, pp. 186–197.

Jacobsen, E., Rotenberg, E. and Smith, J.E. (1996), 'Assigning Confidence to Conditional Branch Predictions', *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 142–152.

Jeng, S., Tullsen, D. and Cai, J. (2000), 'Power-Sensitive Multithreaded Architecture', *IEEE International Conference on Computer Design*, pp. 199–206.

Jeng, S., Tullsen, D. and Cai, J. (2012), 'Retrospective on Power-Sensitive Multithreaded Architecture', *IEEE International Conference on Computer Design*, pp. 15–16.

Jothi, K., Sharafeddine, M., and Akkary, A. (2011), 'Simultaneous Continual Flow Pipeline Architecture', *IEEE 29th International Conference on Computer Design*, pp. 9–12.

Jothi, K. and Akkary, H. (2013), 'Tuning the Contiual Flow Pipeline Architecture', *International Conference on Supercomputing (ICS)*, pp. 142–152.

Jothi, K. and Akkary, A. (2013), 'Streamlining the continual flow processor architecture with fast replay loop', *EUROCON*, pp. 1821–1828.

Khubaib, Suleman, M.A., Hashemi, M., Wilkerson, C. and Patt, Y.N. (2012), 'MorphCore: an Energy-efficient Microarchitecture for High Performance ILP and High Throughput TLP', *45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 305–316.

Leibholz, D. and Razdan, R. (1997), 'The Alpha 21264: a 500 MHz Out-of-order Execution Microprocessor', *Proceedings of the 42nd IEEE Computer Society International Conference (COMPCON)*, pp. 28–36.

Martinez, J.F., Renau, J., Huang, M.C., Prvulovic, M. and Torrellas, J. (2002), 'Cherry: Checkpoint Early Resource Recycling in Out-of-order', *Proceedings 35th Annual IEEE/ACM International Symposium on Microprocessors(MICRO-35)*, pp. 3–14.

Moudgill, M., Pingali, K. and Vassiliadis, S. (1993), 'Register Renaming and Dynamic Speculation: an Alternative Approach', *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pp. 202–213.

Papworth, D.B. (1996), 'Tuning the Pentium Pro Microarchitecture', *IEEE Micro*, Vol. 16, No. 2, pp. 8–15.

Safi, E., Akl, P., Moshovos, A., Veneris, A. and Arapoyianni, A. (2007), 'On the Latency, Energy, and Area of Checkpointed, Superscalar Register Alias Tables', *The International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 379–382.

Sharafeddine, M., Akkary, H. and Carmean, D. (2013), 'Virtual Register Renaming', *26th International Conference on Architecture of Computing Systems*, pp. 43–48.

Smith, J.E. and Pleszkun, A.R. (1985), 'Implementation of Precise Interrupts in Pipelined Processors', *12nd Annual International Symposium on Computer Architecture*, pp. 36–44.

Smith, J.E. and Sohi, G.S. (1995), 'The Microarchitecture of Superscalar Processors', *Proceedings of the IEEE*, Vol. 83, No. 12, pp. 1609–1624.

Sodani, A. (2015), 'Knights Landing: 2nd Generation Intel Xeon Phi Processor', To appear in August issue of *Proceedings of Hot Chips: A Symposium on High Performance Chips*.

Tomasulo, R.M. (1967), 'An Efficient Algorithm for Exploiting Multiple Arithmetic Units', *IBM Journal of Research and Development*, Vol. 11, pp. 25–33.

Tullsen, D., Eggers, S.J. and Levy, H.M. (1995), 'Simultaneous Multithreading: Maximizing on-chip Parallelism', *22nd Annual International Symposium on Computer Architecture*, pp. 392–403.

Yeager, K (1996), 'The MIPS R10000 Superscalar Microprocessor', *IEEE Micro*, Vol. 16, No. 2, pp. 28–40.

View publication stats