Idempotent Processor Architecture

Marc de Kruijf Karthikeyan Sankaralingam
Vertical Research Group
University of Wisconsin – Madison
{dekruijf, karu}@cs.wisc.edu

ABSTRACT

Improving architectural energy efficiency is important to address diminishing energy efficiency gains from technology scaling. At the same time, limiting hardware complexity is also important. This paper presents a new processor architecture, the *idempotent processor architecture*, that advances both of these directions by presenting a new execution paradigm that allows speculative execution without the need for hardware checkpoints to recover from misspeculation, instead using only re-execution to recover. Idempotent processors execute programs as a sequence of compiler-constructed idempotent (re-executable) regions. The nature of these regions allows precise state to be reproduced by re-execution, obviating the need for hardware recovery support. We build upon the insight that programs naturally decompose into a series of idempotent regions and that these regions can be large. The paradigm of executing idempotent regions, which we call idempotent processing, can be used to support various types of speculation, including branch prediction, dependence prediction, or execution in the presence of hardware faults or exceptions.

In this paper, we demonstrate how idempotent processing simplifies the design of in-order processors. Conventional in-order processors suffer from significant complexities to achieve high performance while supporting the execution of variable latency instructions and enforcing precise exceptions. Idempotent processing eliminates much of these complexities and the resulting inefficiencies by allowing instructions to retire out of order with support for re-execution when necessary to recover precise state. Across a diverse set of benchmark suites, our quantitative results show that we obtain a geometric mean performance increase of 4.4% (up to 25% and beyond) while maintaining an overall reduction in power and hardware complexity.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Hardware/Software Interfaces*

MICRO '11, December 3-7, 2011, Porto Alegre, Brazil

General Terms

Design, Performance, Reliability

1. INTRODUCTION

Emerging challenges in technology scaling present diminishing opportunity to improve processor energy efficiency at the transistor level: while the doubling of transistors every generation is expected to continue, the historically commensurate increase in their energy efficiency is not [3, 16]. As a result, the task of improving processor energy efficiency is increasingly falling to computer architects. At the same time, mobile computers are proliferating at an explosive rate [41, 20]. For mobile processors, energy efficiency is important due to both limited battery life and limited opportunity to manage heat dissipation through fans or heatsinks.

While specialized and accelerator architectures tackle the energy efficiency problem by augmenting conventional processors [11, 18, 22, 24, 40], this paper investigates whether fundamental inefficiencies in the processor core itself can be eliminated. In particular, we consider the overheads arising from speculative execution, including branch prediction, memory dependence prediction, and execution in the presence of hardware faults or exceptions. While most processors use some kind of hardware buffer or checkpoint to store speculative state until it is safe to commit to architecture state, these hardware resources and their interaction with the processor pipeline introduces significant complexity and energy overheads [4, 26, 37, 38].

In this paper, we observe that exploiting the mathematical property of idempotence in programs allows the benefits of both speculative execution and precise state to be achieved with minimal power or complexity overhead in the hardware. In particular, we show how programs can be constructed in a special way, such that execution is always restartable over some interval of instructions, regardless of the order in which those instructions issue or retire. In the event that precise state is needed for some specific instruction inside an interval, the processor can jump back to the beginning of the interval and re-execute precisely up to the point of the specific instruction and no further. We call a processor using this execution model an idempotent processor. While an idempotent processor recovers similarly to a speculative processor that uses hardware checkpoints [4, 13, 26, 34], it is software co-designed to incur none of the associated hardware power and complexity overheads.

We develop as a concrete example the case of exception support in modern processors. We observe that in-order instruction retirement—a feature widely assumed necessary in

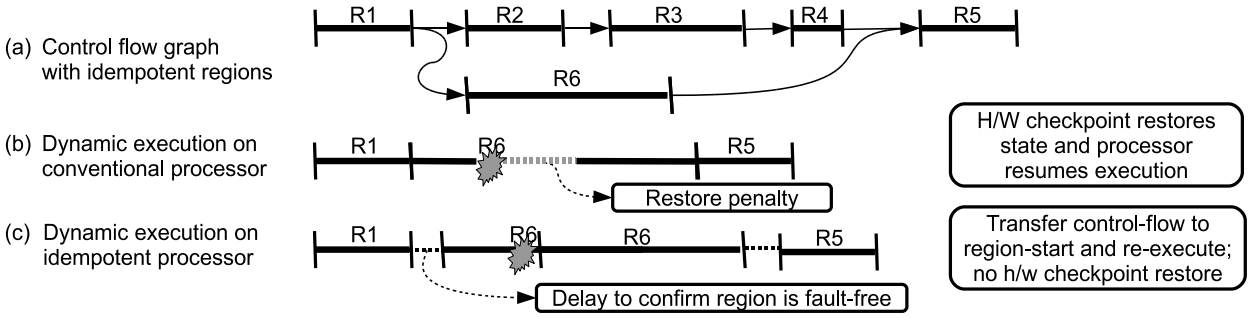


Figure 1: Idempotent processing overview.

general-purpose processors to ensure that processor state is consistent with respect to a running program at all times—introduces substantial complexity and inefficiency in the processor design. In-order retirement simplifies program debugging and enables seamless support of page faults and other exceptions in software. However, without any hardware support, it also hinders performance by preventing the out-of-order completion and/or issue of independent instructions. Hence, modern processors typically employ one or more special-purpose hardware structures, such as a reorder buffer or speculative register file, to manage the bookkeeping of in-order retirement state while attempting to maximize performance [26, 37]. Unfortunately, the resulting hardware complexity and power consumption can be high. In this paper, we develop an idempotent processor that can issue and retire instructions out of order to achieve better performance than a modern in-order processor with no overall increase in power or hardware complexity.

Overall, we make the following contributions:

- We develop idempotence as a means to recover from speculative execution in general and out-of-order retirement specifically (Section 2).
- We describe the process of identifying idempotent regions that a compiler can use to create idempotent binaries (Section 3).
- We present the design of an idempotent processor that retires out of order and uses idempotent regions to implement exception recovery, demonstrating power and complexity savings over a typical in-order processor (Section 5 with background in Section 4).
- Our results show that our idempotent processor commonly achieves 4.4% better performance over an energy-efficient baseline with reduction in both power and hardware complexity. Performance is also within 24.2% of a full-fledged out-of-order processor (Section 6).

Section 7 discusses some of the broader implications and subtleties of idempotence-based recovery. Section 8 presents related work. Finally, Section 9 concludes.

2. IDEMPOTENT PROCESSING OVERVIEW

Figure 1 contrasts the execution approach of a conventional processor with that of an idempotent processor. A conventional processor checkpoints or buffers state at various points as instructions execute through the pipeline. The idempotent processor, on the other hand, executes program binaries that are demarcated into idempotent regions as shown in Figure 1(a). The property of idempotence guarantees that any region can be freely re-executed, even after partial execution, and still produce the same result.

Figure 1(b) illustrates recovery from mis-speculation in a conventional processor, where a mis-speculation could be a branch misprediction, a hardware transient fault, or an exception. As region R6 executes, a mis-speculation is detected after some time. To correct the mis-speculation, a checkpoint is restored; execution then resumes and completes. Figure 1(c) contrasts this recovery behavior with that of an idempotent processor. Again, region R6 executes with a mis-speculation occurring after some time. However, the idempotent processor recovers by simply jumping back to the beginning of the region. To guarantee successful recovery, the idempotent processor must ensure that the side-effects of a mis-speculation are appropriately contained, and also that execution does not proceed beyond the end of region until that region is verified to be free of mis-speculation.

In this paper, we demonstrate how idempotent processors are able to retire out of order while maintaining the ability to reproduce precise state in the event of an exception by re-executing, leading to better overall efficiency. Although we focus on exception recovery, idempotence can be used to recover from other forms of mis-speculation as well, such as branch mispredictions or hardware faults, as previously mentioned. These topics are not developed further in this paper, but are promising extensions for future work.

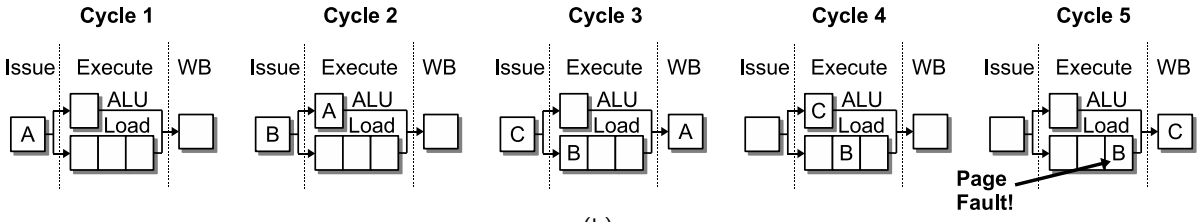
2.1 Idempotence and out-of-order retirement

Figure 2 illustrates how idempotence enables precise state to be achieved despite out-of-order retirement for a simple sequence of three instructions. It assumes a simple single-issue processor that issues instructions in program order but retires them potentially out of order. Figure 2(a) shows the instruction sequence along with the issue cycle, execute latency, and writeback cycle of each instruction. Figure 2(b) shows the cycle-by-cycle state of the processor pipeline.

In the example, a page fault occurs during the execution of instruction B in cycle 5. Because instruction C retires in that same cycle, normally speaking a processor is unable to cleanly resume execution after handling the page fault because the program state at the end of cycle 5 is not consistent with respect to the start of either instruction B or C. However, observe that the program state *is* consistent with respect to the start of instruction A; after servicing the page fault, it is possible to resume execution from instruction A.

Instruction	Issue Cycle	Execute Latency	WB Cycle
A. R2 ← add R0, R1	1	1	3
B. R3 ← ld [R2 + 4]	2	3	6
C. R2 ← add R2, R4	3	1	5

(a)



(b)

Figure 2: Out-of-order retirement over a simple instruction sequence.

Alternatively, the processor can first issue and execute from instruction A precisely to instruction B, service the page fault, and then resume execution from instruction C.

An intelligent compiler can construct programs in this way, producing regions of code over which instruction retirement does not affect the state of the program with respect to the start of the region; from any point within the region, execution of the region can be abandoned and retried at any time. We call these regions *idempotent regions* because they exhibit the property of idempotence—they can be executed multiple times, and the effect is as if they are executed only a single time. In effect, the beginning of a region marks an implicit checkpoint in the program's execution that can be used for program recovery. In Section 3 of this paper, we show how idempotent regions can be constructed, and that they can be large—sufficiently large that there is potential to exploit this property in architecture design.

2.2 An example idempotent processor design

Within an idempotent region, an idempotent processor may execute and retire instructions out of order. This enables three key simplification opportunities in the processor design. First, it allows the results of low latency operations to be used immediately after they are produced, without the need for complex staging and bypassing as in conventional processors. Second, it simplifies the implementation of precise exception support, particularly for long latency operations such as floating point operations. Finally, it enables instructions to retire out of order with respect to instructions with unpredictable latencies, such as loads. Figure 3 compares the resulting execution behavior of an idempotent processor to that of an in-order and out-of-order processor. While the in-order and out-of-order processor both stage and bypass instruction results until they are able to retire in order, the idempotent processor does not.

Table 1 summarizes our comparison by considering the three basic steps of instruction execution. It contrasts the responsibilities of the hardware and the compiler in conventional processors and an idempotent processor. The last column highlights the fundamental challenge each processor type faces in scalability to higher performance: in-order processors are limited by their ability to extract ILP dynamically, out-of-order processors are limited by the super-linear power and complexity growth as instruction window

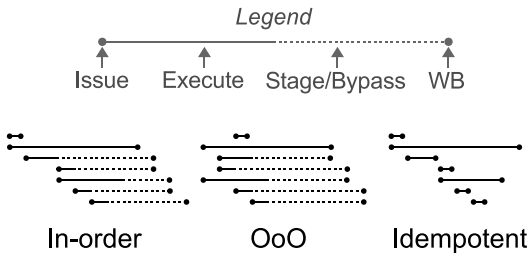


Figure 3: A comparison between idempotent and traditional processor designs.

size grows, and an idempotent processor, whose target performance is likely to exist somewhere between these two classifications, is limited by the size of the idempotent regions that can be efficiently extracted from applications.

In this paper, we explore idempotent processing considering the design space of dual-issue in-order processors in detail. In Section 4, we first investigate the complexities arising from in-order retirement in modern dual-issue in-order processors, giving supporting references to commercially available processor designs. We then show in Section 5 how out-of-order retirement enables power and complexity savings, and how these savings can be traded off for better performance by introducing modest out-of-order execution. Section 6 presents a quantitative evaluation.

Although our technique can also be applied to the design of out-of-order processors, their energy budget relative to in-order processors is typically higher to implement out-of-order issue, reducing the relative benefit. Additionally, these processors may re-use some of the retirement ordering logic to also resolve data hazards and recover from branch mispredictions. Hybrid designs that exploit traditional out-of-order checkpointing with idempotence are likely to improve the performance of these processors; however, those designs and their characteristics are not explored in this work.

3. IDEMPOTENT REGIONS

This section describes idempotent regions, the software-level building blocks of an idempotent processor. First, we describe how idempotent regions are identified. Second, we use an example to demonstrate how idempotence can be har-

	Issue	Execute	In-order retirement	Challenges to scalability
In-order	Compiler	Hardware	Hardware	Dynamic effects limit ILP
OoO	Hardware	Hardware	Hardware	Complexity and energy
Idempotent	Either	Hardware	Compiler	Application limits region sizes

Table 1: Contrasting the responsibilities of the hardware and the compiler.

Operation Sequence	Data Dependence Chain	Idempotent?
Write $x \rightarrow$ Read x	RAW	Yes
Write $x \rightarrow$ Read $x \rightarrow$ Write x	WAR after RAW	Yes
Read $x \rightarrow$ Write x	WAR after no RAW	No!

Table 2: Idempotence in terms of data dependences.

nessed in applications and show that applications naturally decompose into a series of idempotent regions. Third, we briefly describe how a compiler can construct large idempotent regions. Finally, we present quantitative data demonstrating that idempotent regions can be very large.

3.1 Idempotent region identification

A region of code (a sequence of instructions) is idempotent if the effect of executing the region multiple times is identical to executing it only a single time[1]. Intuitively, this behavior is achieved if the region does not overwrite its inputs. With the same inputs, the region will produce the same outputs. An input is defined as a variable that is live at the entry point of a region ("live-in" to the region). Thus, a region is idempotent if it does not overwrite its live-in variables. A more precise definition based on data dependence information follows.

By definition, a live-in variable has a definition (a write) that reaches the region's entry point, and has a corresponding use (a read) of that definition after the region's entry point. Because the write must come before entry to the region, the write is not inside the region, and hence there is no write that precedes the first read of that variable inside the region. Hence, *a live-in has no RAW (read-after-write) data dependence before the first read of that variable.* Since a live-in has no RAW data dependence, overwriting a live-in must occur after the point of the read. Thus, an overwritten live-in variable has a WAR (write-after-read) dependence after the absence of a RAW dependence. Finally, it follows that a region of code is idempotent if *there are no WAR dependences that are not preceded by a RAW dependence.* Table 2 guides the reader's intuition.

3.2 Idempotent region construction

As derived in the previous section, a region is idempotent if there is no WAR dependence (an antidependence) that is not preceded by a RAW dependence (a flow dependence). This type of dependence—an antidependence not preceded by a flow dependence—we give a special name: a *clobber antidependence.* Some clobber antidependences are strictly necessary according to program semantics. These clobber antidependences we label *semantic.* The other clobber antidependences we label *artificial.* The following example illustrates semantic and artificial clobber antidependences and how idempotent regions form around them.

An example. Consider the C function `list_push` shown in Figure 4(a). The function checks a list for overflow and then pushes an integer onto the end of the list. The function is not idempotent: with or without overflow, re-executing the function will put the integer onto the end of the already-modified list, after the copy of the integer that was pushed onto the list during the original execution.

The source of the non-idempotence is the increment of the input variable `list->size` on line 22. Without the increment, the function would be idempotent because re-execution would simply cause the value that was written during the initial execution to be overwritten with the same value. However, we cannot do this—the antidependence is required by the semantics of the function. It is a *semantic* antidependence, and it is a semantic *clobber* antidependence because it overwrites a live-in to the function.

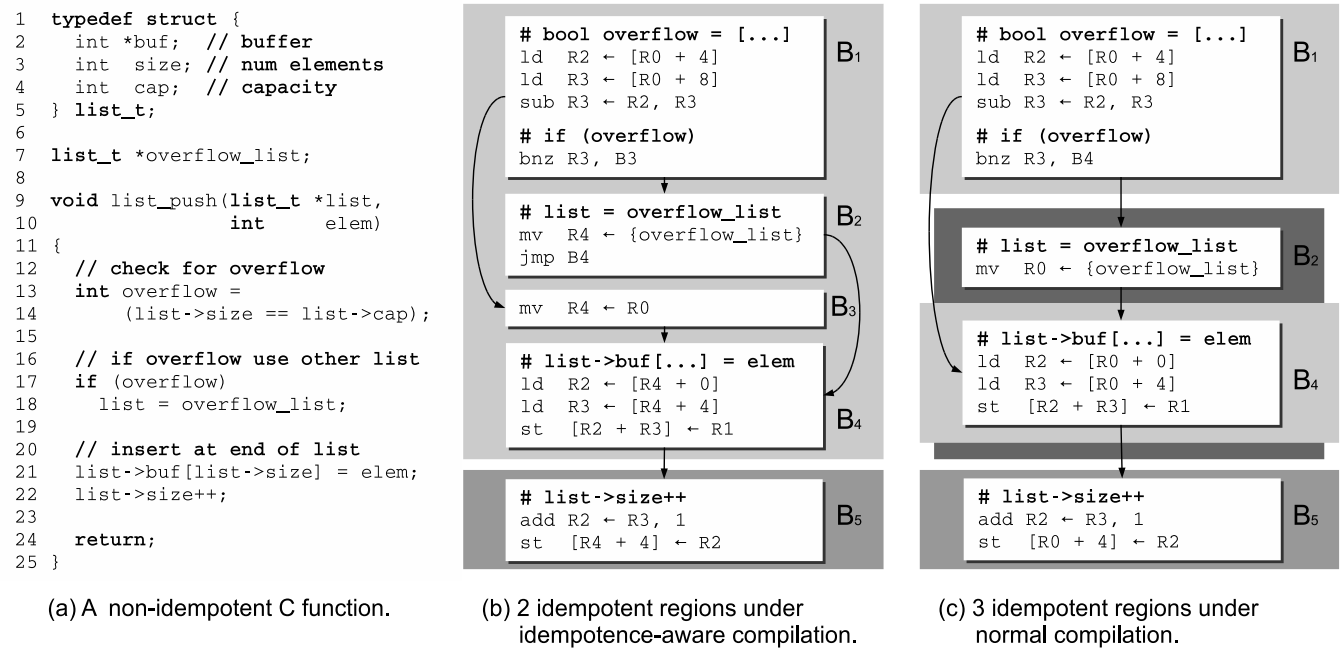
Since semantic clobber antidependences are unavoidable and an idempotent region may not contain any clobber antidependences, the function's idempotent regions necessarily form around the antidependence. Hence, the read and the write of `list->size` must occur in separate idempotent subregions. Figure 4(b) shows this using the control flow graph of the function, with psuedoassembly shown for the contents of the basic blocks. In the figure, the read portion of the antidependence (the load) is in B_4, the write (the store) is in B_5, and each is in a separate idempotent region.

Not all antidependences are semantic antidependences. Many times a region's inputs are overwritten in a way that is merely convenient. With only a limited number of storage resources, a compiler must naturally re-use the available registers and stack memory that are available to it. Re-using these resources may involve overwriting inputs that have already been accessed and are no longer needed—a harmless action under normal circumstances. In fact, most compilers would not have assigned to the extra register R_4 as in blocks B_2 and B_3 of Figure 4(b). Most compilers would instead re-use register R_0 as shown in Figure 4(c). This eliminates the need to move R_0 to R_4 in block B_3, and thus eliminates the need for block B_3 altogether.

However, in doing so the compiler overwrites the input argument `list` initially contained in R_0. This value is now lost. As a result, the idempotence property of the containing region is lost as well. The region must further sub-divide into two idempotent regions separated between the point where R_0 is first read in B_1 and where it is overwritten in B_2. Note in Figure 4(c) that we now have two overlapping regions that both contain B_4: one (dark) has entry block B_2 and the other (light) has entry block B_1.

The kind of re-use exemplified by the write to R_0 is gen-

[1]Although this includes partial executions in general, the final execution must be to completion.



(a) A non-idempotent C function.

(b) 2 idempotent regions under idempotence-aware compilation.

(c) 3 idempotent regions under normal compilation.

Figure 4: An example illustrating the idempotence inherent in applications.

erally good for performance since it minimizes data movement and maximizes locality. However, it reduces the sizes of idempotent regions in the function by introducing clobber antidependences. The clobber antidependences are *artificial* because they are not necessary.

Region definition. We earlier defined a region as a sequence of instructions. However, the "region" containing B_1 and B_4 in Figure 4(c) contains control flow. For this paper, we maintain the definition of a region as a sequence of instructions, and hence consider the two paths through that "region" as distinct and overlapping regions: B_1, and $B_1 \rightarrow B_4$. As a result, there are four idempotent regions in total: B_1, $B_1 \rightarrow B_4$, B_2, and B_5.

3.3 Region construction algorithm

Idempotent regions can be constructed statically or dynamically by a compiler. For this paper, we use a compiler we have built using LLVM [31] that sacrifices some performance to statically compile applications so that their idempotent regions are relatively large. In essence, the compiler works by removing all artificial clobber antidependences on inputs to potentially idempotent regions. These regions then become idempotent. In the case of registers, the registers are assigned such that the artificial antidependences do not emerge. The same technique is applied to registers that are spilled to stack memory—the stack slots are assigned such that artificial antidependences do not emerge. Semantic clobber antidependences are determined using LLVM's built-in alias analysis infrastructure. In particular, may-alias and must-alias clobber antidependences on heap and global memory form the semantic clobber antidependences in the program that divide idempotent regions. Exactly how potentially idempotent regions are identified and where and how these assignments are performed is complex and involves detailed program analysis; in the interest of space, these topics are not covered in this paper and are deferred to a companion technical report [14].

3.4 Idempotent region sizes

To give a sense for how large idempotent regions can be in practice, Figure 5 compares the sizes of the idempotent regions produced by a conventional compiler to those produced by our idempotent compiler. The idempotent compiler constructs idempotent regions using an intra-procedural analysis, forcing splits at function call boundaries. The compiler is robust: it can successfully compile arbitrary C/C++ code and targets both the ARM and x86 instruction sets.

We simulate benchmarks using gem5 [9]. For each benchmark, we measure the average region size occurring over a 100 million instruction period starting after the setup phase of the application. In the case of the idempotent compiler, region size is measured as the distance from region entry to exit. In the case of the conventional compiler, it is measured more optimistically as the distance between dynamic occurrences of clobber antidependences. This more optimistic measurement is used in the absence of explicit region markings in the binary of the conventional compiler.

Figure 5 reports results across the SPEC 2006 [39], PARSEC [8], and Parboil [1] benchmark suites. Across all benchmarks, our idempotent compiler (Idempotent) enables idempotent regions that are, geometrically averaged, 42.9 micro-ops in size, compared to 18.9 micro-ops for the conventional compiler (Conventional). We show results in terms of the number of ARM (specifically ARMv7) micro-ops, and observe that the results for x86 are similar, although region sizes tend to be quite a bit smaller for the x86 conventional compiler due to higher levels of register re-use.

The Parboil benchmarks show the largest region sizes due to their highly regular, streaming memory interactions. Three benchmarks—456.hmmer, 470.lbm, and sad—have larger idempotent regions in the conventional binary than the idempo-

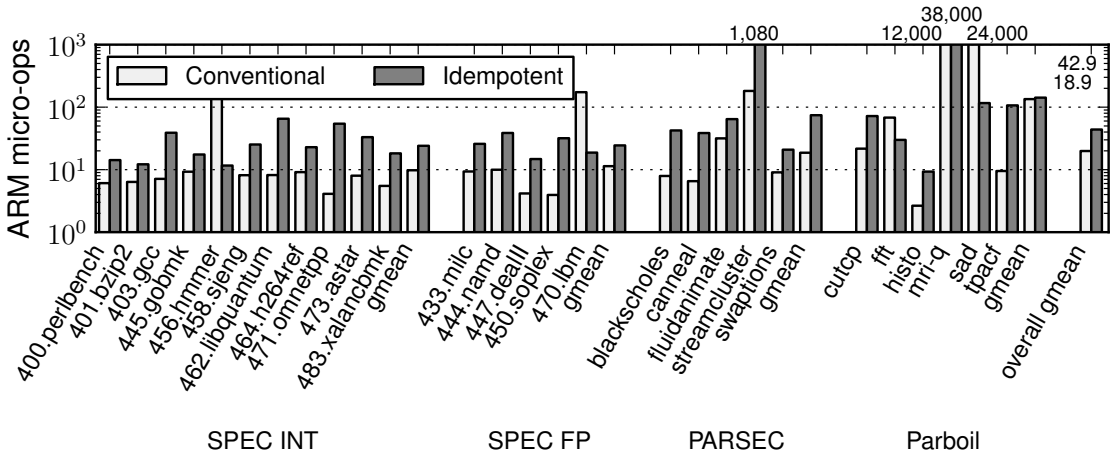


Figure 5: Average region sizes (the y-axis is log scale).

tent binary. This is due to limited aliasing information in the idempotent compiler's region construction algorithm; with small modifications to the source code that improve aliasing knowledge, larger region sizes can be achieved[2]. However, the region sizes across all suites are already quite large—large enough, we argue, to be of practical value to architecture design.

4. ON THE COMPLEXITIES OF IN-ORDER RETIREMENT

Idempotence enables safe out-of-order retirement within a region. This enables valuable pipeline simplifications for processors that employ special-purpose hardware to manage in-order retirement. In this section, we show the ways in which in-order retirement complicates a conventional processor design. In the next section we exploit idempotence to build a significantly simpler design with similar performance.

4.1 A representative processor

As a representative processor, we model an aggressively-designed two-issue in-order processor. We loosely base it on the energy-efficient ARM Cortex-A8 processor core [6]. However, we also make reference to two other widely-used two-issue in-order processor implementations—the Cell SPE [29], and the Intel x86 Atom [21]—when their design choices differ significantly from the Cortex-A8.

Figure 6(a) shows an initial configuration for our processor with features similar to those of the Cortex-A8, but without any high-performance optimizations to overcome the inefficiencies introduced by in-order retirement. In the following sections, we incrementally add in these optimizations. The processor is aggressively pipelined with 3 cycles for fetch, 5 for decode and issue, and the following execution unit latencies: 1 cycle for integer ALU operations and branches; 4 cycles for integer multiply, load, store, and floating point ALU operations; and 8 cycles for floating point multiply and divide. Since the processor is dual-issue, the processor has two integer ALU units. The pipeline employs branch prediction in the fetch stream and we add the capability to flush mispredicted instructions that have issued by suppressing writes to the register file and memory. The pipeline also supports bypassing results over the result bus immediately before writeback.

[2]Note that the conventional compiler measurement does not suffer from this problem because its region sizes are determined based on run-time aliasing that results in actual clobber antidependences.

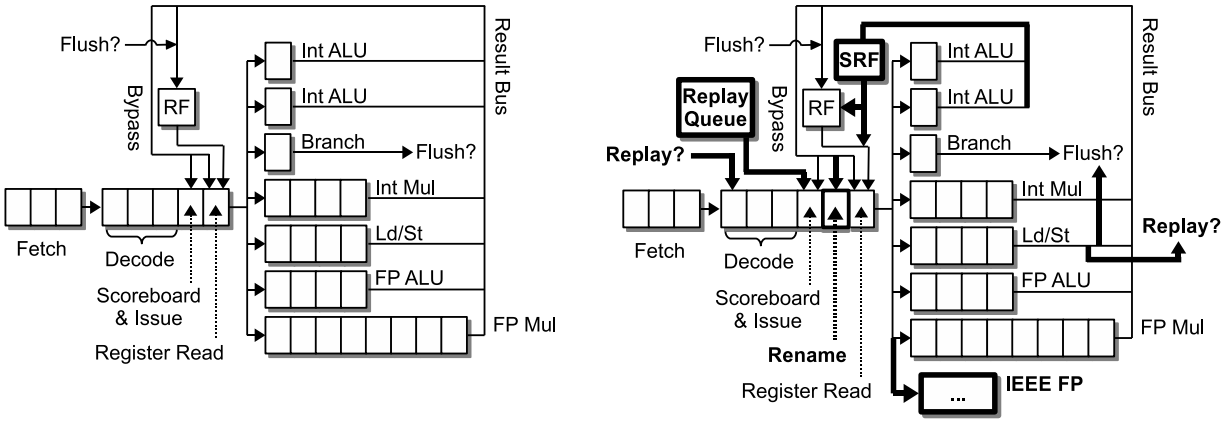
4.2 In-order retirement complexities

Below, we demonstrate the ways in which in-order retirement complicates the design of (1) the integer processing pipeline, (2) the floating point processing pipeline, and (3) support for cache miss handling.

The integer pipeline: staging and bypassing. In Figure 6(a), integer ALU operations complete before integer multiply and memory operations. Hence, the processor cannot immediately issue ALU instructions behind these other instruction types because it will lead to out-of-order retirement. To improve performance, the pipeline can be modified to retire the ALU instructions in order using a technique called *staging*. Using staging, retirement of ALU instructions is delayed so that they write back to the register file in the same cycle as the multiply and memory operations. This technique is used to achieve in-order retirement of integer operations on each of the Cortex-A8, Cell SPE, and Atom processors. The Cortex-A8 has up to 3 cycles [6], Atom has up to 5 cycles [21], and the Cell SPE has up to 6 cycles of staging [29].

A basic implementation of staging involves the insertion of staging latches to hold in-flight results as they progress down the pipeline. Results are then bypassed to executing instructions by adding a unique data bus for each staging cycle and destination pair. Unfortunately, this implementation results in combinatorial growth in the complexity of the bypass network. For this reason, a sometimes preferred solution is instead to implement a special register-file structure to hold results until they are allowed to retire. Results are then bypassed out of this structure in a manner similar to how results are read from the ROB in an out-of-order processor. González *et al.* describe this structure in more detail, calling it the staging register file (SRF) [17].

Figure 6(b) shows the addition of an SRF to our processor to temporarily hold completed results. To determine



(a) Initial configuration.

(b) Supporting in-order retirement.

Figure 6: The complexities introduced by in-order retirement.

whether values should be read from the register file (RF) or SRF after issue, it also includes an extra rename stage in the pipeline to map registers to their location in either the RF or SRF (the Cortex-A8 includes such a rename stage after issue [6]). Finally, the pipeline flush logic is extended for branch mispredicts to also flush the pipeline in the event of an exception in the integer pipeline. It is worth noting that support for result bypassing is more complicated than depicted in Figure 6(b) for processors with execution unit pipelines that consume and produce values at multiple stages, such as the actual Cortex-A8 and the Atom, and for processors with very deep pipelines, such as the Cell SPE.

The floating point pipeline: exception support. In Figure 6(b), integer operations retire after all possible exception points in the integer pipeline, but the floating point pipeline may experience exceptions as well. Although floating point exceptions are typically rare, the IEEE floating point standard requires that five specific floating point exception conditions be minimally detected and signaled [2]. The IEEE standard also recommends precise floating point exception support for trapping to software.

Regarding the latter, supporting precise floating point exceptions in hardware is difficult. Hence, many processors, including the Cortex-A8 and Cell SPE, do not support it [5, 27]. Atom does support it [28]. Unfortunately, the specific details on Atom's implementation are not publicly available. The remaining aspects of the IEEE standard are implemented in the Cortex-A8 using a second, much slower, non-pipelined floating point unit that handles IEEE compliant floating point operations, with exceptions handled entirely in hardware. Figure 6(b) shows this support added to our representative processor.

Cache miss handling. Occasionally, a load or store will miss in the L1 cache. In this case, an in-order processor must typically wait until the cache is populated with the missing data before it can resume execution. In the case of a simple in-order pipeline, the pipeline often simply asserts a global stall signal that prevents instructions from advancing down the pipeline. However, for deeper pipelines with many stages this stall signal must propagate over long distances to many latches or flip flops and thus often forms one or more critical paths [10, 15].

The Cortex-A8 has such a deep pipeline. Hence, it uses a different approach, which is to issue a replay trap—re-use the exception logic to flush the pipeline and re-execute—in the event of a cache miss [6]. The pipeline is then restarted to coincide precisely with the point where the cache line is filled. To enable this coordination of events, the Cortex-A8 employs a replay queue to hold in-flight instructions for reissue. Figure 6(b) shows the addition of the replay queue to our representative processor. Compared to the Cortex-A8, the Cell SPE does not implement special support for cache misses since the SPE cache is software managed, while specific details on Atom's implementation are unavailable.

Summary. While conceptually simple, modern in-order processors are quite complex and exhibit multiple sources of overhead relating to in-order retirement as demonstrated by the differences between Figures 6(a) and 6(b). This includes the extra register state to hold bypass values in the SRF, the additional rename pipeline stage to map operand values into the SRF, additional circuitry to flush the pipeline for exceptions and cache misses, additional floating point resources, a special replay queue to hold in-flight instructions, and associated circuitry to issue from the replay queue and force replay traps. As we show in the next section, all of this additional complexity can be eliminated in an idempotent processor.

5. IDEMPOTENT PROCESSOR DESIGN

For an idempotent processor, out-of-order retirement of integer and floating point operations are not problematic within the confines of a region, floating point exceptions are easily supported precisely, and an idempotent processor pipeline must neither stall nor flush in the presence of a cache miss. It uses only slightly modified scoreboarding logic, and the only additional hardware requirement is the ability to track the currently active idempotent region and only issue instructions from that region. In particular, if a potentially excepting instruction from the currently active region is still executing in the processor pipeline, an instruction from a subsequent region may not issue if it might retire

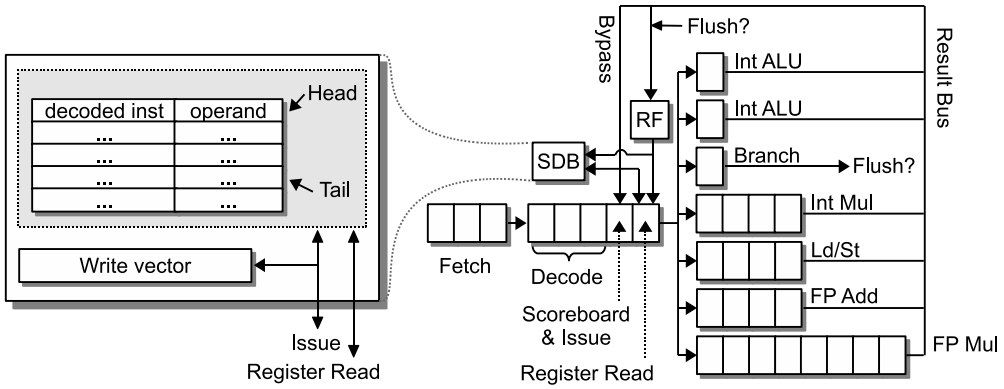


Figure 7: An idempotent processor with a slice data buffer (SDB).

ahead of the exception signal. Constraining issue in this way ensures that the contents of regions retire in order with respect to the contents of other regions. In our design, we use a special counter to track the number of potentially excepting instructions executing in the pipeline (up to 4). The processor only advances the active region and begins issuing instructions from the next region when the counter is at zero.

Enhancing performance. Power and complexity savings are enabled by the opportunity to eliminate staging, bypassing, and support for replay traps in an idempotent processor. To demonstrate how idempotent processing enables out-of-order issue at low complexity cost, we propose converting some of these savings into improved performance by adding a small, 4-entry slice data buffer (SDB), as proposed in previous work [25, 35], to hold miss-dependent instructions and their operands. Instructions that are dependent on a load miss are issued to the SDB for re-issue at a later stage, unblocking the pipeline. Subsequent instructions may then issue out of order with respect to instructions in the SDB. In an idempotent processor, these instructions may also retire out of order; hence, the functionality of the SDB is synergistic with the relaxation of in-order retirement in an idempotent processor. To further exploit the SDB, we add a non-blocking cache with 2 MSHRs. We also allow the processor to use the SDB when no misses are outstanding, in which case instructions blocked on a single operand may issue to the SDB, unblocking the pipeline. This allows the processor to execute ahead of long latency multiplies and floating point operations. Note that, to allow recovery from branch mispredictions as before (i.e. by flushing the pipeline), dependent branch instructions do not issue to the SDB. Details of our particular SDB design follow.

Figure 7 shows the SDB integrated into our idempotent processor. It is implemented as a circular buffer and hence drains in order. Two instructions may issue to and from the SDB per cycle. Similarly to the replay queue described in Section 4, the SDB issues miss-dependent instructions to coincide with the time that the cache miss is serviced. Dependence information is tracked using a special write vector (one bit per register), which tracks which registers are written by instructions in the SDB. The write vector is updated when an instruction issues to the SDB, and it is cleared only once the SDB is empty. Between idempotent regions, if the SDB is non-empty, issue blocks and the SDB drains to ensure that the contents of regions retire in order with respect to other regions.

Because the SDB effectively results in out-of-order issue, mechanisms to handle WAR and WAW hazards are needed (the SDB mechanism itself handles RAW hazards). For register WAR hazards, each SDB entry has space for one slice-independent operand value read from the register file at the time of issue, which allows that value's register to be subsequently overwritten. For register WAW hazards, the write vector is checked to prevent instructions from issuing if they write to a register written to by the SDB. Finally, for memory data hazards we assume all loads and stores alias all stores. Hence, all memory instructions are dependent on earlier stores. We allocate an extra bit in the SDB write vector for this purpose. It is set when a store issues to the SDB, which prevents issue of later loads and stores.

Power and complexity. We argue that our idempotent processor with a 4-entry SDB and a non-blocking cache has better power and complexity characteristics than the representative processor developed in Section 4. While we add a 4-entry dual-ported circular buffer, a bit vector, non-blocking cache functionality, and some associated control logic, we eliminate all of the following: a 6-entry SRF and a 8-entry replay queue (both dual-ported circular buffers), the *entire* rename pipeline stage including the rename table, the pipeline flush and issue logic for exceptions and replay traps (likely to form one or more critical paths), an entire IEEE-compliant floating point execution unit, and all of the control logic associated with each of these various pieces.

Idempotent region recovery. In the event of misspeculated out-of-order retirement (i.e. an exception), the processor takes the following corrective action: (1) it stores the PC value of the excepting instruction to a register (e.g. the exception register); (2) it sets the PC to the PC of the most recent idempotent region boundary point; and (3) it resumes the processor pipeline, issuing instructions only up to the PC of the excepting instruction. When the excepting instruction executes and causes the exception a second time, the processor traps to software precisely with respect to that point in the program. After the exception is handled, execution is resumed from the PC of the excepting instruction. For time-sensitive interrupts, we also allow immediate servicing of the interrupt first, followed by re-execution afterwards.

Processor Configuration	Characteristics
In-Order Processor	Staging, full bypass, replay logic (Figure 6(b))
Idempotent Processor Lean	Out-of-order retirement (Figure 6(a))
Idempotent Processor Fast	Out-of-order retirement, 4-entry SDB, non-blocking cache (Figure 7)
Out-of-Order Processor	24-entry ROB, 16-entry IQ, 16-entry LSQ, 24 INT & 24 FP extra regs

Table 3: Processor configurations.

Hardware	Configuration
Pipeline	fetch/decode/issue/commit: 2-wide, rename/ROB (OoO only): 2-wide
Branch prediction	8kB tournament predictor, 512-entry BTB, 16-entry RAS
L1 caches	32kB ICache & 32kB DCache, 2-way set-associative, 64-byte line size
L2 cache	1MB 8-way set-associative, 64-byte line size, 10-cycle hit latency
Memory	200-cycle access latency

Table 4: Common hardware functionality across all processor configurations.

6. EXPERIMENTAL EVALUATION

Our experimental evaluation compares the performance of an idempotent processor to a conventional processor. Section 6.1 describes our experimental methodology and Section 6.2 presents our results.

6.1 Methodology

We choose to evaluate a diverse set of three benchmark suites: SPEC2006 [39], a suite of conventional single-threaded workloads; PARSEC [8], a suite of emerging workloads; and Parboil [1], a suite of compute-intensive data-parallel workloads. Each benchmark we compile to two different binary versions: an *idempotent binary*, which is the version compiled using the modified LLVM compiler producing the average dynamic region sizes presented in Section 3.4; and an *original binary*, which follows the regular LLVM compiler flow to generate typical program binaries. Both versions are compiled with the maximum level of optimization.

We simulate our benchmarks using a modified version of the gem5 simulator [9]. To account for the differences in instruction count between the idempotent and original binary versions, simulation length is measured in terms of the number of functions executed, which is constant between the two versions. The SPEC and PARSEC benchmarks are fast-forwarded by the number of function calls needed to execute at least 5 billion instructions on the original binary, and execution is then simulated for the number of function calls needed to execute 100 million additional instructions. The Parboil benchmarks are also simulated for the function-level equivalent of 100 million instructions, although, since some benchmarks run for less than 5 billion instructions, the fast-forwarding distance varies by benchmark.

Processor configurations and experiments. Table 3 shows the processor configurations we explore in our evaluation. The *In-Order Processor* configuration models the representative dual-issue in-order processor developed in Section 5. This processor stages and bypasses ALU execution for in-order retirement, does not support precise floating point exceptions, and has a replay queue and associated logic to handle replay traps. *Idempotent Processor Lean* is our idempotent processor configuration that does not include an SDB or non-blocking cache while *Idempotent Processor Fast* includes a 4-entry SDB and a non-blocking cache. Finally, to place our results in context, we also simulate a dual-issue out-of-order processor configuration: *Out-of-Order Processor*. All processors assume the same execution unit composition and latencies, and are otherwise configured identically with details given in Table 4.

Label	Processor Configuration	Binary
Baseline	In-Order Processor	Original
Switch	In-Order Processor	Idempotent
Lean	Idempotent Processor Lean	Idempotent
Fast	Idempotent Processor Fast	Idempotent
OoO	Out-of-Order Processor	Original

Table 5: Simulation configurations.

Our experiments are designed to isolate the performance impact of the idempotent processor's various components, namely: (1) the idempotent region construction by the compiler, (2) the additional result bus contention introduced by variable-length instruction completion, (3) the inter-region issue constraint of the idempotent processor, and (4) the addition of the SDB and non-blocking functionality to the idempotent processor. Table 5 shows the simulation configurations used to measure these components with results normalized to *Baseline*, the performance of the In-Order Processor configuration running the benchmark's original binary version. Switch measures component (1), Lean the addition of components (2) and (3), and Fast the addition of component (4). Finally, OoO measures the performance a traditional out-of-order processor as a reference point. Due to limited x86 support in gem5 at the time of writing this paper, we evaluate these configurations for ARMv7 only.

6.2 Results

Figure 8 shows our experimental results. The leftmost bar (Switch) shows the performance of the in-order processor after switching to the idempotent binary. In all cases, there is some performance degradation as our idempotent compiler forfeits some register locality to create large idempotent regions. For the control- and memory-intensive SPEC INT benchmarks, the performance drop is substantial: the geometric mean is a 8.5% loss. For SPEC FP, PARSEC, and Parboil, the losses are more modest at 4.5%, 3.6%, and 1.0%, respectively. Across all benchmarks, the geometric mean is a 5.2% performance drop.

The second bar (Lean) presents the performance of Idempotent Processor Lean executing the idempotent binary. Compared to Switch, Lean consumes less power and is less com-

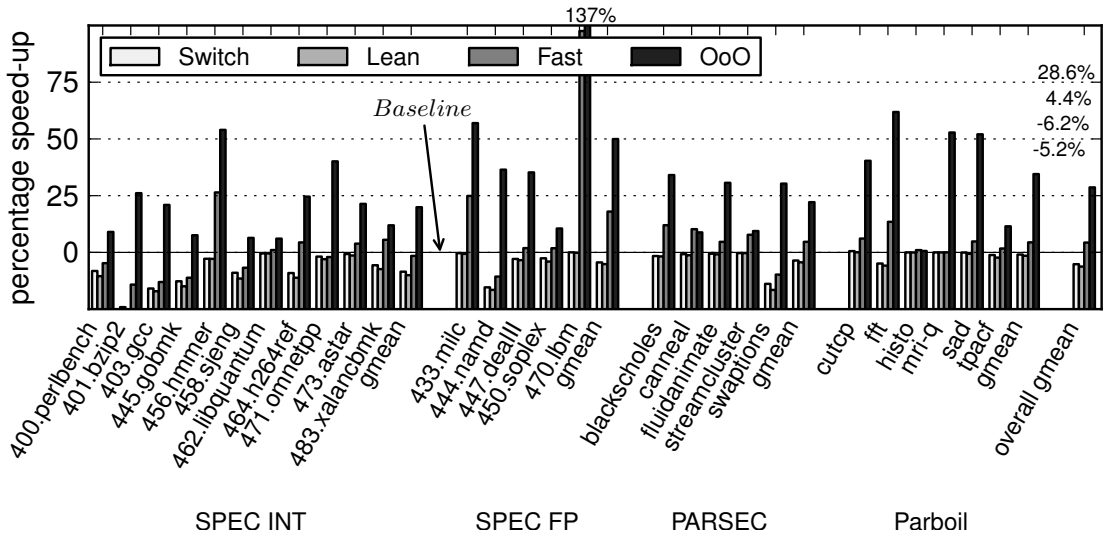


Figure 8: Performance results.

plex overall, but suffers from two additional sources of overhead necessary for correct execution: (1) result bus contention due to out-of-order completion and (2) constrained issue between regions. However, for all benchmarks we see that the additional performance loss over Switch is small: for SPEC INT, SPEC FP, PARSEC, and Parboil, the geometric mean performance loss is 10.0%, 5.2%, 4.4%, and 1.5% over the baseline. The two sources of overhead contribute roughly an equal amount to the performance degradation.

The third bar (Fast) shows results for Idempotent Processor Fast, which adds a 4-entry SDB and non-blocking cache to improve performance at low power and complexity cost. Almost all benchmarks benefit from this functionality, with 470.lbm benefiting most due to its very high store miss rate, which leads to its 98% performance gain. However, the level of ILP exposed by the SDB is sometimes limited, and one application, mri-q, does not benefit at all despite heavy use of long latency floating point operations. Across the benchmark suites the geometric mean performance gains are -1.5%, 18.0%, 4.7%, and 4.3% over the baseline. Across all benchmarks the gain is 4.4% over the baseline and 10.6% over Lean.

The fourth and final bar (OoO) shows the performance of a modern out-of-order processor running the original binary. Although the out-of-order processor performs the best overall, our goal was never to exceed the performance of an out-of-order processor, which achieves its 28.6% better performance over the in-order baseline at substantial power and energy costs. By comparison, Idempotent Processor Fast achieves its better performance over the baseline at effectively no hardware cost.

Summary. Our results show that our fast idempotent processor configuration achieves substantial performance gains over an energy-efficient in-order processor baseline. Despite the absence of quantitative power measurements, which are challenging to obtain for the level of detail we explore in this paper, we nevertheless also claim an overall reduction in both power and processor complexity based on the qualitative analysis presented in Section 5. Hence, our geometric mean 4.4% performance improvements come at, at worst, no additional power or complexity cost. Additionally, we find that our performance is only 24.2% lower than a power-hungry out-of-order processor.

7. DISCUSSION

In this paper, we have described idempotent processing and demonstrated its potential. In this section, we discuss some of the more subtle aspects of idempotent processing.

Forward progress. In our presentation of recovery using idempotent regions, we have assumed that re-execution up to the point of an exception is free of exceptions itself. This may not be true in the case of, for instance, multiple page faults occurring inside the same region. In particular, it is possible for a later page fault to evict from the TLB a page loaded by an earlier page fault. Then, when the region is re-executed, the earlier page fault will occur again, leading to live-lock. To prevent this situation, we propose a transition to in-order retirement: if, during re-execution, the idempotent processor experiences an exception whose PC does not match that of the expected instruction, the processor re-executes the region issuing and retiring instructions in-order and handling exceptions precisely as normal. While this results in a much slower execution of that region, the occurrence of this behavior is intuitively very rare.

Multi-threading and memory consistency. We currently assume that during recovery the values read from memory during re-execution are the same values read during the initial execution. This may not be true in, for instance, a multi-threaded environment where threads modify shared memory in an unsynchronized fashion (i.e. there are data races). Idempotent processing also assumes a relaxed consistency memory model where speculative retirement of memory operations is safe. In cases where these assumptions are too restrictive, idempotent regions could be treated as having atomic memory semantics, such that all stores are buffered until the containing region completes.

I/O and Synchronization. Some instructions are inherently non-idempotent. Certain memory-mapped I/O operations and some types of synchronization instructions (e.g.

atomic increment) are two examples. In this work, we consider such non-idempotent instructions as single-instruction idempotent regions. As single-instruction regions, these instructions execute and retire in-order with respect to all other instructions in the program and it is therefore never necessary to re-execute them for recovery.

8. RELATED WORK

There is a long history of prior work on the complexities of out-of-order execution and support for precise program state in hardware [26, 37, 38]. With respect to out-of-order retirement in particular, the Alpha architecture supported imprecise floating point exceptions with hardware replay to achieve precise state [12]. Additionally, recent work has focused on speculative out-of-order retirement using checkpoints [4, 13, 34]. Our work is similar, but distinct in that it does not use hardware checkpoints for recovery. Rather, we identify and construct implicit, low-overhead checkpoint locations in the application itself using a compiler. The power and complexity overheads of supporting re-execution from these locations in the hardware is minimal, and we demonstrate that the software performance overheads are low enough that the technique is practical and can improve energy efficiency.

Related work on the compiler side includes Gschwind and Altman's exploration of support for precise exceptions in a dynamic optimization environment using repair functions invoked prior to exception handling [19]. In contrast to their technique, our technique can be applied in either a static or dynamic compilation environment. Additionally, Li *et al.* develop compiler-based recovery over a sliding window of instructions [32]. We find that our use of static program regions allows for the construction of large recoverable regions with lower performance overheads. Finally, Mahlke *et al.* explore the idea of exception recovery using restartable (idempotent) instruction sequences in their work on sentinel scheduling [33]. However, their treatment is brief and they apply it only over specific program regions.

The concept of idempotence has been leveraged by Kim *et al.* in the context of thread-level speculation to allow idempotent references to see speculative state [30]. Additionally, Shivers *et al.* use idempotence to achieve atomicity guarantees for garbage collectors [36]. Restart markers, developed by Hampton, also present a similar concept to idempotent regions [23]. They have been proposed as a mechanism for exception recovery for parallel architectures.

Finally, in terms of analysis, Bell and Lipasti explore the conditions under which out-of-order retirement is safe [7]. In the context of a wide-issue out-of-order processor, they find that many operations could commit early given the opportunity to do so. However, the hardware complexity to enable it is considerable. Our work suggests that idempotent processing could bypass this complexity and also simplify certain out-of-order issue structures such as the ROB and LSQ.

9. CONCLUDING REMARKS

Due to slowing energy-efficiency gains from process technology scaling and the continued growth of the mobile computing space, energy efficient microarchitecture design has never been more important. This paper tackles fundamental inefficiencies in the processor by revisiting one of the basic design principles in a microprocessor. It develops the concept of *idempotent processing*, which leverages the principle of idempotence to break programs into regions of code than can be recovered through simple re-execution. Thus idempotent processors allow speculative execution without the need for hardware buffering or checkpoints, relaxing decades-old assumptions on speculative execution that were thought difficult or impossible to overcome. Specifically, we use this approach to simplify the design of in-order processors by allowing out-of-order retirement with the option of instruction-precise in-order recovery when necessary.

We showed that large idempotent regions can be extracted from modern applications by a compiler that can produce binaries with idempotent regions demarcated in them. We developed a simple idempotent processor that eliminates much of the complexity in a conventional processor and shows a performance gain of 4.4% (up to 25%) over a sophisticated conventional in-order processor. Conceptually, the complexity, area, and energy savings are clear. In future work, we will implement our idempotent processor design in RTL or FPGA to quantify these benefits more carefully. Additionally, multi-threading, handling I/O, and operating system issues may require further consideration before a full system can be implemented around this idea.

Re-thinking some of the fundamental principles of machine organization from an energy and power constrained perspective can significantly improve the efficiency of processors. Future architectures are likely to require techniques like the one explored in this paper to improve the efficiency of the processor core itself in addition to techniques like specialization that can opportunistically side-step the core. The general principle of idempotent processing decouples speculation from recovery and could thus enable devices, circuits, and microarchitecture to all be speculative with simple and efficient recovery through re-execution.

Overall, the idempotent processor architecture is a good fit for the energy- and power-efficient processor design space. This includes the mobile computing space, power-constrained many-core architecture space, and vector architectures like GPUs which have traditionally avoided speculation because of recovery overheads. Additionally, application transformations or idempotence-based optimizations could increase the size of idempotent regions thus enabling further uses. In this paper, we have shown that across a wide range of applications idempotent processors form a new class of processor that straddles in-order and out-of-order processors in terms of achievable performance for a given energy budget. Idempotent processors thus provide a valuable platform for the exploration of future architecture designs.

10. ACKNOWLEDGMENTS

We thank the anonymous reviewers and the Vertical group for comments and the Wisconsin Condor project and UW CSL for their assistance. Many thanks to Guri Sohi for several discussions that helped refine this work. Also, thanks for Mark Hill and David Wood for sharing valuable insights during discussions on this work. Thanks to Simha Sethumadhavan for comments on drafts of this work which helped improve the presentation. Thanks to Chen-Han Ho for his early work on microarchitecture designs exploiting the idempotent processing paradigm. Support for this research was provided by NSF under the following grants: CCF-0845751. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

11. REFERENCES

[1] *Parboil Benchmark Suite*. http://impact.crhc.illinois.edu/parboil.php.

[2] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–58, 2008.

[3] International technology roadmap for semiconductors, 2009. http://www.itrs.net.

[4] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO '03*, pages 423–434.

[5] ARM. *Cortex-A8 Technical Reference Manual, Rev. r2p1*.

[6] M. Baron. Cortex-A8: High speed, low power. *Microprocessor Report*, 2005.

[7] G. B. Bell and M. H. Lipasti. Deconstructing commit. In *ISPASS '04*, pages 68–77.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT '08*, pages 72–81.

[9] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, July 2006.

[10] E. Borch, S. Manne, J. Emer, and E. Tune. Loose loops sink chips. In *HPCA '02*, pages 299–310.

[11] N. Clark, A. Hormati, and S. Mahlke. Veal: Virtualized execution accelerator for loops. In *ISCA '08*, pages 389 –400.

[12] Compaq. *Alpha architecture handbook*. October 1998.

[13] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *HPCA '04*.

[14] M. de Kruijf and K. Sankaralingam. Compiler construction of idempotent regions. Technical Report TR-1700, University of Wisconsin-Madison, Department of Computer Sciences, 2011.

[15] J. H. Edmondson, P. Rubinfeld, R. Preston, and V. Rajagopalan. Superscalar instruction execution in the 21164 alpha microprocessor. *IEEE Micro*, 15(2):33–43, 1995.

[16] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ISCA '10*.

[17] A. González, F. Latorre, and G. Magklis. *Processor Microarchitecture: An Implementation Perspective*. Morgan & Claypool, 2010.

[18] V. Govindaraju, C.-H. Ho, and K. Sankaralingam. Dynamically specialized datapaths for energy efficient computing. In *HPCA '11*.

[19] M. Gschwind and E. R. Altman. Precise exception semantics in dynamic compilation. In *CC '02*, pages 95–110.

[20] L. Gwennap. Mobile processors multiply at MWC. *Microprocessor Report*, 2011.

[21] T. R. Halfill. Intel's tiny Atom. *Microprocessor Report*, April 2008.

[22] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA '10*.

[23] M. Hampton. *Reducing Exception Management Overhead with Software Restart Markers*. PhD thesis, Massachusetts Institute of Technology, 2008.

[24] M. Hempstead, G.-Y. Wei, and D. Brooks. Navigo: An early-stage model to study power-contrained architectures and specialization. In *Proceedings of Workshop on Modeling, Benchmarking, and Simulations (MoBS)*, 2009.

[25] A. Hilton, S. Nagarakatte, and A. Roth. iCFP: Tolerating all-level cache misses in in-order processors. In *HPCA '09*, pages 431–442.

[26] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *ISCA '87*, pages 18–26.

[27] IBM. *SPU Instruction Set Architecture, Rev. 1.2*. 2007.

[28] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. 2011.

[29] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), September 2005.

[30] S. W. Kim, C.-L. Ooi, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Exploiting reference idempotency to reduce speculative storage overflow. *ACM Trans. Program. Lang. Syst.*, 28:942–965, September 2006.

[31] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–88.

[32] C.-C. J. Li, S.-K. Chen, W. K. Fuchs, and W.-M. W. Hwu. Compiler-based multiple instruction retry. *IEEE Transactions on Computers*, 44(1):35–46, 1995.

[33] S. A. Mahlke, W. Y. Chen, W.-m. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *ASPLOS '92*, pages 238–247.

[34] J. Martinez, J. Renau, M. Huang, and M. Prvulovic. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *MICRO '02*.

[35] S. Nekkalapu, H. Akkary, K. Jothi, R. Retnamma, and X. Song. A simple latency tolerant processor. In *ICCD '08*, pages 384–389.

[36] O. Shivers, J. W. Clark, and R. McGrath. Atomic heap transactions and fine-grain interrupts. In *ICFP '99*.

[37] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37:562–573, May 1988.

[38] G. S. Sohi and S. Vajapeyam. Instruction issue logic for high-performance, interruptable pipelined processors. In *ISCA '87*, pages 27–34.

[39] Standard Performance Evaluation Corporation. *SPEC CPU2006*, 2006.

[40] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *ASPLOS '10*, pages 205–218.

[41] S. Weintraub. *Industry first: Smartphones pass PCs in sales*. http://tech.fortune.cnn.com/2011/02/07/idc-smartphone-shipment-numbers-passed-pc-in-q4-2010/.