

# Profligate Execution

## Abstract

Nearly four decades after its first implementation in the IBM 360/91 floating-point unit, out-of-order execution remains a powerful microarchitectural tool for improving performance by tolerating long execution latencies. Increasing relative memory latency and the relentless quest for deeper pipelining and higher frequency are driving the need for larger and larger instruction windows. These windows must be large in order to extract enough instruction-level parallelism to fill these deep pipelines and successfully overlap hundreds of cycles of memory latency. Unfortunately, physical constraints and unmanageable design complexity prevent designers from scaling conventional structures like reorder buffers and physical register files to span hundreds of active instructions. This paper proposes a radically different approach that employs multiple small on-chip cores to build a giant logical instruction window that can span thousands of in-flight instructions. We advocate *profligate execution*, where many--sometimes most--instructions are executed in each processing element to reduce the need to communicate and synchronize across processing elements, and disjoint execution threads are spawned only when needed to prevent stalling due to a full instruction window. We describe the microarchitectural structures needed to support profligate execution and show that with a 4-way chip multiprocessor, average speedups of 17% and 212% are achievable for integer and floating point benchmarks, respectively. We also show that in addition to reducing application latency, profligate execution can provide a system throughput benefit as well with speedups that are super-linear with respect to the number of processors in the system.

## 1.0 Introduction and Motivation

Despite a gradual trend towards multi-threaded server workloads, the majority of programming is still dominated by single-threaded code, and single-thread performance continues to be the primary metric by which many processors are evaluated. As decreasing feature size has enabled larger transistor budgets, the Holy Grail of microarchitecture has evolved into identifying efficient mechanisms to partition single-threaded code into independent units that can be executed on multiple functional units in parallel. In superscalar processors these units are individual instructions, and the bulk of the complexity in these processors comes from identifying independent instructions. Out-of-order processors help facilitate simultaneous execution of many in-flight instructions by extending the window of candidate independent instructions at the cost of even more complexity. However all of this complexity is not conducive towards building fast circuits with short clock periods, and current instruction windows are not sufficiently large enough to extract all available instruction-level parallelism in modern applications.

Diminishing performance returns and the difficulty of building large, monolithic instruction windows has resulted in a shift of focus from instruction-level parallelism towards thread-level parallelism, which favors aggregate throughput of multiple threads over single thread latency. The increasing number of transistors that can fit on a single chip as well as the decreasing distance that a signal can travel in a single clock cycle has led to the introduction of chip multiprocessors (CMPs), in which several symmetric processor cores are placed on the same die. However this

technique traditionally offers no benefit in terms of single-thread performance.

This paper proposes a *profligate execution* paradigm, in which the additional resources in a CMP can be leveraged to boost single-thread performance. We describe a technique that splits a single instruction stream into multiple instruction windows that obtains comparable performance to a single monolithic window. As with many techniques that attempt to parallelize programs based on a sequential programming model, the most difficult aspect is untangling multiple dependence chains that have become intertwined through data flow and control flow joins. This work presents solutions to all of these problems, and shows that significant speedup can be achieved through an implementation that requires only minor modifications to each underlying microprocessor core.

We present results showing average speedups of 17% (integer) and 212% (floating point) over a contemporary baseline uniprocessor, and average speedups of 63% (integer) and 204% (floating point) for an alternative system composed of simple, high-frequency in-order cores. We also show that super-linear speedups are possible for some applications (i.e. speedup greater than  $n$  with  $n$  cores), yielding not only latency reduction of a single thread, but also an increase in total system throughput as well. This last result is perhaps counter-intuitive -- it demonstrates that running a collection of threads sequentially on a CMP when each thread is allocated the resources of multiple processors can be faster than the conventional execution model where each processor executes an independent thread in parallel.

This paper is organized as follows: Section 2 introduces the notion of profligate execution and discusses why it is a compelling solution to modern computer architecture performance barriers. Section 2.3 examines this model in greater detail and explains how it can leverage existing memory hierarchies to achieve higher performance. Section 3 presents experimental data collected with a simulation infrastructure that implements profligate execution, Section 4 compares this technique with other related work, and Section 5 proposes future work and concludes the paper.

## **2.0 Profligate Execution Overview**

### **2.1 Background**

The performance benefit of dynamically scheduled processors stems from their ability to continue to perform useful independent computation upon encountering a long-latency event that would otherwise stall the machine. They maintain a window of dispatched but not yet committed instructions that can execute in parallel while waiting for the oldest in-flight instruction to finish. When a long latency event stalls younger instructions from committing (as an L2 data cache miss could), the instruction window fills to capacity. Instructions independent of the miss can begin execution early, however once they have been selected the processor stalls until the miss resolves and instruction resources are freed through the normal commit mechanism.

As the relative gap between processor cycle time and memory access time continues to widen, an outstanding miss to main memory will take increasingly more clock cycles to service. It will therefore require larger instruction windows to identify a sufficient number of independent instructions that can execute to hide the miss penalty. For example: if servicing an L2 cache miss takes 200 clock cycles, then the processor would need to find useful work during the next 200 cycles in order to mask the miss latency. A 4-wide superscalar processor that can issue 4 instructions

per cycle would need to issue 800 instructions beyond the miss in order to sustain its peak performance. Because at least some later instructions will inevitably depend on memory-resident data returned by the miss, the instruction window of candidate instructions will likely need to be larger than 800 entries. Current processors' instruction windows are nowhere near this size, and as memory becomes relatively slower in relation to clock speed the need for large instruction windows grows.

Novel schemes such as early reclamation and reference counting of critical instruction window resources have been proposed in a number of studies in order to achieve a large virtual window without the physical design challenges of a large physical window [23],[5],[9],[10],[20],[26]. However, dedicating significant additional design complexity and/or substantial hardware resources to the sole task of tracking a large number of instructions from a single thread is difficult to justify, particularly since future designs are likely to be limited by thermal and power considerations. At the same time, the increasing prevalence of thread-level parallelism has led to the development of chip multiprocessors. Any attempt to allocate die area or design time for single-thread performance enhancements like large (virtual or physical) instruction windows must now compete with a demand for additional processor cores on the same die, since those additional cores provide attractive levels of performance for many important workloads.

This work is predicated on the assumption that most future high-performance microprocessors will not be uniprocessors, as they are today, but rather chip multiprocessors (CMP's). CMPs consist of multiple processor cores tightly integrated onto the same die, often with shared caches and I/O interfaces. Given an example system with local L1 write-through caches and an L2 cache shared among all the processors, we propose a scheme that can leverage CMP resources to not only exploit thread-level parallelism, but also improve single-thread performance by using those resources to create a large virtual instruction window. Hence, we believe it is a useful solution in environments where single-thread performance is important, and enables design of a single chip that can reap both instruction-level and thread-level parallelism effectively.

The challenges of maintaining a single virtual instruction window across multiple processing elements are numerous. Prior proposals have leveraged compiler support to break sequential programs into speculative threads, and to use varying levels of hardware support to guarantee correct sequential execution semantics for those threads, ranging from fairly simple coherence protocol extensions in the thread-level speculation work [27],[18],[14] to extensive support for resolving both register and memory dependences in the Multiscalar proposal [25]. We advocate an approach that maintains binary compatibility and trades computation for communication and complexity: rather than attempting to precisely partition a program into completely disjoint threads, we partition the work only when cache misses cause a reasonably-sized instruction window to fill up and stop making forward progress. To enable this approach, we redundantly execute most instructions on all processing elements to minimize communication and synchronization. This simplifies the tasks of maintaining precise exceptions, enables a very simple algorithm for partitioning work, and allows us to utilize existing CMP resources to extract much higher single-thread performance out of a single chip.

One might argue that profligate execution is not power efficient, since many instructions are executed repeatedly. We agree; in fact, we envision profligate execution as a mode that is selectively enabled when single-

thread performance is important, or when thread-level parallelism does not exist in the workload. We also point out that profligate execution does not introduce substantial new thermal or power delivery issues into an existing CMP design, since it consumes no more peak power than when all cores are busy executing independent threads. In cases where a fixed power budget prevents all processors from simultaneously running at full speed, utilizing additional cores may require decreasing clock frequency on others. However, profligate execution is most useful with memory-intensive workloads that experience a large number of cache misses, where multiple cores will be stalled waiting for different cache misses to complete. In aggressively clock-gated designs these additional cores will consume only a fraction of the power of a busy core, and are thus unlikely to trigger frequency and voltage throttling of other cores. Detailed evaluation of the power consumption of profligate execution and the necessity and applicability of voltage and frequency scaling to prevent thermal or power delivery issues is beyond the scope of this initial paper, and is left to future work.

## **2.2 Basic Idea**

All processors in a CMP initially execute identical copies of the same program. As long as no cache misses occur they will execute at approximately the same rate and will not communicate with each other. At some point each processor will execute an identical load that misses in the L2 cache. If each behaved as a conventional processor, they would all hide as much of the L2 miss latency as possible until their ROB fills and they stall. Under profligate execution, however, only one stalls and waits for the miss to resolve. The others immediately discard the miss and mark the load destination register poisoned with a special INV bit to indicate that it does not contain a valid value; any subsequent instructions that read an invalid value will not be executed. Instructions already in-flight when the miss is detected will inherit the INV bit through the normal out-of-order wake-up mechanisms and be converted into specially-marked no-ops. Instructions dispatched after the miss is detected read INV bits from the architectural register file and are inserted into the machine as specially-marked no-ops. Selectively executing only the miss-independent instructions is similar to other proposed techniques, such as Runahead Execution [22].

The responsibility of waiting for data from memory and retaining precise state belongs to one processor, and the others can make further progress by ignoring the load and its consuming instructions. When the data is returned from memory it is installed into all L1 data caches, and any subsequent loads to that address will at least partially hit as long as that line has not been replaced or invalidated. The result is that processors that would normally suffer full-window stalls can omit window-wedging events and continue executing future instructions. If they encounter a second cache miss, one will be assigned to wait for the miss to resolve, while the others discard and continue as they did before. This results in an execution model that exploits memory-level parallelism (MLP), in addition to the traditional focus of instruction-level parallelism (ILP). It allows threads to be spawned on CMP nodes that run ahead of the trailing processor in an attempt to execute independent cache misses early. Decades of microarchitectural innovation have fine-tuned modern microprocessors' ability to extract ILP from sequential programs, and increasing transistor budgets have led to the advent of CMPs that can exploit thread-level parallelism (TLP). Profligate execution can bridge the two by utilizing CMP cores to increase single-thread performance when doing so is helpful, however if application performance is insensitive to increased window size profligate execution can be disabled and additional threads

can alternatively be conventionally executed.

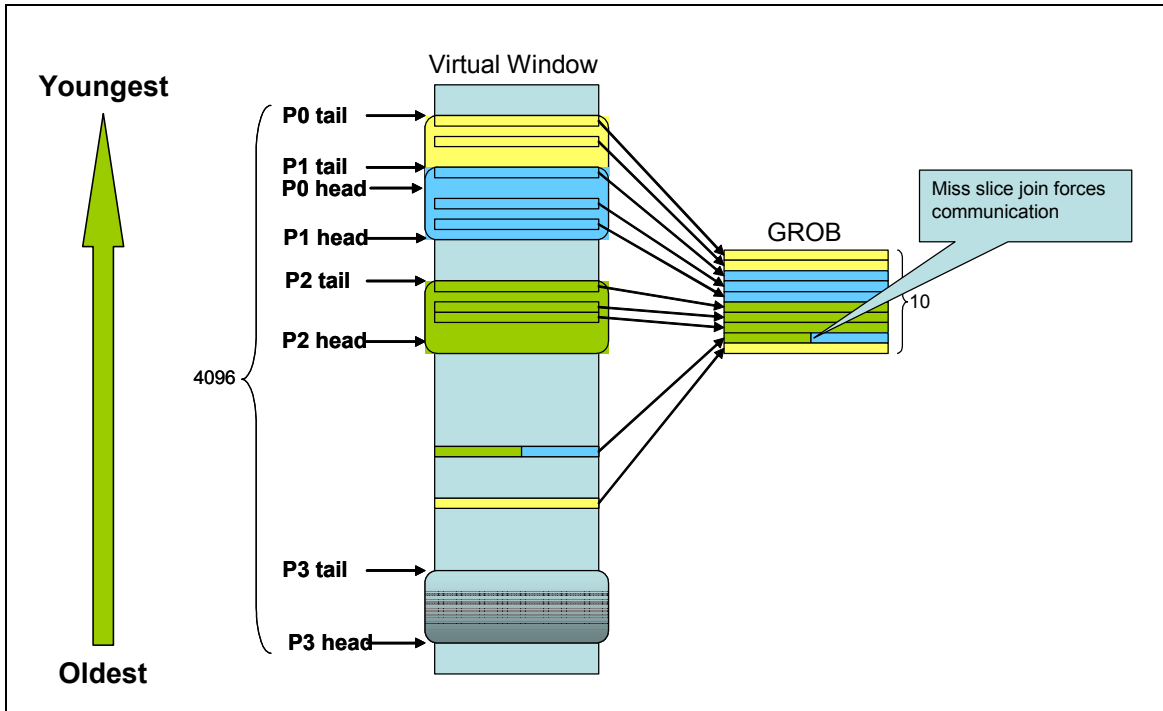
### 2.3 Implementation Details

Conceptually, the profligate execution model exploits a hierarchy to realize a large virtual instruction window. At the lower level of the hierarchy, conventional and well-understood techniques are used to enable out-of-order execution, maintain precise semantics, and guarantee correct memory ordering within a single processor. At the higher level, a collection of techniques must be employed to ensure the same correctness constraints across each of the processing elements. Furthermore, the higher level must also provide a policy and mechanism for partitioning work across the processing elements. None of these techniques fundamentally differ from existing techniques for supporting instruction-level parallelism, but they merit discussion, particularly since the proposed profligate paradigm trades local computation for communication and synchronization, hence enabling reasonably-sized and relatively simple structures and mechanisms for inter-element communication.

Dynamically scheduled processors enable multiple instructions to concurrently execute by using a reorder buffer (ROB) whose purpose is two-fold: it buffers speculatively executed instructions until they can update architected state in-order, and it provides a mechanism to pass speculative results between these instructions. Profligate execution exploits parallelism by executing instructions across multiple processor cores. It can be viewed as implementing a *virtual instruction window* consisting of the set of unique instructions executing on any core at one time. Figure 1 depicts this concept. Each processor conventionally executes instructions in the global instruction stream and allocates a local ROB slot and other associated execution resources. Head and tail pointers indicate their oldest and youngest in-flight instructions. Because multiple processors may be simultaneously executing the same set of instructions, their local windows may overlap, as is the case with processors P0 and P1.

A monolithic instruction window would need to buffer all unique in-flight instructions in the virtual window, which Section 3.5 will show can span several thousand instructions. However because most buffering and forwarding can be satisfied through the smaller individual ROB, we only need to provide a mechanism that handles the cases where this functionality cannot be locally provided. Therefore the only instructions that need to be buffered beyond those already contained in individual instruction windows are the forward instruction slices that depend on L2 misses, which previous work has shown to be quite small [19],[26]. We propose containing slice instructions in a *Global Reorder Buffer* (GROB) to facilitate inter-core communication and precise exception recovery. Figure 1 illustrates the relationship between the GROB, virtual instruction window, and local instruction windows. While the GROB only needs to contain enough entries to contain the forward slice of the miss, the remainder of this section shows that actual number of entries that participate in synchronization between processing elements is far fewer.

This section describes the rare cases where cores must communicate, and describes an implementation to accomplish such synchronization. Specifically it presents four requirements under which synchronization is necessary: miss slice joins between threads that are executing privately on separate processing elements; maintaining precise state for branch recovery and exception handling; and, finally, maintaining a coherent and consistent view of memory. It also describes how a Global Reorder Buffer can detect and implement these synchronization requirements.



**Figure 1. A Large Virtual Instruction Window.** Local reorder buffers in P0, P1, P2, and P3 contain partially overlapping sets of instructions in the virtual instruction window. Only those instructions that are part of forward L2 cache miss slices allocate physical space in the global reorder buffer. Only a subset of those need to communicate, due to (for example) a miss slice join between P1 and P2.

## 2.4 Global Structures

The Global Reorder Buffer (GROB) functions similar to a conventional ROB in that it buffers and forwards speculative results before they irrevocably update program state. It is organized as a circular FIFO queue where each entry corresponds to a unique slice instruction “owned” by a single processor and contains three fields: the output register number, that register’s value, and a logical timestamp used for relative age comparisons. When a slice instruction is committed by its owner its result is copied into the GROB at the index specified by that processor’s *GROB tail*. When a processor commits a slice instruction (regardless of whether it actually executed that instruction) it increments its GROB tails modulo the GROB size. The fact that all processors observe the same instruction stream and identify the same slice instructions enable them to update their GROB pointers locally and consistently, albeit at different times.

After all GROB tails have advanced past a GROB entry it is reclaimed and its output value is copied into the *Global Architectural Register File* (GARF). Each processor also maintains a *Global Register Alias Table* (GRAT) that provides the GROB index of the most recent miss-dependent producer of each logical register. This provides a mechanism to match an arbitrary source register to its corresponding GROB entry in the event that a value needs to be communicated across cores. The entry timestamps are used to determine if the source register value resides in the GROB or the GARF. A newer producer timestamp indicates that that GROB entry has been recycled and re-allocated to a younger instruction and that the correct value has been retired to the GARF. The GROB, GRF, and GRAT have analogous counterparts in traditional dynamically-scheduled systems (such as the Intel P6 microarchitecture’s ROB,

RRF, and RAT [23]). Their operation is illustrated by example in the following sections.

## 2.5 Slice Joins

Localizing computation dependent upon cache misses removes the primary culprit of full-window stalls from other instruction windows. The processor responsible for handling the miss waits for data to return from memory and is the only core that executes dependent instructions. However, some instructions' source operands can reside in different slices executing on different cores (for example an add that sums two values in memory, each of which missed in the cache and was serviced by a different processor). In this case no processor has both operands and therefore some form of communication is required. A variety of possibilities exist for determining which processor will wait for the missing operand and which will discard the instruction and poison its output register. We adopt a simple policy that designates the producer of the left operand responsible for retrieving the right operand from the GROB and executing the join instruction.

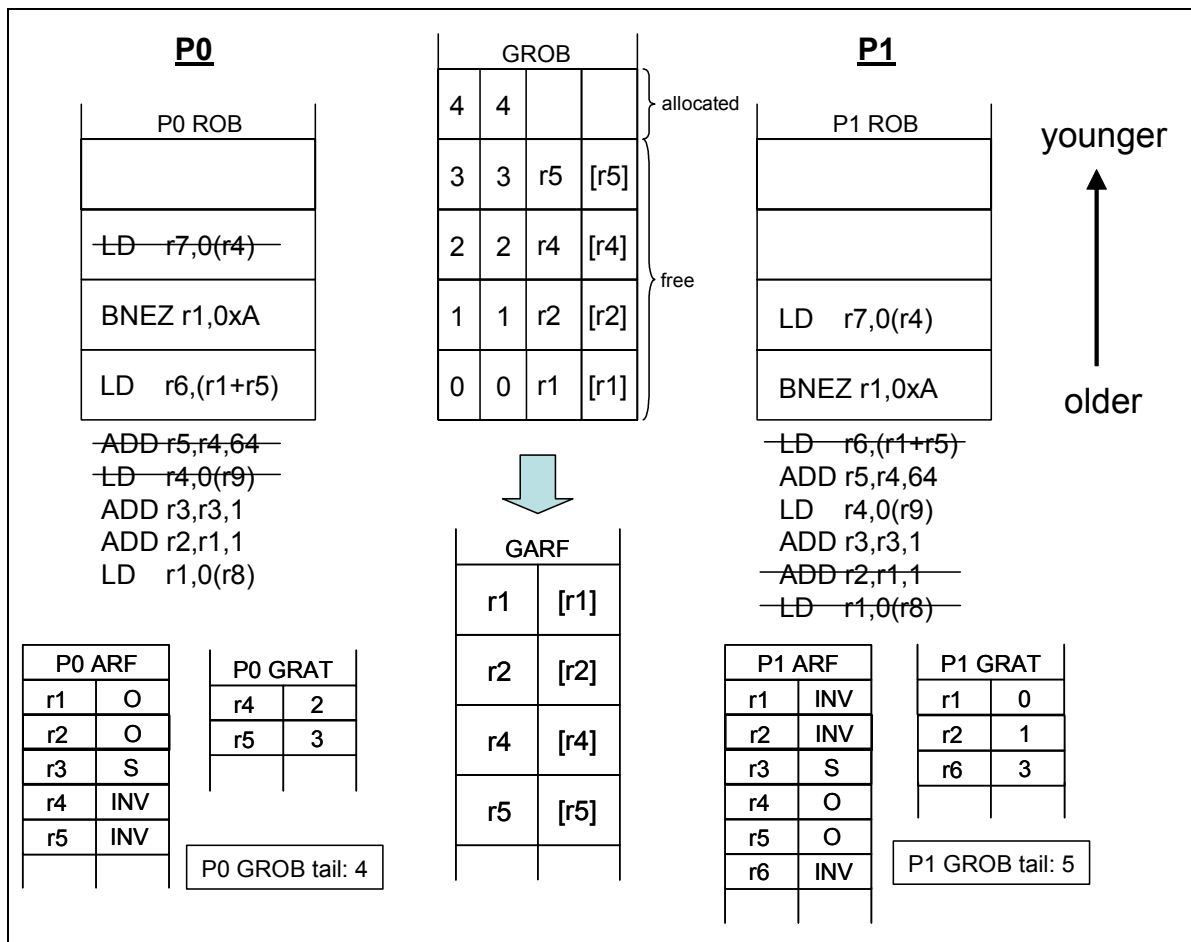
In addition to an INV bit, each register contains an O ("Owner") bit to indicate dependence on a miss executed by that processor. The INV and O bit are mutually exclusive; a register can have either set or none (denoted "Shared" state). Instructions with two source operands can therefore have the nine states listed in Table 1. For a given combination of source states, each row indicates the resulting destination register's state, whether the instruction is executed, and whether a GROB read or write is necessary. If a processor has both operands locally available in the S or O state, it executes the instruction normally. If either operand is marked INV the processor discards the instruction and continues executing, with the exception of when the left operand is marked O and the right operand is marked INV. In this case no processor has both inputs and the owner of the left operand needs to retrieve the right operand value from the GROB as follows: The right source operand number is looked up in the GRAT to determine the GROB index of the last producer. Depending on the result of the timestamp comparison between the producing and consuming instructions, the right operand value is copied from either the GROB or GARF and the instruction is executed.

**Table 1: Miss Slice Joins -- Register States**

Left Op	Right Op	Dest	Execute?	Read GROB?	Write GROB?
INV	INV	INV	N	N	N
INV	S	INV	N	N	N
INV	O	INV	N	N	N
S	INV	INV	N	N	N
S	S	S	Y	N	N
S	O	O	Y	N	Y
O	INV	O	Y	Y	Y
O	S	O	Y	N	Y
O	O	O	Y	N	Y

Figure 2 provides an example. It shows two processors in a profligate execution system as well as their interaction with the additional hardware structures described above. P0 loads address [r8] and upon detecting that it misses in the L2 cache, determines that it will be responsible to wait for the data to return from memory. It marks r1's O bit, which is then inherited by the subsequent instruction and propagated to r2. When P0 eventually commits the load it copies the result from r1 into GROB[0]. It then increments its GROB tail from 0 to 1. When it commits the

next instruction it likewise copies r2 into GROB[1] and increments its GROB tail to 2. The third instruction does not depend on a miss and commits normally. The fourth instruction misses the L2 and is assigned to P1. P0 discards the miss, marks r4 INV, updates the entry for r4 in its GRAT to indicate that GROB[2] will have the missing value, and increments its GROB tail to 3. The fifth instruction inherits r4's INV bit and similarly marks r5 as occupying GROB[3] before incremented the GROB tail to 4. When the sixth instruction reaches the ROB head P0 finds its right operand (r5) in INV state and its left operand (r1) in owned state. This corresponds to the seventh row in Table 1, and requires that P0 retrieve r5 and exclusively execute the instruction. P0 looks up r5 in its GRAT, which indicates that r5 will reside at GROB[3]. It issues a blocking read to that entry and when r5 is supplied by the GROB it is copied into P0's architectural register file (ARF) in shared state. The instruction now has both source operands and can execute.



**Figure 2. Synchronization Example.** Each processor maintains a ROB (oldest four entries shown), GRAT, ARF (register state shown), and GROB tail. The GROB columns are: index, timestamp, register number, register value. The GARF contains the register number and register contents. Brackets are used to denote register contents.

P1 operates similarly. Because P0 is handling the initial load miss to [r8], P1 marks r1 INV. The second instruction propagates the INV bit to r2. Neither of these instructions consume execution bandwidth nor do they access the GROB, however they will increment P1's GROB tail as they commit. The third instruction (ADD) is miss-

independent and executes normally. The fourth instruction is an L2 load miss that P1 will handle; therefore it sets the O bit for r4 in its ARF. The O bit in r4 is then propagated to r5 in the next instruction. The sixth instruction is a miss slice join that P0 is responsible for executing (because it has the left operand), therefore P1 does not execute it and marks r6 INV. Finally P1 increments its GROB tail from 4 to 5 to point to the GROB entry that it should write to next.

At this point P0 and P1's GROB tail pointers are 4 and 5, respectively. Because they have both advanced past entries 0-3, these first four entries can be allocated to later slice instructions. When a GROB entry is overwritten its output value is copied in the GARF. For example, P1 may continue committing instructions and allocating GROB entries for their miss-dependent results. Its GROB tail will eventually wrap around the circular queue and entry 3 will be overwritten. If P0 reads GROB[3] and determines that its timestamp is newer than the load join instruction, it retrieves r5 from the GARF instead. Logical timestamps are incremented concurrently with GROB tail increments, and can be limited to  $\text{LOG}_2(\text{GROB size}) + 1$  bits by utilizing a technique such as that proposed by [16] to deal with counter overflows.

## 2.6 Control Flow

While individual processors can discard instructions that contribute towards the global dataflow of a program, similarly discarding branches will result in ambiguous control flow. If a branch direction or target depends on an L2 miss, then the processor waiting for the miss needs to communicate the branch outcome to the remaining processors in the system. The other processors therefore need to wait until the branch is resolved before they can commit it. When a branch with an invalid source operand reaches a processor's ROB head, it retrieves the operand value from the GROB/GARF in the same way it would for a miss slice join operand. It looks up the source operand in the GRAT to identify its last producer, copies the missing value from the GROB or GARF into its own local register file in shared state, and executes the branch. In the example presented in Figure 2, P1 executes a branch whose source operand (r1) is marked INV. It can predict the branch outcome and continue fetching and executing instructions within its local window, however it cannot commit the branch until its prediction has been verified. It obtains the GROB index of the last producer of r1 from the GRAT (entry 0), and reads that entry from the GROB. It compares the timestamp of GROB[0] with that of the branch to determine if the GROB output is valid or if it instead needs to read the value from the GARF. Even though the branch falls within the miss slice, it does not write to a general-purpose register and therefore will not allocate a GROB entry when it commits.

It may initially seem that stalling miss-dependent branches will limit parallelism and impact performance. However other work has shown that branches that depend on cache misses are more likely to be mispredicted, and those mispredictions are likely to occur shortly after the branch [17]. Therefore even if branches marked invalid were speculatively committed, little additional useful work would be exposed. This is consistent with our findings that relaxing this constraint and allowing branches to speculatively commit while waiting on a dependent load miss does not increase performance for most benchmarks.

## 2.7 Maintaining Precise State

Committing an instruction updates the architected state of the processor and deallocates any resources it consumed during execution. The danger in committing instructions out of program order is that doing so implies that

no older definitions of that logical register will be needed, which may not be true in the event of an exception. Although individual processors in profligate execution commit instructions in-order with respect to their local instruction windows, commits occur out-of-order with respect to the global instruction stream. When an exception is raised some registers may depend on a miss handled by another processor and are marked invalid. Furthermore, that other processor may not have the missing registers if it discarded the excepting instruction and, in the course of committing subsequent instructions, overwrote those registers.

Correct exception recovery in profligate execution is enabled by the fact that all slice register values are first copied to the GROB, and then to the GARF. Consider again the example in Figure 2. If the load from [r4] causes a page fault when it reaches P1's ROB head, precise exception semantics mandate that all older instructions but no younger instructions have committed their results. However P1's register state is incomplete because r1, r2, and r6 are marked invalid in its ARF. Because the load address depends on a miss handled by P1, P0 will not observe the exception and will continue committing instructions.

Before P1 branches to the page fault exception handler routine it scans its ARF to identify its missing register values. It looks up invalid registers in its GRAT to determine their locations and reads those values from the GROB or GARF. At that point it has a valid set of registers that it copies to the remaining cores and restarts their execution at the excepting load's program counter address.

## **2.8 Enforcing Memory Dependences**

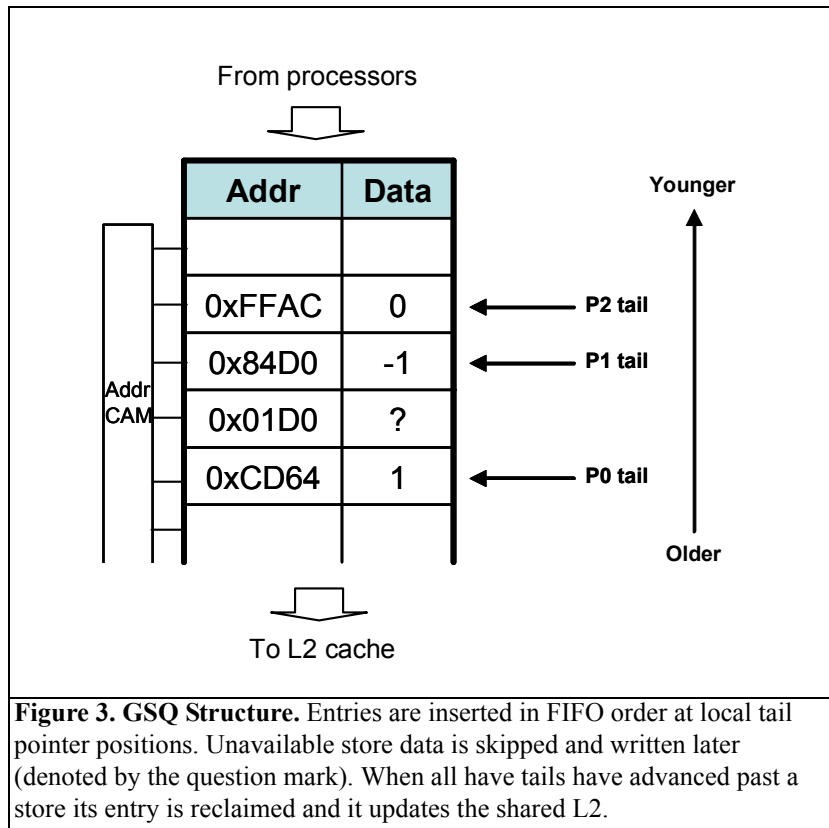
Profligate execution exposes additional ILP and MLP by allowing some cores to execute and commit instructions ahead of others. However memory operations must still appear to complete in program order and adhere to uniprocessor program semantics. This section discusses the requirements sufficient to track and honor memory-based dependencies between instructions. Conceptually, these techniques are similar to conventional methods of tracking memory dependencies employed in out-of-order uniprocessors. We propose an intermediary FIFO structure called the *Global Store Queue* (GSQ) that resides between the private L1 caches and shared L2. Similar to a conventional store queue, the GSQ's purpose is to prevent WAW and RAW violations.

### **2.8.1 WAW Violations**

Allowing writes to update the L2 and memory in the order that they locally commit can result in one processor committing a store before an older store to the same address on a different processor, creating a write-after-write (WAW) violation. To deal with this problem all stores are inserted into the GSQ when they locally commit and are released to the L2 in program order.

Similar to the GROB, all cores maintain a GSQ tail index that points to the entry corresponding to its oldest uncommitted store. When a processor commits a store it writes its address into the entry pointed to by its GSQ tail and, if locally available, the store data. One or both of these fields may be redundantly written by multiple cores; this can either be permitted (since the values written are identical), or a potential write-port optimization could first read the GSQ and only write a field only if it does not yet exist. After a processor writes a store into the GSQ it increments its tail modulo the GSQ size to point to the next entry. As the trailing tail advances past store entries they are released to the shared L2 in program order. This design is depicted in Figure 3 and is similar to an ordered, non-coalescing

store buffer [24].



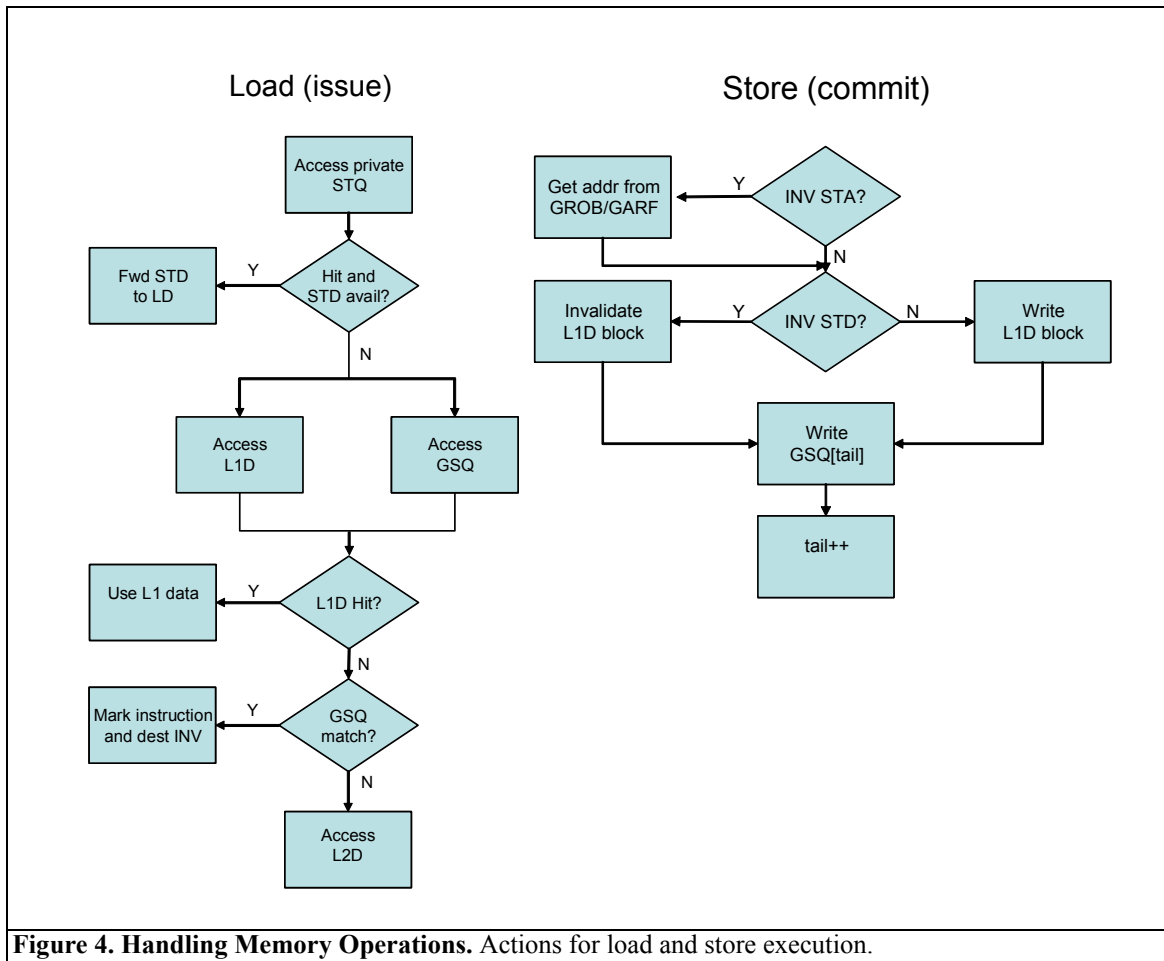
### 2.8.2 RAW Violations

In addition to applying stores to memory in program order, we must also ensure that loads receive the value written by the most recent store to that address. A conventional uniprocessor accomplishes this by comparing loads to older in-flight stores in its store queue. If an address match occurs, the load must either wait for the store to commit, or the store can forward data directly to the load.

Profligate execution must perform an analogous match across the window of in-flight stores in the GSQ. However, we exploit the inherent hierarchy in our design to streamline this process. Processors still perform conventional local lookups in their L1 data cache and local store queue; empirically, the vast majority of independent and dependent loads are handled in this fashion. That is to say, independent loads will usually hit in the L1, requiring no off-core communication. Similarly, dependent loads usually occur within the scope of the local store queue, and are resolved locally with conventional store queue forwarding, or, if the load and store are further separated, are handled by the local cache, since the vast majority of store instructions are executed profligately on all cores, and write their results into each local cache. The only loads that must leave the core are those that miss the L1 and the local store queue. In this case, the load is sent to the L2 and the GSQ, and its address is compared against older stores in the GSQ. An address match indicates that the data in the L2 or main memory is stale and the correct value is (or will be) in the GSQ. The fact that most load addresses are resolved within the scope of the local core is important because the associative nature of the GSQ can make searching it costly.

When a processor commits a store it writes its private L1 cache. However if the store data is not available

(due to an invalid source register), updating the GSQ without concurrently updating the L1 can violate RAW ordering. In this case a younger load of the same cache line could read the stale value from the L1 and fail to identify that the correct data will be supplied by a younger store to the GSQ. Therefore when a processor commits a store with invalid data it must invalidate any matching block from its L1. This forces younger loads to miss the L1 and search the GSQ for an address match. Figure 4 depicts this decision process.



Several options exist when a load matches an older store in the GSQ. The load can simply wait until the store updates the L2, which it can then read from. Alternatively the matching GSQ entry can forward data directly to the load if it is available. We have performed studies suggesting that adding this forwarding ability does not significantly increase performance. This makes sense: unavailable data likely depends on a cache miss and won't be ready for some time. Therefore the time between when it arrives and when it exits the GSQ tends to be small. On the other hand, stores with miss-independent data are executed by all processors and would therefore be available in the L1 cache of the processor executing the load. A third option treats loads that match GSQ stores with incomplete data as misses, rather than letting potentially all cores stall and wait for the delinquent store data to arrive. We adopt this last approach.

### 2.8.3 Store Addresses

Normally, when a source operand is invalid the instruction is not executed by that processor. However treat-

ing stores in this manner can result in a loss of memory dependence information between instructions. If a store is discarded, an older store may incorrectly forward data to a younger load. The load actually needs the value produced by the younger store, which is no longer visible. We deal with this problem by forcing all processors to compute all store addresses. If a store address register is marked invalid it is looked up in the GRAT and read from the GROB or GARF as it would for a slice join or invalid branch operand. This ensures that the addresses of all stores older than a given load appear either in that processor's local store queue or the GSQ. We have observed that the fraction of store addresses that depend on load misses is typically small, and forcing all processors to generate all store addresses does not significantly impact performance.

#### **2.8.4 Invalidation Granularity**

Initially it may seem that better performance can be attained by allowing stores of poisoned data to invalidate the L1 cache at sub-line granularity. We have investigated performing L1 invalidates for individual words within a cache line, and have found that performance remains essentially unchanged. This can be explained as follows: When we eliminate false sharing with word-level invalidates, a younger load to a different address within the same line as the private store will still hit in the L1. With line-level invalidates, the load will be treated as a miss, and one processor will stall and wait for it. The other processors will discard this load and continue to make forward progress. Therefore the penalty of false sharing in this case is that additional loads will be needlessly discarded, possibly resulting in additional GRF communication. However the remaining processors that discard the load can continue executing and uncover further parallelism, and the performance impact is minimal.

#### **2.8.5 Complexity**

The additional complexity introduced by the GSQ comes from two sources: inserting locally committed stores and comparing load addresses to those stores.

**Stores:** Inserting a committed store into the GSQ involves a simple RAM access indexed by the processor's GSQ tail pointer. The address field is written and updated with store data (if available). Because this GSQ access is non-associative and stores are typically removed from the critical path of execution through the addition of a store buffer, we do not expect the latency of store insertion to have performance implications.

**Loads:** Loads, on the other hand, are not off the critical execution path, and overall performance can be sensitive to their latency. However unlike a conventional uniprocessor store queue used for comparison to all younger loads, only loads that miss in private L1 caches must access the GSQ. When an L1 miss occurs the GSQ is indexed by the load address and indicates whether or not it contains an older store to the same address. If it does, the load is treated as a miss and discarded; otherwise the load can access the L2 normally. Unlike the case of store insertion, the GSQ needs to be content-addressable by load address, however it can be accessed in parallel with the L2. In the common case the load will not depend on an earlier in-flight private store and there will be no additional penalty as long as the GSQ access latency does not exceed the L2 access latency.

**Size:** The GSQ must be large enough to contain all in-flight stores across all processors. Section 3.5 will show that profligate execution can enable virtual window sizes of up to several thousand instructions. If we estimate that roughly one in eight instructions executed is a store, the GSQ will need to hold about 250 entries to enable a

2000-entry instruction window. As previously discussed, if these entries can be searched within the L2 access latency, the GSQ is unlikely to inhibit performance. Furthermore, unlike other large-window proposals, profligate execution only requires a single large store queue for chip, rather than one per core. Local store queues handle most accesses and can continue to be small and fast.

## 2.9 Miss Partitioning

Up to this point we have described the mechanics of profligate execution, but have said little regarding how it is determined which processor will handle a given cache miss. This section describes one possible strategy that we implemented which attempts to maximize the size of the virtual instruction window.

One processor is designated the *lead processor* and never stalls on cache misses to main memory. However even though it drops all misses and their dependent instructions, it still accesses the memory system and prefetches data into the shared L2 and private L1s. When another processor later executes the same load, either the prefetch will have successfully completed and the load will hit in the cache, or the load will miss to main memory (either because the prefetch has not yet completed or because the prefetched data has since been evicted). If the load misses, the cache controller assigns the first non-lead processor that executes it the responsibility of waiting for the miss to complete. If the trailing processors also experience a miss when they later execute that load, the cache controller indicates to them that they should discard the load and mark the destination register poisoned.

Disallowing the lead processor to stall on misses can increase ILP and MLP by maximizing the number of unique in-flight instructions across all cores. However it is possible that the lead processor gets too far ahead, to the point where a large number of its local registers become poisoned. Because most instructions will therefore be discarded, this processor will fail to perform useful work. Similarly, the further ahead the lead processor gets the more likely it becomes that one of its branches will depend on private data and will stall until that data is made available by its producer. In either of these two cases it may have been more beneficial if this processor waited for any cache misses it uncovered to complete, particularly if the number of outstanding misses exceeds the number of cores available to wait for them. There clearly exists a large spectrum of miss partitioning schemes that can be envisioned and we leave an in-depth exploration of them to future work.

## 3.0 Results and Analysis

### 3.1 Methodology

We evaluated profligate execution with a simulator loosely based on SimpleScalar 3.0 [6], and substantially modified to model multiple active instruction windows, inter-processor communication delays, a global reorder buffer (GROB), and a global store queue (GSQ). The SPEC2000 integer and floating point benchmarks used were compiled for the Alpha instruction set with peak optimization by the DEC OSF optimizing compiler, and executed with the reference input sets. Benchmarks were fast forwarded 1 billion instructions before timing simulation was collected on the following 10 billion instructions using the SMARTS sampling methodology [28]. All no-ops are removed from execution and consume no processor resources. Table 2 presents the machine model configuration used to collect data.

**Table 2: Machine Configurations**

	Base	Niagara
Out-of-order execution	4-wide fetch/issue/commit, 10-cycle pipeline, 64 ROB, 32 LSQ, 64 rename registers	In-order execution
Functional Units	4 integer ALU, 2 FP ALU, 2 integer MULT/DIV, 2 FP MULT/DIV, 2 memory ports	1 integer ALU per core, 1 FP ALU per chip, 2 memory ports
Branch Prediction	Combined bimodal (16k entry) / gshare (16k entry) with selector (16k entry), 16 RAS, 1k entry 4-way BTB	Combined bimodal (16k entry) / gshare (16k entry) with selector (16k entry), 16 RAS, 1k entry 4-way BTB
Memory System (latency)	32KB 2-way 32B line IL1 (2), 8KB 4-way 16B line DL1 (2), 512 KB 4-way 64B line unified L2 (15), main memory (500), hardware prefetcher	16KB 2-way 32B line IL1 (2), 8KB 4-way 16B line DL1 (2), 2MBKB 4-way 64B line unified L2 (15), main memory (500), hardware prefetcher
Profligate Execution Parameters	4 processor CMP, 512-entry GROB, 10-cycle GRAT+GROB latency, 256-entry GSQ, 15-cycle GSQ access latency	same

### 3.2 Area Requirements

Any optimization that adds new hardware structures must compete for die area that could otherwise be allocated to additional processor cores or cache capacity. Here we provide a short summary of the additional area required for profligate execution for an architecture with 64-bit addresses, a 64-bit datapath, 32 integer registers, and 32 floating-point registers.

**GROB** (1 per chip): Each entry contains three fields: an output register specifier (6 bits), a register value (64 bits), and a logical timestamp (10 bits).  $512 \text{ entries} \times 80 \text{ bits/entry} = 5\text{KB}$

**GSQ** (1 per chip): Each entry contains two fields: an address (64 bits) and a data value (64 bits).  $256 \text{ entries} \times 128\text{bits/entry} = 4\text{KB}$

**GRF** (1 per chip): Each entry contains a 64-bit register value.  $64 \text{ entries} \times 64\text{bits/entry} = 512\text{B}$

**GRAT** (1 per processor): Each entry contains 9 bits (for 512-entry GROB).  $64 \text{ entries} \times 9 \text{ bits/entry} = 72\text{B}$

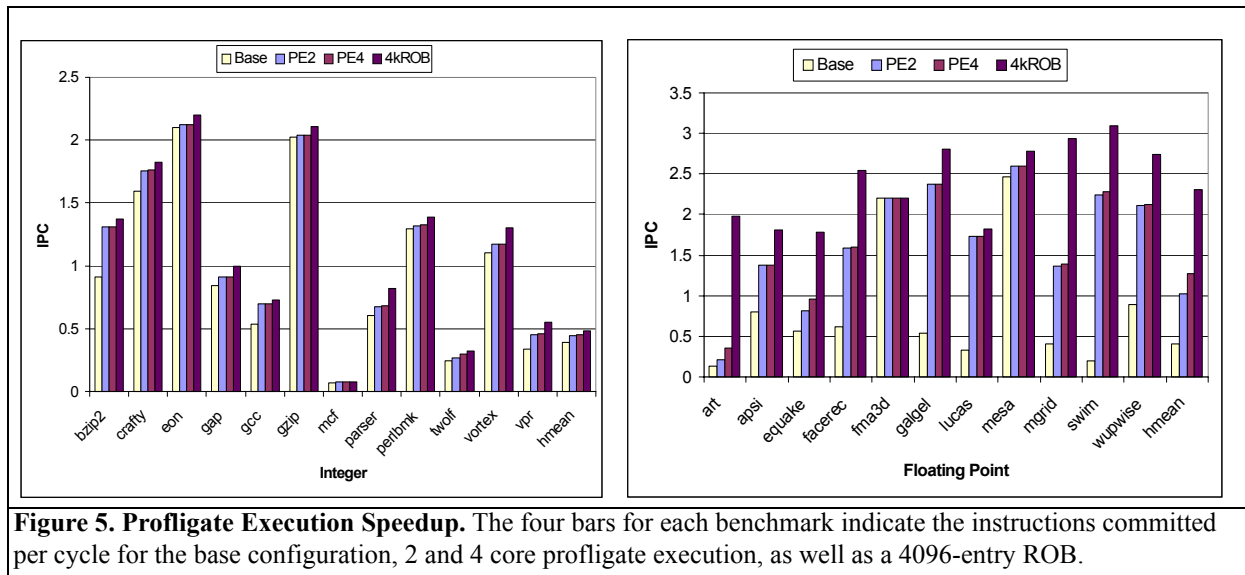
For a four processor system this would translate to less than 10KB of total additional storage, which is insignificant compared to today's per-die transistor budgets.

### 3.3 Number of Processing Elements

Figure 5 presents speedups achieved by our design over a uniprocessor system for a collection of SPEC2000 integer and floating point applications. Four bars are shown for each benchmark. The first bar shows the unoptimized base configuration. The following two bars correspond to speedups of profligate execution in two and four processor CMPs. The final bar represents speedup of an overly aggressive machine with a 4096 entry ROB and serves as a pseudo-upper bound on performance.

We observe an average SPECINT speedup of 17% for four cores, which is 75% of the speedup attainable with a 4k-entry ROB. Not surprisingly, the benchmarks that exhibit the most speedup also suffer from a large number of cache misses and full window stalls. The SPECFP benchmarks obtained an average speedup of 212%, which is nearly half of the speedup attainable with a 4k-entry ROB. The high cache miss rates exhibited by these applications as well as the existence of long-latency floating point operations make their performance highly sensitive to window size. For example, the benchmark *swim* accesses main memory every 43 instructions on average, and profligate execution enabled a speedup of over 10x.

Figure 5 also shows the degree that each additional core speeds up execution. Because one additional core is sufficient to uncover most miss-independent instructions, the first bar shows the most significant improvement. Adding more cores helps by executing those instructions dependent on misses that were skipped by other cores. Finally, Figure 5 shows that in addition to reducing the latency of a single application’s execution, profligate execution can also improve overall system throughput. A traditional multiprocessor with  $n$  threads can at best increase throughput by a factor of  $n$ . However profligate execution achieves *super-linear speedup* for six of the 11 floating point benchmarks (*facerec*, *galgel*, *lucas*, *mgrid*, *swim*, *wupwise*). In these cases adding a second core reduces overall execution time by a factor greater than two. It shows that two cores cooperatively executing a single thread can expose more MLP than two cores executing two independent threads.



### 3.4 Processing Element Size

Figure 6 takes a closer look at the performance of one particular benchmark -- *art*. It graphs instructions per cycle for 1, 2, 4, and 8, processor CMPs with varying local ROB capacities of 32, 64, 256, and 512 entries. Clearly performance improves as both the number and size of the windows grow. However it also shows that a machine with several smaller windows can outperform one with a single large window. For example, eight 32-entry windows or four 64-entry windows perform equivalent to a single 256-entry instruction window. Furthermore, because smaller windows are more amenable to high-frequency design, it may be possible to not only increase the number of instructions committed per cycle, but also the decrease the cycle time itself.

#### 3.4.1 Alternative Design Point: Simpler Processor Cores

The difficulty in scaling the necessary data structures required to support large instruction windows has led towards a shift in server system design. Rather than building large cores that are decreasingly effective at extracting ILP from programs, some manufacturers are instead focusing on thread-level parallelism (TLP) and overall system throughput. While such systems can provide high throughput in workloads with abundant TLP, they typically cannot achieve the same single-thread performance as more traditional processors capable of supporting dozens or hundreds of in-flight instructions. Profligate execution can provide a mechanism to boost single-threaded performance in such

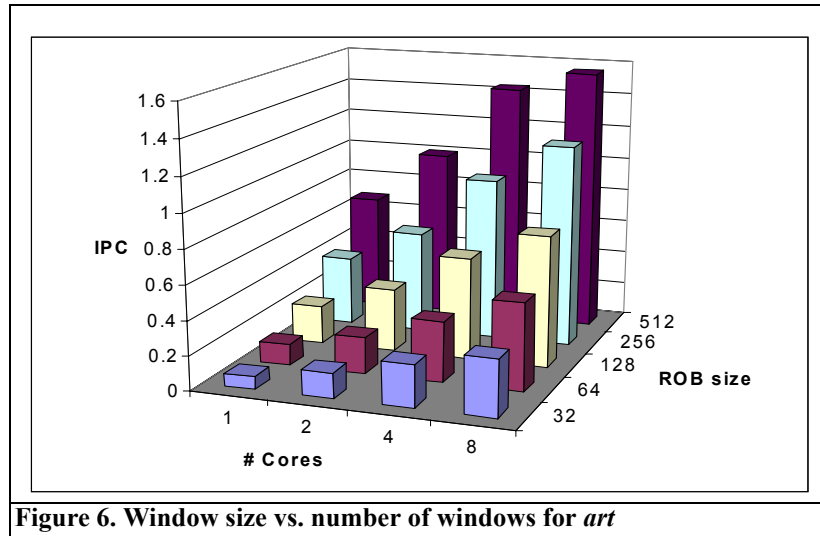


Figure 6. Window size vs. number of windows for art

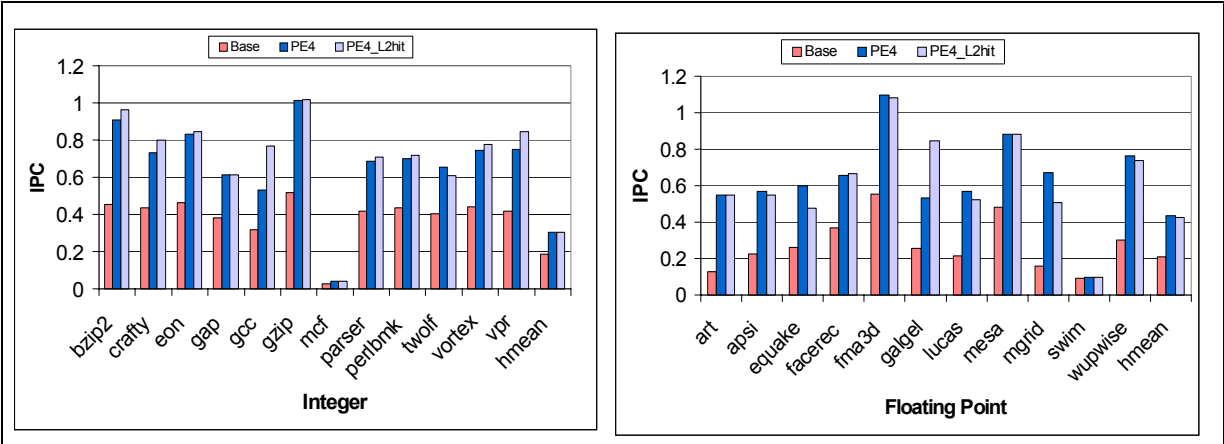
systems.

To gauge the effectiveness of profligate execution in such a design, we modeled a machine based on Sun’s Niagara processor, which contains eight in-order 4-way multithreaded cores on a single die. Additional parameters are presented in Table 2, and IPC speedups for SPEC2000 benchmarks are shown in Figure 7. The first bar graphs the IPC of the baseline processor. The second bar shows instruction throughput of a 4-core profligate execution system (which effectively uses half of Niagara’s execution resources) over the baseline uniprocessor case. Because these smaller cores are less effective at extracting ILP and MLP on their own, they can achieve dramatic performance improvements (63% speedup for integer benchmarks, and 105% for floating point). While still considerable, the floating point speedups are less dramatic than those in Figure 5 corresponding to the more aggressive processor. We attribute this to the fact that the Niagara configuration has only a single FP ALU per chip which is shared among all the processors. As profligate execution enables more simultaneous in-flight instructions, performance becomes limited by the FP ALU’s execution bandwidth. We would expect greater speedups in similar processors tuned for floating point performance, such as Sun’s forthcoming Rock CPU.

The third bar shows instruction throughput if we discard instructions that not only depend on L2 misses, but L2 hits as well. Because the simpler in-order cores cannot mask the full L2 hit latency, it makes sense to allow them to throw out L2 hits and their dependent instructions. While partitioning L2 hits in the more aggressive processor configuration results in slowdown compared to partitioning only L1 misses (data not shown), it speeds up several of the benchmarks on in-order cores. This is due to the fact that out-of-order execution mechanisms are more effective at hiding L2 hit latency, and partitioning L2 hits in that case only increases communication through the GROB and GRF. We therefore only consider this policy in the context of this machine configuration.

### 3.5 Virtual Window Size

Figure 8 plots the virtual window size vs. time for the integer benchmark *gap*. It shows the dynamic instruction distance between the oldest and youngest in-flight instructions among all cores, and indicates the degree that processors can run ahead of each other. In a traditional uniprocessor the virtual window size is the same as the physical instruction window size, and is bounded by the capacity of the ROB. Conceptually, it represents the number of in-



**Figure 7. Speedup with Simple In-order cores.** The two bars for each benchmark indicate instructions per cycle for four-core Profligate Execution that partitions L2 misses and L2 hits

flight instructions a traditional uniprocessor would need to sustain in order to expose an equivalent degree of parallelism.

We observe that even though our baseline processors can only support a maximum of 64 in-flight instructions each, profligate execution can allow processors to execute thousands of instructions ahead into the future. In our machine model this distance does not generally exceed 2000 instructions (500-cycle DRAM access \* 4-IPC peak bandwidth), however as clocks speeds continue to outpace memory latencies this span will increase and can make profligate execution a more attractive design alternative.

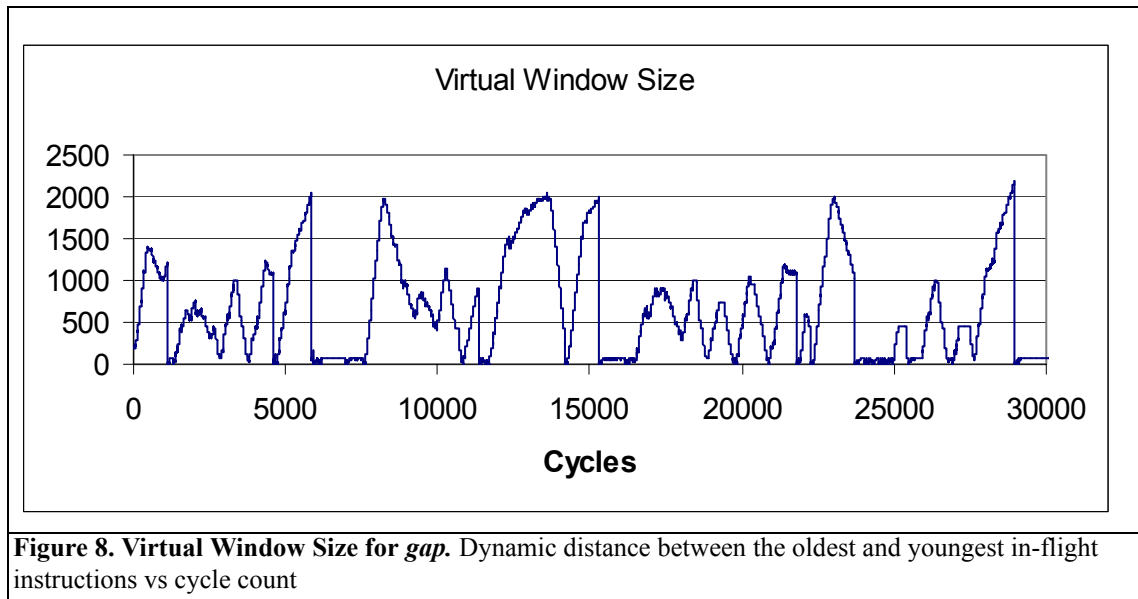
Figure 8 also shows that, while large instruction windows are helpful in dealing with cache misses, they are not always required. The virtual window size falls below 64 instructions at some times, indicating that a single baseline processor’s 64-entry ROB is adequate to expose sufficient parallelism. This is an advantage of profligate execution over other large-window techniques, which uncover parallelism at the expense of larger and more complex cores. A reasonable extension of this work would be a hybrid mode of operation, where only a single core normally executes an application in the absence of cache misses (leaving the others free to execute independent threads), but other cores cooperatively form a larger virtual window when it would be beneficial. The fact that cache misses tend to cluster together in many applications further support that such a mode could be useful. While we leave in-depth exploration of hybrid execution to future work, we also believe that this adds significant appeal to profligate execution.

### 4.0 Related Work

The problem of finite sized instruction windows in the face of a growing processor-memory gap is a substantial problem in modern computer architecture and has solicited considerable attention. Two related techniques to expose MLP -- runahead execution and dual core execution -- are quantitatively compared against in this section. We also provide a short survey of other relevant work in this area as it relates to profligate execution.

#### 4.1 Runahead Execution

Runahead execution [11][22] is a prefetching technique designed to exploit memory-level parallelism. When a miss to main memory occurs, the architected state of the processor is checkpointed and any destination regis-



ters that directly or indirectly depend on cache misses are marked invalid. Instructions with one or more invalid source operands are discarded by the processor and will not block commit. This allows instructions that would otherwise wedge the ROB to pass through the pipeline unimpeded, thereby enabling the processor to reach distant load misses that are independent of the initial miss. We modeled runahead execution and present its IPC in Figure 10. Runahead execution is similar to profligate execution in that both attempt to start independent cache misses early by discarding long-latency instructions. However there are several important differences that help explain why profligate execution exhibited larger speedups for all benchmarks:

**Runahead can only exploit MLP, not ILP:** The instructions executed in runahead mode serve only as prefetches; they must be re-executed when the miss returns and architected state is restored. Profligate execution, on the other hand, does not need to re-execute, and can achieve true large-window ILP benefits.

**Runahead is limited by miss latency:** Runahead can only issue prefetches for load misses that it reaches while the miss that caused the transition to runahead mode is outstanding. Once the initial miss returns, execution is restarted at the next instruction. Because profligate execution does not replay instructions executed in the miss shadow, processors can still make useful progress during the miss even if they do not reach subsequent misses before the initial miss returns.

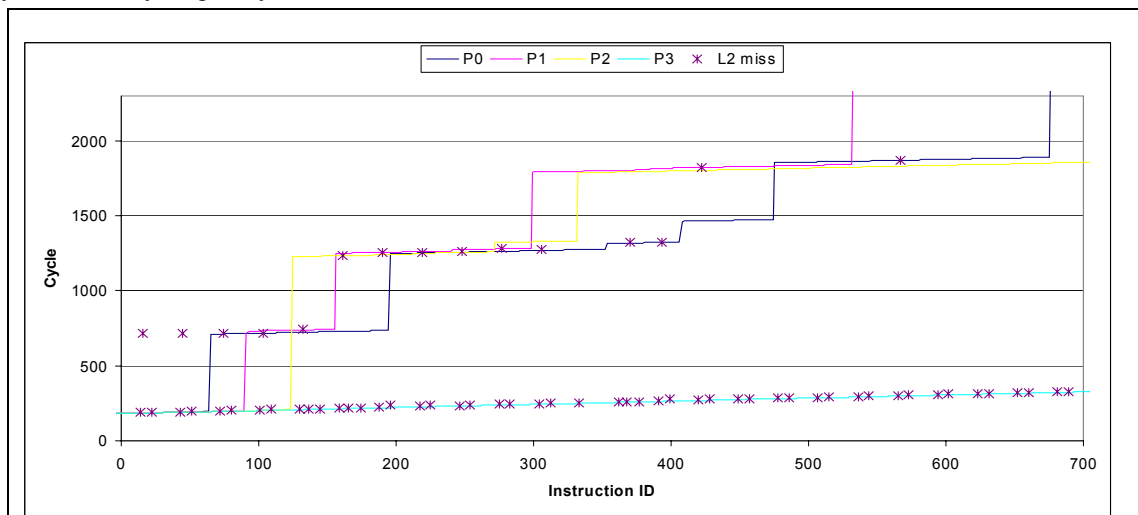
**Dependent Misses:** Runahead mode will prefetch misses independent of the initial miss, but neither they nor their dependent instructions will execute until the initial miss completes. However due to non-uniform memory latency caused by a growing number of levels in the memory hierarchy, open/closed DRAM pages, and a variety of other factors, it is possible for younger independent misses to complete before the initial miss. Therefore any of their dependent instructions (including misses) will not execute until exiting from runahead mode.

Figure 9 depicts how dependent misses are handled by profligate execution. Each point represents a dynamic instruction in a selected segment in the benchmark *mcf*. The X-axis indicates the instruction identifier, in program order. The Y-axis indicates the cycle that instruction was dispatched by each of a system's four cores. The

slope of each line, therefore, represents each processor's rate of execution in terms of cycles per committed instruction. Discontinuities in the graph represent periods of time in which no useful instructions are dispatched (from full-window stalls, I-cache misses, or branch mispredictions)

Processor P0, P1, and P2 are occasionally assigned miss slices to execute, and suffer full-window stalls that prevent them from fetching new instructions. However because P3 is designated the lead processor, it never stalls and only prefetches misses, as indicated by its constant slope. The final data series (depicted by X's) represents the time in which cache misses are initiated. As can be seen, most of these misses are prefetched by P3 and are thus independent of prior misses. Those that do not fall along P3's trajectory are executed by trailing cores and therefore depend on older misses.

Runahead execution handles dependent misses by restarting the runahead thread when the initial miss completes and allowing it to catch up to the skipped instructions. As Figure 9 shows, the lead processor in profligate execution (P3) always runs ahead and never restarts, and the responsibility of executing instructions it skips belongs to the trailing processors. If there are a large number of dependent misses, adding more processing elements increases the system's ability to quickly execute them.



**Figure 9. Dependent Misses.** Time at which each processor dispatches each instruction in *mcf*. 'X's indicate time when cache misses are initiated.

## 4.2 Dual-Core Execution

Runahead Execution's primary shortcoming is that its reach is limited to the product of the memory latency and the processor's maximum issue width. Any future misses separated from the initial miss by more than this number of instructions will never be prefetched. Zhou proposed Dual-Core Execution [29] in response to this limitation by re-locating the runahead thread to a separate processor core in a CMP. Instead of squashing and re-executing instructions, the runahead thread is able to continue making forward progress even after the miss completes. However it still is unable to handle the case of dependent load misses. Loads with invalid address registers are skipped by the speculative front processor and are not re-executed until they reach the back processor via a result queue. Because several thousand instructions may separate the two cores, cache misses may not start until much after their operands







