A Novel Architecture for Large Window Processors

Isidro González	Marco Galluzzi	Alex Veindenbaum	Marco Antonio Ramírez	Adrián Cristal	Mateo Valero
DAC – UPC	DAC – UPC	ICS – UCI	CIC – IPN	BSC – CNS	BSC – CNS
iglez@ac.upc.edu	galluzzi@ac.upc.edu	alexv@ics.uci.edu	mars@cic.ipn.mx	Adrian.cristal@bsc.es	mateo@ac.upc.edu

< This Technical Report was sent to Advisory Committee of ISCA-35 (November 19th, 2007) for review >

Abstract

Several processor architectures with large instruction windows have been proposed. They improve performance by maintaining hundreds of instruction in flight to increase the level of instruction parallelism (ILP). Such architectures replace a re-order buffer (ROB) with a *check-pointing* mechanism and an out-of-order release of the processor resources. Check-pointing, however, leads to an imprecise state *recovery*[*] on mis-predicted branches and exceptions and frequent re-execution of correct-path instructions during the state recovery. It also requires large register files complicating renaming, allocation and release of physical registers.

This paper proposes a new processor architecture, called a *Multi-State Processor (MSP)*, that does not use either a traditional ROB or check-pointing, avoids the above-mentioned problems, and has a fast, distributed state recovery mechanism. Its novel register management architecture allows implementation of large register files with simpler and more scalable, register renaming and commit. It is also key to the precise recovery mechanism. The MSP is shown to improve IPC by 15.5%, on average, compared to a check-pointing based mechanism ([2]) for the integer SPEC CPU2000 suite. More precise state recovery results in a 16.5% reduction in the number of executed instructions. The MSP processor is thus more energy efficient.

1. Introduction

Recently proposed large instruction window processors, such as *Kilo-instruction Processors* [1] and *Check-point Processing and Recovery* (*CPR)* [2, 15], allow thousands of in-flight instructions to uncover distant ILP and mask long memory latencies. They use check-pointing mechanisms, which

[*] In our context, the *precise recovery* means restoring the processor's state to the exact instruction producing an exception or a branch mis-prediction. Check-pointing processors, even those supporting *precise interrupts*, do not always recover the state to the exact point of the exception or branch mis-prediction but at the point where the previous checkpoint has been set.

allow the release of resources associated with each instruction as soon as the instruction has been successfully executed. This allows large instruction windows to be implemented with a *tolerable* increase in required processor resources.

Check-pointing mechanisms define a checkpoint as a hardware structure containing the information necessary to recover a processor's state. An exception or a branch mis-prediction lead to restoration of the processor state to a previous check-point and re-execution from the check-point. In general, the components of a state include physical register values, mapping of logical to physical registers, and pending stores. However, a check-point only stores the register mapping, with processor releasing registers only when a check-point commits. This makes restoring register state relatively simple and fast [18], but requires more physical registers. The pending stores in a store queue are handled separately and require a more complex and time consuming mechanism. The time delay of scanning a large, 2^{nd}-level store queue [2] for state roll-back can be significant.

The performance of this type of processor depends on the available resources, e.g. register file size, store queue size, instruction queue size, etc and on the check-pointing mechanism itself. How well the latter functions, depends on the number of available checkpoints and on check-point placement along the program execution path. Typically, a new checkpoint is created at branches with high mis-prediction rates or at other instruction likely to produce an exception. Several different check-point management mechanisms have been proposed [19]. For instance, one of them is based on a confidence estimator that computes the confidence for every branch prediction done. A new check-point is created if the estimator gives a low confidence for the prediction of the current branch [6], assuming there is a free check-point.

When all instructions between the oldest and the second oldest check-points have successfully executed and thus could not be discarded due to a recovery, the oldest check-point can be released. At this time all instructions between the oldest check-point and the next one are committed, potentially requiring a large number of instructions to be committed simultaneously.

When an exception or branch mis-prediction occurs, the processor rolls back the state to the youngest checkpoint preceding the exception-causing instruction. On return from the exception, execution cannot resume at the excepting instruction. Instead, it has to resume at the preceding check-

point and therefore will not be precise. This will require re-execution of a number of instructions, which were correctly executed. There can be a significant amount of instructions to re-execute depending on the number of check-points and the check-point management mechanism [18, 19]. This lack of precision in branch mis-prediction or exception recovery degrades processor performance and increases power consumption. Increasing the number of check-points to minimize the impact of imprecise recovery is undesirable due to hardware costs of check-pointing [18] and the hardware delays it may introduce. Also, increasing the number of check-points does not guarantee an improvement in performance. Thus a new solution is necessary to avoid this loss of performance due to imprecise recovery.

Another problem with large-window, check-pointing processors are that they require a large number of registers which complicates register management: renaming of a logical register, allocating a physical register, freeing a physical register, and recovering from branch mis-predictions and exceptions. Consider renaming, for instance. Many modern processors use a CAM-based structure, which stores a physical to logical register mapping. Thus for a processor like the CPR with 192 registers, the required CAM is large resulting in an increase in access time and energy consumption. Even larger register files may be desirable for very large instruction windows. Also, tracking when all the uses of a physical register have occurred and it can be released complicates things. For instance, CPR used reference counters [9] to release physical registers which can improve performance. Counters are easy to update but introduce additional complexity in recovery when instructions are squashed. Last but not least, wider issue width requires wider renaming, such as in IBM Power4 [23], which is harder to implement, has high power consumption, and is a thermal hot-spot which can lead to hardware faults. This calls for a new mechanism to more efficiently manage large register files.

The architecture proposed in this paper solves both of the above problems, allowing precise recovery and efficient management of a large register file in a unified approach and without using either check-pointing or a traditional ROB. It is called a *Multi-State Processor (MSP)*. MSP assigns a state to instructions in flight and defines an efficient and scalable state management mechanism for instruction commit and branch mis-prediction or exception recovery. A new state is created on every instruction assigning a register, with adjacent states differing by at most one change in the register

state. This allows fast register state recovery. Instructions not assigning a register, such as branches, become associated with "current" state.

In terms of physical register management, MSP proposes a new, scalable register renaming mechanism, which eliminates traditional, physical register free list and Register Alias Table (RAT) and integrates a mechanism for releasing a physical register based on its use by dependent instructions. It is also integrated with state management and commit. The proposed mechanism has the potential to reduce the high power density and overall power dissipation of a renaming unit. Furthermore, it allows the register file itself to be banked in a novel way, reducing the bank port requirements.

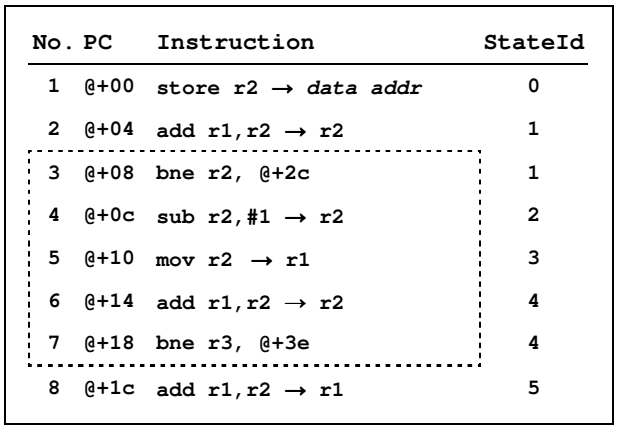
The rest of the paper is organized as following. Section 2 introduces our definition of processor state and state management. Section 3 describes the micro-architecture of the MSP. Section 4 presents the state recovery mechanism and Section 5 describes how to implement limited-size state identifiers. Section 6 presents the performance evaluation of the MSP and compares it with the CPR processor. The paper concludes by discussing related work and summarizing our results.

2. Processor State Management

This section describes state definition and management of the proposed MSP architecture. Let us start with an example of a dynamic instruction sequence shown in Figure 1 and discuss its execution on a check-pointing processor to motivate the state management of the MSP. *(Ignore for now the column labeled StateId in the figure*).

Assume that a check-point is set at instruction 3, a branch with a low-confidence estimate. However, this branch is predicted correctly. But a branch mis-prediction occurs at instruction 7, where a check-point has not been set. As part of branch mis-prediction recovery, the processor's state is restored to the state stored in the previous closest checkpoint –the one set at instruction 3. Execution resumes from there and the processor re-executes instructions from 3 to 7 (shown inside the dashed box in Figure 1), even though they were already correctly executed.

In summary, restoring processor state to a check-point causes all instructions in the pipeline that are younger than the check-point to be squashed. It also restores the logical to physical register mapping, adds released physical registers to the free list, and releases possible younger checkpoints.



No.	PC	Instruction	StateId
1	@+00	store r2 → *data addr*	0
2	@+04	add r1,r2 → r2	1
3	@+08	bne r2, @+2c	1
4	@+0c	sub r2,#1 → r2	2
5	@+10	mov r2 → r1	3
6	@+14	add r1,r2 → r2	4
7	@+18	bne r3, @+3e	4
8	@+1c	add r1,r2 → r1	5

Figure 1. Example of a dynamic instruction sequence

2.1. MSP State and State Management

The goal of the MSP processor is to restore processor state precisely to the desired instruction. To achieve this the MSP processor defines a processor *state* to correspond to a register assignment. It assigns a new state to each instruction that writes a destination register. Thus the difference in state between two adjacent instructions is in at most the state of one register. The allocation, recovery and release of a state in the MSP processor are thus strongly tied to register management. Stores to memory are dealt with separately via the store queue.

When an instruction is added to the instruction window the following actions take place:

1. *The processor state is assigned a value called a StateId.* The StateId is maintained by a binary counter, which is incremented only if the current instruction assigns a logical register (allocates a new physical register). The StateId counter is a modulo 2*log(# of registers) counter.
2. *The current value of the StateId is associated with the current instruction entering the window.* Note that instructions not assigning a register, such as branches or stores, get the same StateId value as the previous instruction.
3. *For each physical register* a range of consecutive states, called a StateId Range, in which the register is valid is updated and maintained. The physical register StateId range is defined as follows. The Lower StateId of the range is the StateId of the instruction causing the allocation of the physical register. The Upper StateId of the range is the StateId of the instruction preceding the instruction that renames the corresponding logical register.

On state recovery, instructions with a StateId greater than the StateId of the instruction causing the recovery are discarded. The StateId range allows identification of all instructions (states) using a given physical register.

For instructions in Figure 1, the assigned StateIds are shown in the column labeled "StateId" of the figure. The StateId range associated with each physical register is shown in Figure 2 using the notation R*x.y* : register version *y* of a logical register *x.* Thus R2.0 and R2.1 are two instances of the logical register R2, they correspond to two physical registers allocated on two consecutive assignments to R2.

StateId Range		Associated registers	
Lower	Upper	Logical	Physical
0	0	R2	R2.0
1	1		R2.1
2	3		R2.2
4	5		R2.3 *
0	2	R1	R1.0
3	4		R1.1 *
5	5		R1.2

Figure 2. StateId Range for instructions in Figure 1

The R*x.y* notation used above for describing a physical register reveals a key idea of our register management mechanism. MSP divides a physical register file into banks and assigns each bank to a given logical register. It then uses a distributed mechanism to manage physical registers in each bank. For instance, renaming a destination register becomes a local operation in each bank. Banking also allows efficient implementation of large register files.

The MSP mis-predicted branch recovery proceeds as follows (for instruction 7 in Figure 1). The MSP sets the *Recovery StateId* (explained in more detail in Sec. 5) to the StateId associated with this branch instruction, i.e. to state number 4. All instructions with a StateId greater than 4 are squashed. All physical registers whose Lower StateId is greater than 4 can be released –only the register R1.2 in the example, and become available for renaming of future instructions.

3. Micro-architecture of the MSP processor

The micro-architecture of the MSP processor and its pipeline are shown in Figure 3. The micro-architecture uses a banked physical register file with 1 read and 1 write port per bank, which requires arbitration to detect read and write conflicts. MSP thus adds an arbitration pipeline stage. The results in this paper show that the IPC degradation due to this new pipeline stage is not significant. The power and area of the banked register file are significantly less compared to a flat register file organization.

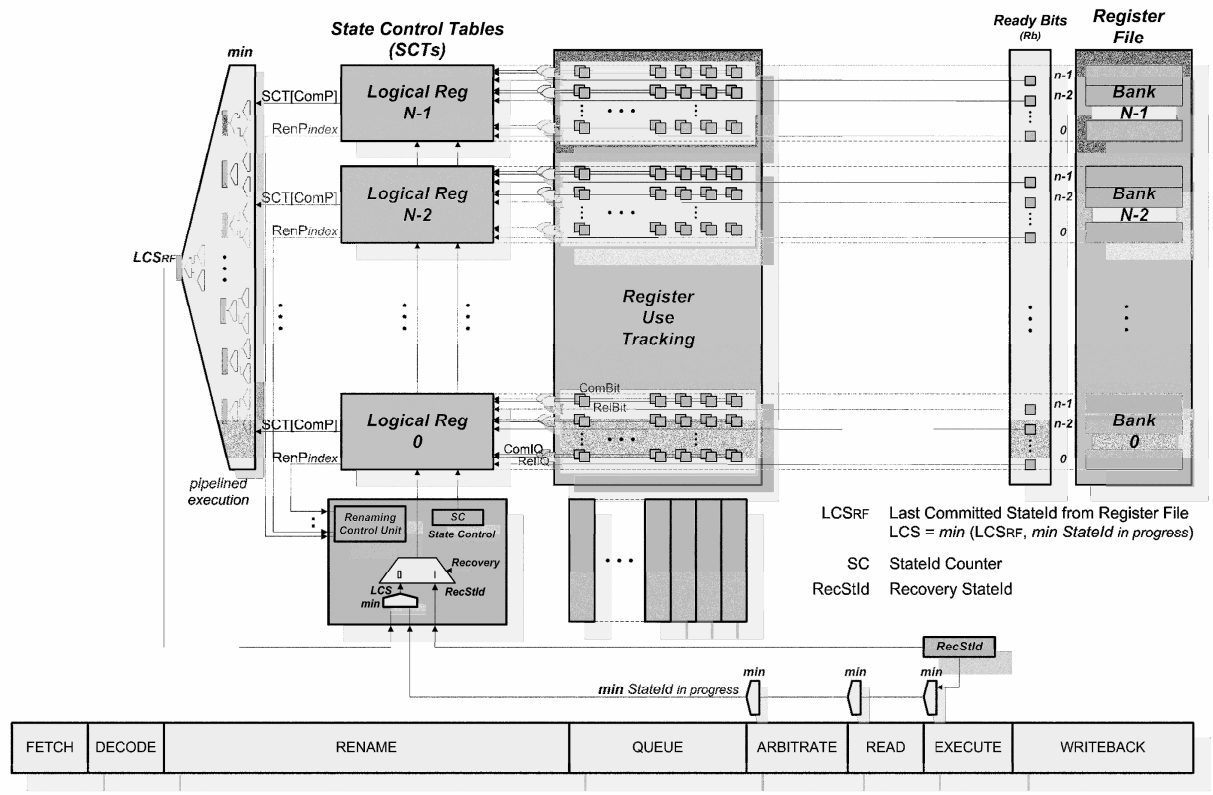


Figure 3. The MSP micro-architecture

3.1. Register Renaming and State Id management

To implement a distributed renaming mechanism efficiently the MSP imposes the following constraints on allocation and renaming of logical registers:

1. Each logical register is renamed to a fixed subset of physical registers (its bank)

2. Physical registers are allocated and released in order within a bank of a logical register

This significantly simplifies the renaming mechanism for a large physical register file. The FIFO allocation policy is used for a new physical register in each logical register bank. A global free list of physical registers is no longer required and neither is a global Register Alias Table. The maximum number of states possible in the MSP is equal to the number of physical registers. Typically, about

30%[†] of instructions do not generate a new state. Thus MSP can have at most 1.3 x *M* instructions in flight, where *M* is the number of physical registers. However, MSP can manage a larger register file than other architectures.

3.2. MSP operation

The control of the MSP registers and state can be divided into a) a *local scope control* for each logical register bank, and b) a *global scope control* interacting with the rest of the processor and between banks.

Local Scope Control

The Local Scope Control shown in Figure 4 performs local control of a logical register bank. It consists of a State Control Table (SCT) plus rename, commit, and release pointers, and their associated logic.

An SCT entry is associated with a physical register and contains the following fields:

- *StateId:* the value of the *Lower StateId,* initialized with the StateId of the instruction assigning a destination operand. Since the StateId values are incremented in program order, the *Upper StateId* is implicit – it is the value of the next SCT entry minus one. For the most recent entry (last renaming) the *Upper StateId* is a null value.
- *Valid Bit (Vb):* specifies whether the entry is in use. There is always at least one active entry, which is the last renaming of the logical register.

The control logic associated with each entry consists of:

- *Range StateId Comparator.* It compares the StateId range of the entry with a StateId broadcast by the processor. The comparator is used for physical register release either on instruction commit or during mis-prediction/exception recovery and to advance the RelP.
  - Logic to detect if an instruction has written this register result and logic to detect if this value has been consumed by all dependent instructions. The commit process and the release of physical

[†] This value (32.14%) is the average of the correct-path instructions in integer SPEC CPU2000 suite which do not generate a new state in the *ideal MSP* simulator (see Section *Performance Evaluation*)

registers for committed instructions is a continuous process and instructions are committed in the StateId order.

- Recovery logic which receives the globally broadcast *Recovery StateId* and detects if a register mapping needs to be released (see Sec. 5 for more detail). A register is released if its *StateId* > *Recovery StateId.*

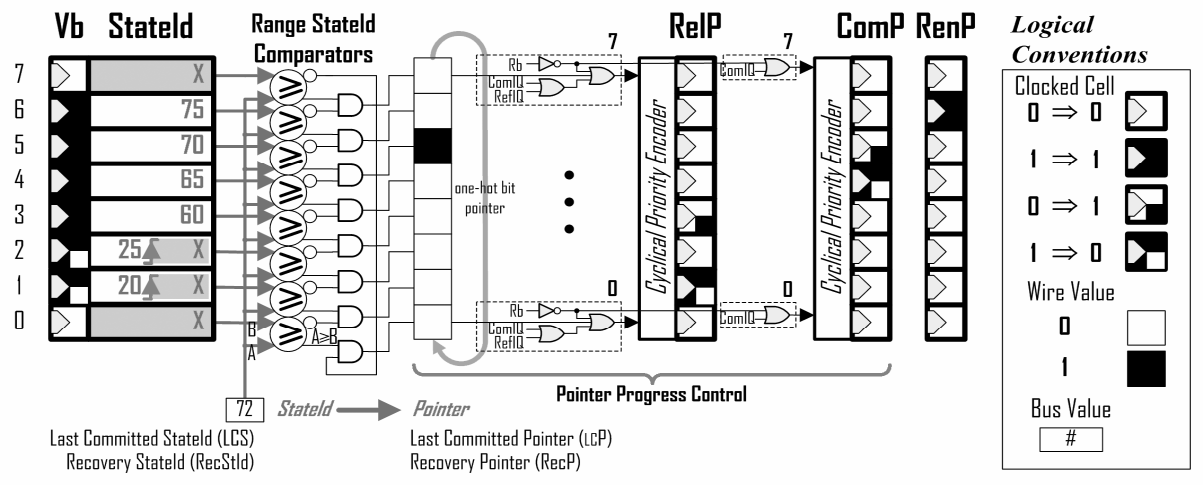


Figure 4. Local Scope Control

Three local pointers are associated with an SCT. Let us assume that they are implemented as *one-hot* bit vectors using circular shift registers, but other implementations are also possible.

- *Rename Pointer* (*RenP):* points to the last entry allocated in its SCT, which corresponds to the most recent renaming of the associated logical register. On a new renaming, the pointer will be shifted by one position to the next spatially adjacent entry. The current mapping of the associated logical register to a physical register is the logical register identifier (*LogRegId,* defined in section 3.2) and the RenP index pair.
- *Commit Pointer* (*ComP):* points at the potentially committable StateId. It is the oldest entry in the SCT that has executed and produced a value in the corresponding physical register (Ready bit Rb=1). It takes part in the global computation of the *Last Committed State* or *LCS* described below.
- *Release Pointer* (*RelP):* points at a register that can be released. The value in this register has been consumed by all its dependent instructions. All the instructions associated with this StateId have

executed. The StateId[RelP] < LCS and this is the oldest entry in this SCT. This enforces the FIFO physical register release policy.

In each SCT/Local Scope Control:

A) the range of states in entries between the RelP and RenP pointers is the set of active states for a logical register. The SCT[RelP] has the oldest StateId and SCT[RenP] has the most recent StateId.

B) the range of states in entries between the ComP and RenP pointers is a subset of the range in A) above. That is, RelP never goes ahead of ComP and ComP never goes ahead of RenP.

C) All the three pointers may point at the same entry, the current logic register mapping.

D) All physical registers in a bank have been allocated if the RenP and RelP point at adjacent entries.

Global Scope Control

The global scope control maintains the current state of the processor and determines the most recent non-committable state. This includes:

- *The StateId Counter* (*SC),* defining the current processor StateId. It is incremented for each decoded instruction renaming a logical register.
- *The Last Committed StateId* (*LCS) unit,* continuously computing the minimum (oldest) StateId of all SCT[ComP] entries. The LCS is the oldest state in the MSP that can be committed. When a new LCS is computed it may make multiple older states eligible to commit. *Several instructions may belong to one state and the state can only be committed when all of them have executed. This is why the state of instructions issued but still in the pipeline is tracked in Figure 3.*

An additional mechanism is used to release processor resources associated with commited instructions/states: registers and Store Queue entries. The logic in the local scope control of each logical register uses dependent instruction progress information to advance the RelP pointer, and thus release the associated physical registers. The Store Queue logic uses LCS to store to memory entries with a StateId older (smaller) than the LCS.

The number of SCTs is equal to the number of logical registers, typically 32, and the StateId is 9bits for a 256-entry physical register file (8 plus an “overflow” bit explained below). Thus the

hardware needed to compute the LCS is a five-level binary tree of comparators and multiplexors. Each comparator finds the smallest of the two StateIds at its inputs and passes it through to the next level. This computation may take multiple clock cycles but can be easily pipelined to produce a new minimum every cycle. However, latency of LCS computation is not a critical timing issue, our investigation showed that even a 4-cycle LCS computation degrades performance by less than 1% compared to a 1-cycle computation.

A deadlock condition occurs if (RenP=ComP and Rb[ComP]=1), i.e. all possible renames of a register have been produced. The StateId of the SCT entry pointed by both ComP and RenP is not used in the computation of the LCS.

Also, if there is only one current renaming of the logical register and its value has been produced, the register cannot be released and can thus exist after its instruction has committed.

3.3. Renaming of Multiple Instructions

The renaming process is complicated by the fact that multiple instructions may assign the same destination logical register in one clock cycle. For instance, a processor capable of issuing four instructions per cycle may need to rename the same logical register four times in one cycle. Our analysis of the impact and frequency of occurrence of such multiple renaming in the same cycle showed that renaming at most two instructions assigning the same logical register per cycle is sufficient. Allowing three or more such instructions to be renamed per cycle does not improve performance. However, allowing only one to be renamed leads to a 4.9% reduction in IPC. Therefore, the renaming logic discussed in this section allows up to four destination registers to be renamed per cycle, two of which can be the same logical register.

Figure 5 shows a block diagram of the renaming logic for a logic register (per SCT). This logic is enabled by a logical register identifier, *LogRegId,* of one of the registers to be renamed. There can be at most four SCTs activated in a cycle (assuming an issue width of 4). In an activated SCT the entry to be written by a new renaming is pointed to by the RenP. The renaming logic also generates the "next RenP" value, which is used if the associated logical register is renamed again in the same cycle. The next RenP bit vector is a logical shift of the current RenP bit vector by one or two positions. A port

decoder identifies the write ports to use, up to two, using the new values of the RenP. Finally, the StateIds to be written into the selected entries are the StateIds of up to two new instructions being renamed. These StateIds are computed by adding the current StateId (the value of the SC counter) and the *SC offset* of each instruction generating a new state. The SC offset is the position of the instruction in the current set of four being renamed (only two of which can be in the given SCT). The figure shows an example of two instructions being renamed, first and last in this group of four. The SCT is assumed to use one write port per entry and two multiplexors are used to select the two computed StateIds to write. A stall is generated if there are more than two instructions renaming the register.

A logical register source operand of an instruction is renamed to a bank specified by the logical register number, LogRegId, and the position in the register bank specified by the *RenP* pointer index. Additional logic detects if the updated RenP value needs to be used.

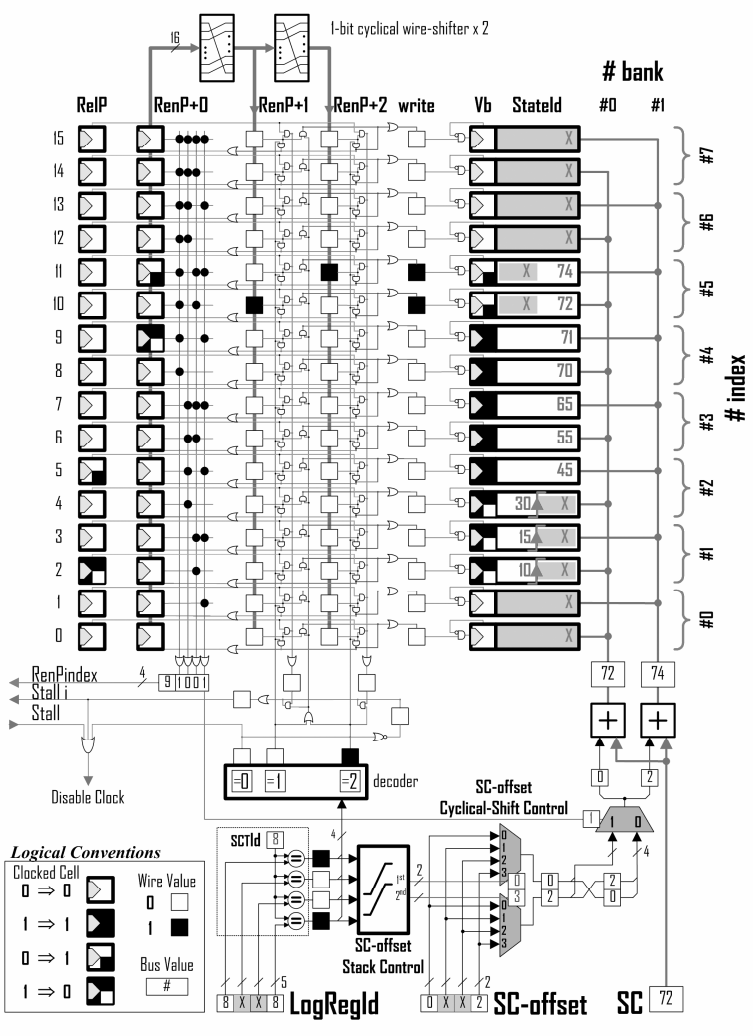


Figure 5. Renaming logic for one logical register

Renaming may cause a stall of the front-end, i.e. of all stages prior to Rename, if a bank does not have enough registers for renaming. Such a stall is detected by an SCT by comparing pointers and is broadcast to all other SCTs and to the global renaming control unit. The stall control logic prevents advancing of RenP pointers in other SCTs, which rename younger instructions in the same cycle.

3.4. Tracking Register Use

MSP needs to track when the last use of a register occurs and also when all instructions associated with a register state have completed execution. While this could be done with reference counters [9], MSP proposes a different solution using bit vectors (also independently proposed in [21]).

A bit vector *RelB* of the size equal to the instruction queue size, is used to track dependents. During renaming of source operands the bits of this vector corresponding to dependent instructions are set to "1". As instructions are issued, they reset the corresponding bit of the *RelB* for each source register.

Similarly, a *ComB* bit vector is used to track instructions that belong to the corresponding register state but themselves do not assign a destination register. The state can only be retired when all such instructions consume their operands and issue. Associated with each physical register is a Ready bit, *Rb,* set in the WriteBack stage. An SCT entry/register can be released when it is ready, Rb=1, and the *RelB* bit-vector of all "0"s. and is the oldest entry in the SCT. Rb=1 and ComB=0 are to used advance the commit pointer ComP (see Figure 4).

Mis-prediction or exception recovery resets all bits in a column of *RelB* and *ComB* vectors corresponding to the position of the source operands of cancelled instructions.

4. A State Recovery Mechanism

The state recovery mechanism is invoked on a branch mis-prediction or an exception. The *Recovery StateId* register is set to the StateId of the state to which the processor needs to recover to. The recovery actions depend on whether recovery is due to a branch mis-prediction or an exception.

On a branch mis-prediction, the processor state is reset to the StateId of the branch instruction. All instructions in the IQ following the branch are squashed and their associated *ComB* and *RelB* bits cleared. The front end is restarted with a branch target PC. The *Recovery StateId* is broadcast to all SCTs and all physical registers with a StateId greater than the *Recovery StateId* are released.

Exceptions are recorded in the WriteBack stage. A detected exception cannot be dealt with immediately, in case there is an exception on an earlier instruction since exceptions need to be taken in order. However, detection of an exception in the MSP immediately stalls the front-end and only instructions with a StateId smaller than the StateId of the excepting instruction are issued to execution. Once the instruction causing the exception is committable (becomes oldest), the exception is actually taken. Any younger instructions are cancelled. Also, the Recovery StateId for exceptions is the StateId associated with the instruction causing the exception or the StateId of the previous one if this instruction produced a new state. Similar to branch mis-prediction recovery, multiple instructions associated with a single state have to be dealt with correctly. After the recovery is complete, the SC is set to the Recovery StateId and the *Recovery StateId* is disabled.

5. StateId size

Any implementation of the State Counter SC has a certain number of bits and will eventually overflow the counter. This is a problem because the StateId serves to record the chronological order of states during the execution. Also, the StateId is stored in each SCT entry and determines the bus and comparator sizes. Thus a smaller StateId is desirable from a hardware point of view.

The number of physical registers is equivalent to the maximum number of states in flight. With *M* physical registers the maximum number of states active at any given time is also *M*. Thus the StateId size is *log2(M)* = *m* bits. Therefore, some action is required when SC value reaches M-1. One could stall the front-end until all instructions already in the pipeline finish execution, reset the SC and then restart the execution. But this would reduce processor performance significantly. The solution is used in the MSP is a *saturation bit, Sb,* added as the most significant bit to the *log2(M)-bit* StateId to control the overflow.

The SC, which now has *m+1* bits and can encode up to 2x*M* states, is initialized with zeros and is incremented until it reaches the maximum value of all "1"s. Since there are at most *M* states in flight, all current states must now have the Sb set to 1. At this point the Sb bits of all stored StateIds are reset to 0 and the SC is set to value *M+1,* that is, the Sb to 1 and the rest of the bits to 0. The process is repeated every time the SC reaches its maximum value.

6. Performance Evaluation

The proposed architecture was evaluated and compared with a baseline architecture, a large-window architecture (CPR), and a version of MSP with unlimited resources. The following gives a brief summary of each architecture.

- *Baseline*. A reasonably standard out-of-order, single-thread, superscalar processor.
- *CPR*. An architecture without an ROB using a selective check-pointing mechanism, a hierarchical store queue, and aggressive release mechanism for physical registers. It has a flat register file with all required ports and does not use the arbitration stage in the pipeline.
- *n-SP*. The Multi-State Processor architecture with *n* physical registers per logical register. It uses the same hierarchical store queue as the CPR architecture.
- *ideal MSP*. MSP with an infinite hierarchical store queue and an infinite, fully-ported register file.

The parameters of the four architecture are shown in Table 1, many of them were chosen to be identical to those used in CPR processor [2]. A notable difference with CPR is the branch predictor used in this paper: it is either a *gshare* or a *perceptron* predictor.

The performance evaluation was conducted using a modified version of the execution-driven simulator SMTsim [17] and the benchmarks of the SPEC CPU2000 suite [14]. The benchmarks were compiled with the Compaq C V5.8-015 compiler running on a Compaq UNIX V4.0 with the optimization option –O3. In order to reduce the simulation time, 300 million of representative instructions of each benchmark were simulated using the input reference set. Representative segment of instructions have been selected by analyzing the distribution of basic blocks as described in [12].

Processor core	**Baseline**	**CPR**	***n*-SP**	**ideal MSP**
Reorder buffer size	128	-	-	-
Instruction queue size	48	128	128	128
Number of checkpoints	-	8 *(out-of-order release)*	-	-
Fetch | Rename | Issue | Retire width	3 | 3 | 5 | 3	3 | 3 | 5 | -	3 | 3 | 5 | -	3 | 3 | 5 | -
Int | Fp register file size	96 | 96	192 | 192	*n* | *n* (each LogReg)	∞ | ∞ (each LogReg)
Ld | L1St | L2St buffer size	48 | 24 | -	48 | 48 | 256	48 | 48 | 256	48 | ∞ | ∞
Confidence branch estimator	-	64 KB | 4 bits	-	-
LCS propagation delay	-	-	1 cycle	0 cycle
Int | Fp | LdSt units	4 | 4 | 2			
Branch predictor	Gshare		Perceptron	
Branch predictor parameters	PHT size: 64k		Global History size: 40 Local History size: 14	Perceptrons: 256 Local History entries: 4k
Memory Subsystem				
I-cache size	64 KB, 4-way, 1 cycle hit			
D-cache size	64 KB, 4-way, 4 cycle hit			
L2-cache size	1 MB, 8-way, 16 cycle hit			
Caches line size	64 bytes			
Main memory latency	380 cycles			

Table 1. Processor configuration

6.1. SPECInt Results

Figure 5 shows the IPC achieved by each of the four architectures described above for SPECInt suite. A 64K-entry gshare branch predictor is used by all four architectures. The *n*-SP processor is evaluated with *n* between 8 and 128 registers in order to understand the impact of *n* on performance.

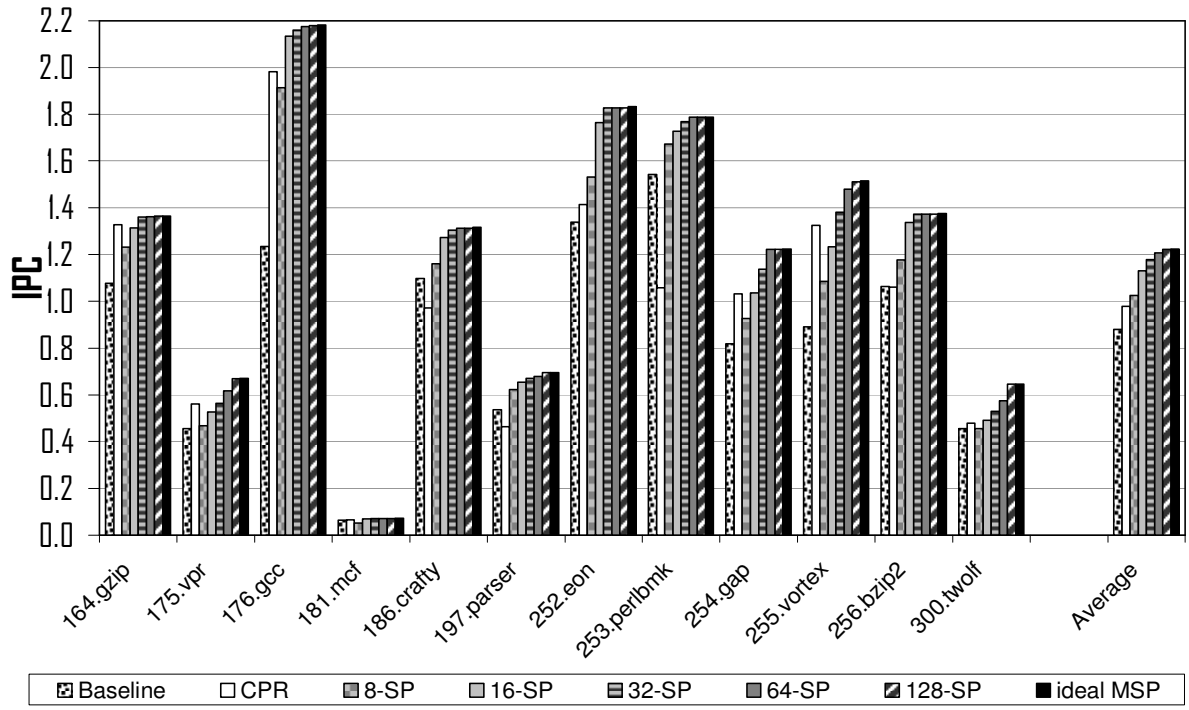


Figure 5. SPECInt IPC for the four architectures (gshare)

The performance of MSP exceeds that of CPR, on average, in all cases. The CPR front end stalls due to an insufficient number of physical registers for renaming. The 8-SP architecture achieves a 4.8% average performance improvement (recall that CPR has an advantage over MSP in not using the arbitration stage or banking in the register file). The 16-SP achieves a 15.5% performance improvement. Beyond that the improvement is relatively small. The performance of the 128-SP is basically identical to the ideal MSP.

One can argue that 16-SP uses more than twice the number of registers used in the CPR processor and this is the reason for performance improvement. However, we argue that MSP can use a larger register file due to its register management architecture. MSP also has a faster and more precise mis-prediction/exception recovery, which significantly improves its performance compared to CPR.

Performance of individual benchmarks for 8-SP varies with respect to CPR, it is only the 32-SP architecture that has better performance than CPR for all benchmarks. The *perlbmk* benchmark shows a high degradation on the CPR due to its branch prediction problems and a large number of re-executed correct-path instructions (see Figure 8 below).

To better understand the impact of branch prediction, the comparison of the four architectures was repeated using the best-performing branch predictor – a large perceptron predictor. The results in Figure 6 show that a branch predictor has a much bigger impact on the CPR processor than on the MSP. The 8-SP IPC average is now 2.6% lower than CPR and the 16-SP is only 7.8% better than CPR. However, overall the IPC trend is the same as with the gshare predictor.

Also shown in the figure are the 16-SP processor stall cycles from just three of the registers that contribute the most to performance loss. Even with 512 registers, the amount of stalls can be very high as MSP exhausts physical registers in a bank. MSP performance can be further improved with compiler assistance if the compiler takes into account renaming restrictions of the MSP. More on this is in the next subsection.

Finally, one can argue that the MSP performance advantage over CPR is due to its much larger register file – 16-SP has 512 physical registers while CPR has only 192. To answer this concern, CPR with 256 and 512 registers was evaluated for SPECInt benchmarks. The CPR with the perceptron

predictor has a 0.6% IPC improvement with 256 registers and a 1.0% improvement with 512 registers (compared to 192 registers). Using 128 registers results in a 3% IPC loss.

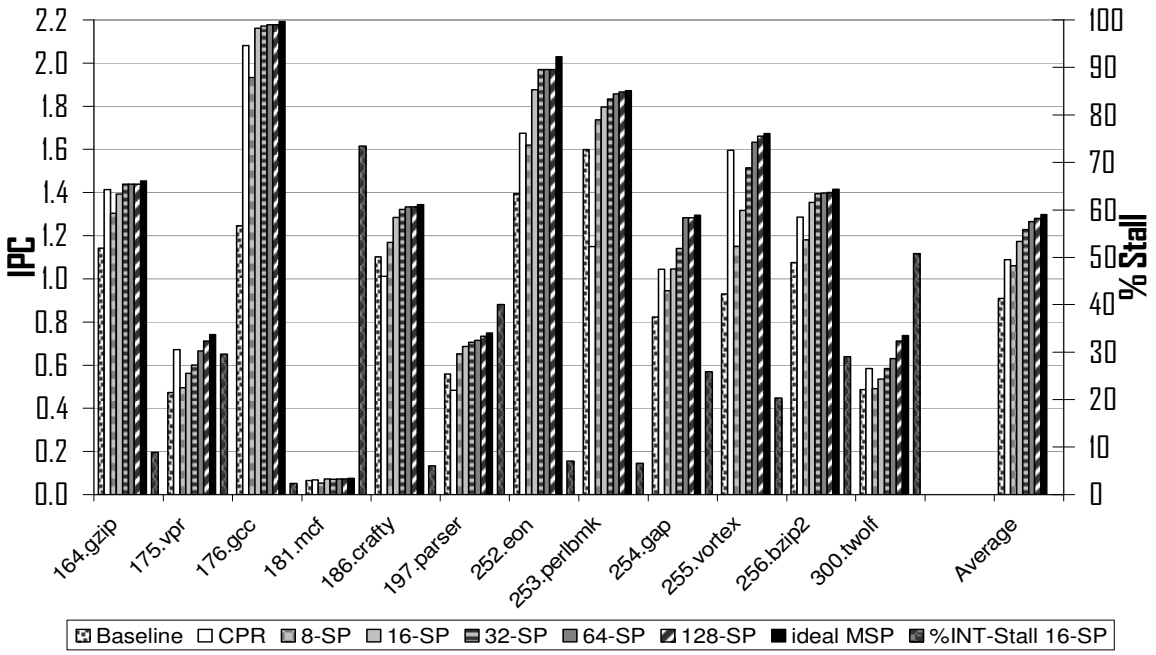


Figure 6. **SPECInt IPC for the four architectures (perceptron)**

6.2. SPECfp Results

The IPC results for the floating point benchmarks are shown in Figure 7. The MSP performance is now better than that of CPR only with 64 physical registers per bank. This is again due to the fraction of execution time when MSP is stalled due to lack of registers (two right-most bars in Figure 7) on most-frequently used registers. In programs with very low stall cycles, such as *lucas,* the 8-SP performance is better than that of CPR. In other cases CPR does better.

The stall cycles shown are the sum of stalls only due to the three integer or f.p. registers producing most stalls. Our analysis of register use showed a very non-uniform register usage in f.p. programs. This is where compiler optimizations would be most beneficial. In particular, loop unrolling of very short loops will produce a significant reduction is such stalls and lead to 8- or 16-SP performance becoming competitive to that of CPR[‡].

[‡] The instruction traces used in this study were produced by another group, thus we were unable to experiment with loop unrolling so far.

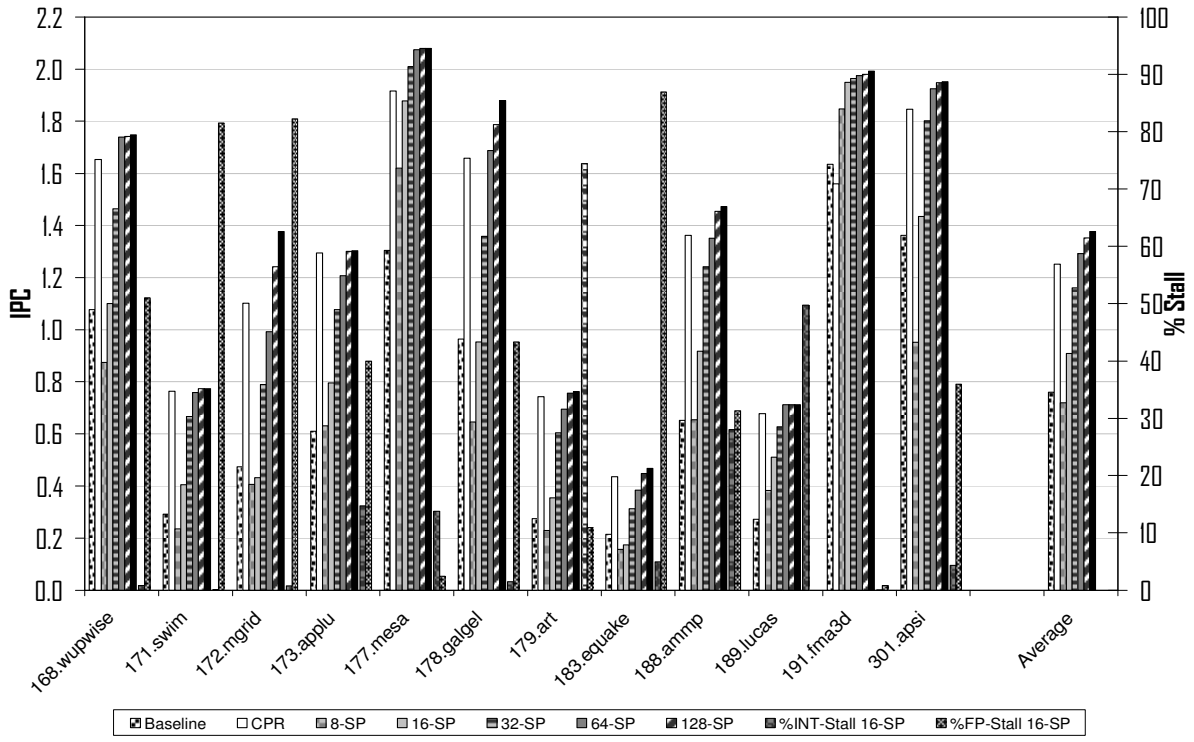


Figure 7. SPECfp IPC and stalls due to lack of registers

6.3. Impact on Power Consumption

One of the advantages of the MSP architecture is the precise state recovery, which avoids re-execution of any correct-path instructions. Figure 8 shows the total number of executed instructions and the number of correct-path instructions executed by the CPR and the 16-SP architectures for integer benchmarks. 16-SP executes, on average, 16.5% fewer instructions than CPR. In some benchmarks, the difference is more than 25%.

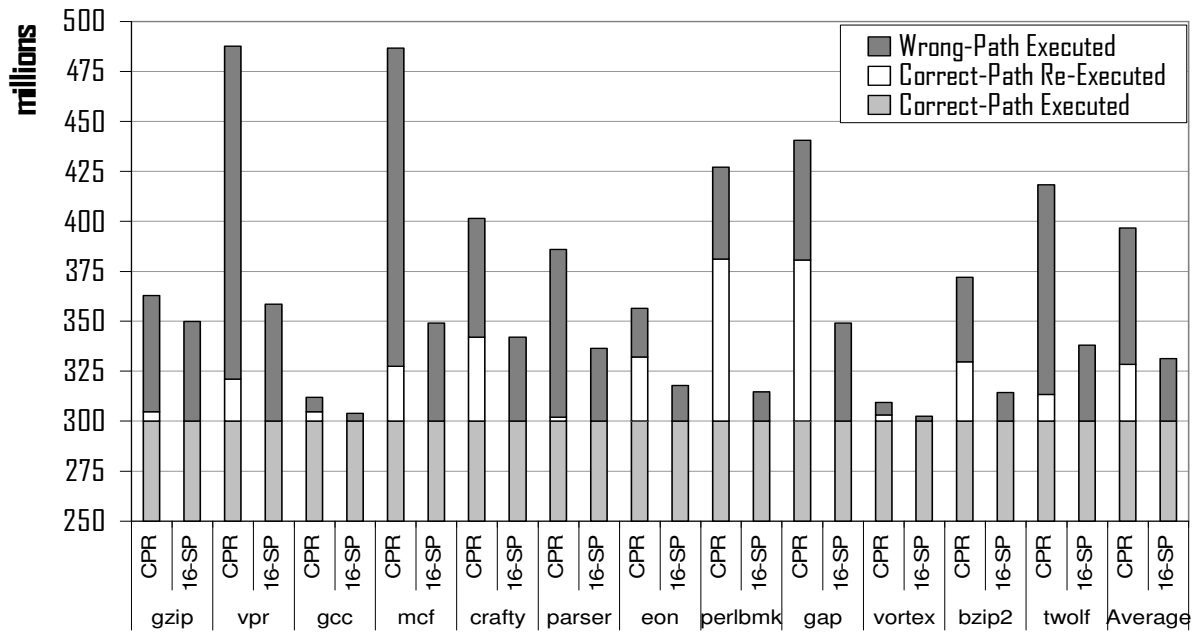


Figure 8. Total number of executed instructions running the specINT

The 16.5% reduction in executed instructions and the reduced execution time lead to significant power savings, especially given power management features of modern processors, which disable unused units.

MSP requires a large physical register file(s) which can affect the power consumption. However, recent studies of multi-banked register files [11, 7] show that, for a large number of banks (such as 32 banks) the number of ports per bank can be reduced with a very minor increase in access conflicts. Such a reduction in the number of ports per bank results in reduced power consumption, as well s a reduction in area and access time of the register file.

The use of a register file with only one read port and one write port per register file bank in the 16-SP architecture results in a degradation of only 0.36% compared with a 8 Read/4 Write port (see [24] for details). The 1R/1W register file uses two mechanisms to reduce the number of conflicts per bank: *conservative bypass-skip* and *read sharing* [16]. Port conflict detection control in the wakeup-select logic can increase the cycle time and hardware complexity. MSP uses to a new pipeline stage, *arbitrate,* to detect conflicts*, which results in* only a 1.66% reduction in IPC. Thus, significant savings in power, area and access time are achieved by the banked register file with minimal loss of performance (see [16] for power details of reduction from banking). MSP can also power down physical registers and SCT entries using the SCT Valid bit, per [4].

To evaluate power consumption, a physical register file was designed and laid out. For 16-SP it consists of 32 banks with 16 64b entries [25]. Each bank has 1 Rd and 1Wr port. The power and access time of one bank were evaluated using SPICE and predictive technology models for 65nm and 45nm process. Total access power was computed using the following equation, which included leakage power of idle banks:

$$TAcc\_power = (Acc\_power) + (N-1) \times Idle\_power \text{ , where:}$$

$TAcc\_power$ is total average power, Acc_power is bank access power, $Idle\_power$ is bank idle state power, N is the number of banks.

Tech	CPR64X192/4B		CPR64X192/8B		MSP-RF512X16/32B	
	Write	**Read**	**Write**	**Read**	**Write**	**Read**
65nm	4.75	4.5	2.75	2.65	2.05	2.1
45nm	3.3	2.6	2.1	2.1	2.0	1.65

Table 2. Register File access power (mW)

Results in the table do not include the power consumption of the decoder, assuming that all designs used a similar address decoder. For comparison, the CPR register file was also banked. The results clearly show that the power consumption of the much larger MSP register file is lower than that of a banked CPR register file.

7. Related Work

Smith and Pleszkun [13] studied support for precise interrupts, such as the *history buffers*, organized similarly to an ROB, and the *future file* that works together with a ROB to improve scalability. However, none of these approaches can support a large amount of instructions in flight.

Hwu and Patt [5] proposed the use of checkpoints to implement precise interrupts but discarding useful work on recovery, i.e. without precise recovery. *Cherry* [8] allows more instructions in flight but still using a ROB in combination with one checkpoint to release resources earlier when it can be guaranteed that all branches have been completed and all memory instructions have been issued.

The *Kilo-instruction Processor* [2] is a multiple check-point based architecture, allowing even more instructions in flight. It uses a pseudo-ROB for younger instructions to minimize the amount of correct-path instructions re-executed. Another similar proposal is the *CPR* [2], which uses check-

pointing without a ROB and thus also has to re-execute useful instructions. CPR proposed other mechanisms like the hierarchical store queue and an aggressive release of physical registers based on reference counters. The *Continual Flow Pipeline* architecture (*CFP)* [15] improves on CPR by incorporating a two-level instruction queue, adding the Slice Data Buffer where the instructions depending on a L2 cache miss are stored. CFP shows some performance improvement over CPR.

Other related work includes: [18] proposes to stall decode while there are many outstanding and likely to be miss-predicted branches. [19] uses a simple confidence estimator to allocate checkpoints selectively to reduce power and maintain performance (it precedes CPR) [20] proposes to overlap recovery with renaming down the correct-path. [22] proposes a virtual context architecture (VCA) to support both multithreading and register windows, providing higher performance with significantly fewer registers than a conventional machine.

[21] Independently proposed a register reference counting scheme based on binary counters represented as matrices (the same idea was part of our Technical Report [24]).

8. Conclusions

The multi-state processor architecture proposed in this paper enables implementation of large-window processors with a large physical register file and precise recovery of execution state on mis-predicted branches and exceptions. It does not use a traditional ROB or check-pointing to achieve this. It also proposed a novel register management architecture, integrated with commit and register release. Using a 256-entry banked register file with only 1Rd and 1 Wr port per bank, it achieved an average IPC increase of 4.8% compared to the CPR architecture with a 192-entry fully-ported register file, even though MSP used an extra pipeline stage. The main sources of performance loss are integer and f.p. register file stalls. The IPC improvement is 16.5% with a 512-entry register file.

MSP also executes 16.5% fewer instructions, in large part due to precise state recovery. This and the use of a banked register file with a minimal number of ports, reduces the power consumption.

References

[1] Cristal, A., Valero, M., Gonzalez, A. and Llosa, J. Large Virtual ROBs by Processor Checkpointing. UPC-DAC-2002-43 Technical Report, Sept. 2002.

[2] Akkary, H., Rajwar, H.R. and Srinivasan, S.T. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. MICRO-36, December 2003.

[3] Goshima, M. et al. A high-speed dynamic instruction scheduling scheme for superscalar processors. MICRO-34, 2001.

[4] Heo, S. et al. Dynamic Fine-Grain Leakage Reduction Using Leakage-Biased Bitlines. ISCA-29, 2002.

[5] Hwu, W. and Patt, Y. Checkpoint repair for out-of-order execution machines. ISCA-14, 1987.

[6] Jacobsen, E., Rotenberg, E. and Smith, J.E. Assigning confidence to conditional branch predictions. MICRO-29, 1996.

[7] Kim, N.S. and Mudge, T. Reducing Register File Ports Using Delayed Write-Back Queues and Operand Pre-Fetch. Proc. ICS-17, 2003.

[8] Martínez, J.F. et al. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. MICRO-35, 2002.

[9] Moudgill, M., Pingali, K. and Vassiliadis, S. Register renaming and dynamic speculation: an alternative approach. MICRO-26, 1993.

[10] Palacharla, S. Complexity-Effective Superscalar Processors. Ph.D Thesis, 1998

[11] Park, I., Powell, M.D. and Vijaykumar, T.N. Reducing Register Ports for Higher Speed and Lower Energy. MICRO-35, 2002.

[12] Sherwood, T., Perelman, E. and Calder, B. Basic block distribution analysis to find periodic behaviour and simulation points in applications. PACT-10, 2001.

[13] Smith, J.E. and Pleszkun, A.R. Implementation of precise interrupts in pipelined processors. ISCA-12, 1985.

[14] SPEC. Standard performance evaluation corporation (spec) 2000 benchmark suite.

[15] Srinivasan, T. et al. Continual Flow Pipelines. ASPLOS-XI, 2004.

[16] Tseng, J.H. and Asanovic, K. A Speculative Control Scheme for an Energy-Efficient Banked Register File. IEEE Transactions on Computers, 2005.

[17] Tullsen, D.M. Simulation and modelling of a simultaneous multithreading processor. Int'l Ann. Computer Measurement Group Conference, 1996.

[18] Akl, P. and Moshovos, A. BranchTap: Improving Performance with Very Few Checkpoints Through Adptive Speculation Control. Int'l Conference on Supercomputing, 2006.

[19] Moshovos, A. Checkpointing Alternatives for High Performance, Power-Aware Processors. Int'l Symposium on Low Power Electronics and Design, 2003.

[20] Zhou, P., Onder, S. and Carr, S. Fast Branch Misprediction Recovery in Out-of-Order Superscalar Processors. Int'l Conference on Supercomputing, 2005.

[21] Roth, A. Physical Register Reference Counting. IEEE Computer Architecture Letters, 2007.

[22] Oehmke, D.W., Binkert, N.L., Mudge, T. and Reinhardt, S.K. How to fake 1000 registers. MICRO-38, 2005.

[23] Buti, T.N., McDonald, R.G., Khwaja, Z., Ambekar, A., Le, H.Q., Burky, W.E. and Willimans, B. Organization and Implementation of the Register-Renaming Mapper for Out-of-Order IBM POWER4 processors. Journal of Research and Development, 2005 Vol. 49, Number 1, p.167.

[24] Technical Report. Authors and institution omitted for blind review. Sept. 2007

[25] Technical Report. The MS-Processor's Register File Evaluation. Authors and institution omitted for blind review. Nov. 2007