A Distributed Processor State Management Architecture for Large-Window Processors

Isidro González[1] Marco Galluzzi[1] Alex Veidenbaum[2] Marco A. Ramírez[3] Adrián Cristal[4] Mateo Valero[1,4]

[1]Department of Computer Architecture, UPC, Barcelona, Spain
[2]Department of Computer Science, UC Irvine, Irvine, USA
[3]Center for Computing Research, IPN, México City, México
[4]Department of Computer Architecture, BSC – CNS, Barcelona, Spain

{iglez, galluzzi, mateo}@ac.upc.edu Alex.Veidenbaum@uci.edu mars@cic.ipn.mx adrian.cristal@bsc.es

Abstract **Processor architectures with large instruction windows have been proposed to expose more instruction-level parallelism (ILP) and increase performance. Some of the proposed architectures replace a re-order buffer (ROB) with a check-pointing mechanism and an out-of-order release of processor resources. Check-pointing, however, leads to an imprecise processor state recovery on mis-predicted branches and exceptions and re-execution of correct-path instructions after 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). The MSP does not use check-pointing, avoids the above-mentioned problems, and has a fast, distributed state recovery mechanism. The MSP uses a novel register management architecture allowing implementation of large register files with simpler and more scalable register allocation, renaming, and release. It is also key to precise processor state recovery mechanism. The MSP is shown to improve IPC by 14%, on average, for integer SPEC CPU2000 benchmarks compared to a check-pointing based mechanism ([2]) when a fast and simple branch predictor is used. With a very aggressive branch predictor the IPC improvement is 1%, on average, and 3% if some of the programs are optimized for the MSP. The MSP also reduces the average number of executed instructions by 16.5% (12% for the aggressive branch predictor), mostly due to precise state recovery. This improves the MSP processor energy efficiency even though it uses a larger register file.

Keywords-component; Check-pointing; register renaming; register file; misprediction recovery; large-window

1 INTRODUCTION

Recently proposed large instruction window processors, such as *Kilo-instruction Processors* [1] and *Checkpoint 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 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 misprediction leads to restoration of the processor state to a previous checkpoint and re-execution from the checkpoint. 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. Pending stores in a store queue are handled separately and require a more complex and time consuming mechanism. For instance, the time delay of scanning a large, 2^{nd}-level store queue [2] for load-forwarding 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 misprediction 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 (including indirect branches). A new check-point is created if the estimator gives low confidence for the current prediction [6].

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.

When a branch misprediction occurs, the processor rolls back the state to the youngest checkpoint preceding the branch. On an exception, execution cannot resume at the excepting instruction and also has to resume at the preceding check-point. 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 branch

978-1-4244-2837-3/08/$25.00 ©2008 IEEE

prediction accuracy, the number of check-points and the check-point management mechanism [18, 19]. This lack of precision in branch misprediction or exception recovery overhead degrades processor performance and increases power consumption. Increasing the number of check-points does not guarantee an improvement in performance and 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 is 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 mispredictions 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 a physical register immediately after the last use of the register. Counters are easy to update but introduce additional complexity in state 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, release of registers, and branch misprediction 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.

For physical register management the MSP proposes a new, scalable register allocation, renaming, and release mechanism. It does not use either of the traditional approaches: a physical register free list and Register Alias Table (RAT) or CAM-based designs. Instead, the MSP uses a mechanism for releasing a physical register based on its use by dependent instructions, which is integrated with state management and commit. Allocation and renaming are distributed and done separately for each logical register. The proposed mechanism can reduce the high power density and overall power dissipation of a renaming unit and the register file. Furthermore, it allows the register file itself to be banked in a novel way, reducing bank port requirements to 1 read and 1 write ports.

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 proposed in the MSP architecture. Consider a dynamic instruction sequence shown in Fig. 1. Let us use it to illustrate the problem of state recovery in a check-pointing processor to motivate the state management of the MSP.

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 misprediction occurs at instruction 7, where a check-point has not been set. As part of branch misprediction recovery, the processor's state is restored to the state stored in the previous checkpoint – the one set at instruction 3. Execution resumes from instruction 3 and the processor re-executes instructions from 3 to 6 (the dashed box in Fig. 1), even though they were already correctly executed prior to branch misprediction.

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.

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* as the state of its registers. It assigns a new state to each instruction that writes a destination register. Thus the state difference between two adjacent instructions is at most the state of one register. The allocation, recovery and release or commit 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 but in order determined by processor state.

No	PC	Instruction	StateId
1	@+00	store r2, @data	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

A processor state is assigned a value called a StateId maintained by a binary counter. StateId is incremented when an instruction is added to the instruction window and the instruction assigns a logical register (allocates a new physical register). Instructions not assigning a register, such as branches or stores, do not change the state. *The current StateId is associated with the instruction entering the window*.

For each physical register a range of consecutive states in which the register is valid, called its StateId Range, is maintained. The Lower StateId of the range is the StateId of the instruction assigning the physical register. The Upper StateId of the range is the StateId of the instruction preceding the instruction that next renames the corresponding logical register. The StateId range allows identification of all instructions (states) using a given physical register.

State recovery becomes a simple process: instructions with a StateId greater than the StateId of the instruction causing the recovery are discarded. Stores in the store queue are also released using the same condition.

For instructions in Fig. 1, the assigned StateIds are in the column "StateId" of the figure. The StateId range associated with each physical register is shown in Fig. 2. The notation R*x.y* describes *(renaming)* 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.

The R*x.y* notation is key to the MSP register management mechanism (described in Sec. 3).

The MSP branch misprediction recovery proceeds as follows (for instruction 7 in Fig. 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. Physical registers whose Lower StateId > 4 can be released – only register R1.2 in the example.

3 MICRO-ARCHITECTURE OF THE MSP PROCESSOR

The micro-architecture of the MSP processor and its pipeline are shown in Fig. 3. The micro-architecture uses a banked physical register file with 1 read and 1 write ports per bank. This requires arbitration to detect register port access conflicts and the MSP adds an arbitration stage to the pipeline. The register management is distributed among banks as described below.

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 Fig. 1

This micro-architecture does not use a standard reorder buffer, renamer, etc. Instead, the State Control Tables (SCT) and associated logic manage these functions in a distributed fashion. Some of the functions require interaction between all SCT, such as commit. This involves the *Last Committed StateId* (LCS) logic and State Counter shown in Fig. 3. In the MSP architecture described in this paper register release and commit happen at the same time. A new LCS value is computed every clock cycle.

Every instruction in the instruction queue sets *Use* bits in the dependency tracking logic for each physical register it uses. A Use bit is reset once such an instruction consumed the register value. An OR of all use bits and of the Ready bit for a given physical register generates the RelIQ signal used in register management.

3.1 Register and State Id management

MSP integrates state and register management in a single, scalable mechanism. To make register management a distributed mechanism the MSP imposes the following two constraints:

a) Each logical register is renamed to *a fixed subset of physical registers* (a *bank* of physical registers)

b) Physical registers are allocated and released in order within such a bank for a logical register

These constraints allow register allocation, renaming, and source register lookup to be performed independently for each logical register and makes them independent of the physical register file size. It allows the physical register to be identified as R.x, where R is the logical register number. A global free list of physical registers and a global Register Alias Table [26] or a CAM-based allocator/renamer [23] are no longer required.

3.2 MSP operation

The management of MSP registers and state is divided into *local management* for each logical register (bank), and *global management* interacting with the rest of the processor and using information from all banks.

3.2.1 Local Management

The local management logic is shown in Figs. 4 and 5. It consists of a *State Control Table (SCT)* plus rename and release pointers with their associated logic. An SCT entry is a descriptor for one physical register and records the processor state assigned to the instruction that assigns the register.

The management performs (locally) allocation of a new physical register, renaming, tracking the use of each physical register in the bank by dependent instructions in the IQ, and release of physical registers on commit or state recovery.

An SCT entry for a physical register in a bank contains:

- *StateId:* the value of the *Lower StateId*, the StateId of the instruction assigning the physical register. The *Upper StateId* is implicit – it is the value of the next SCT entry minus one (recall Fig. 2). For the most recent entry (last renaming) the *Upper StateId* is Null.

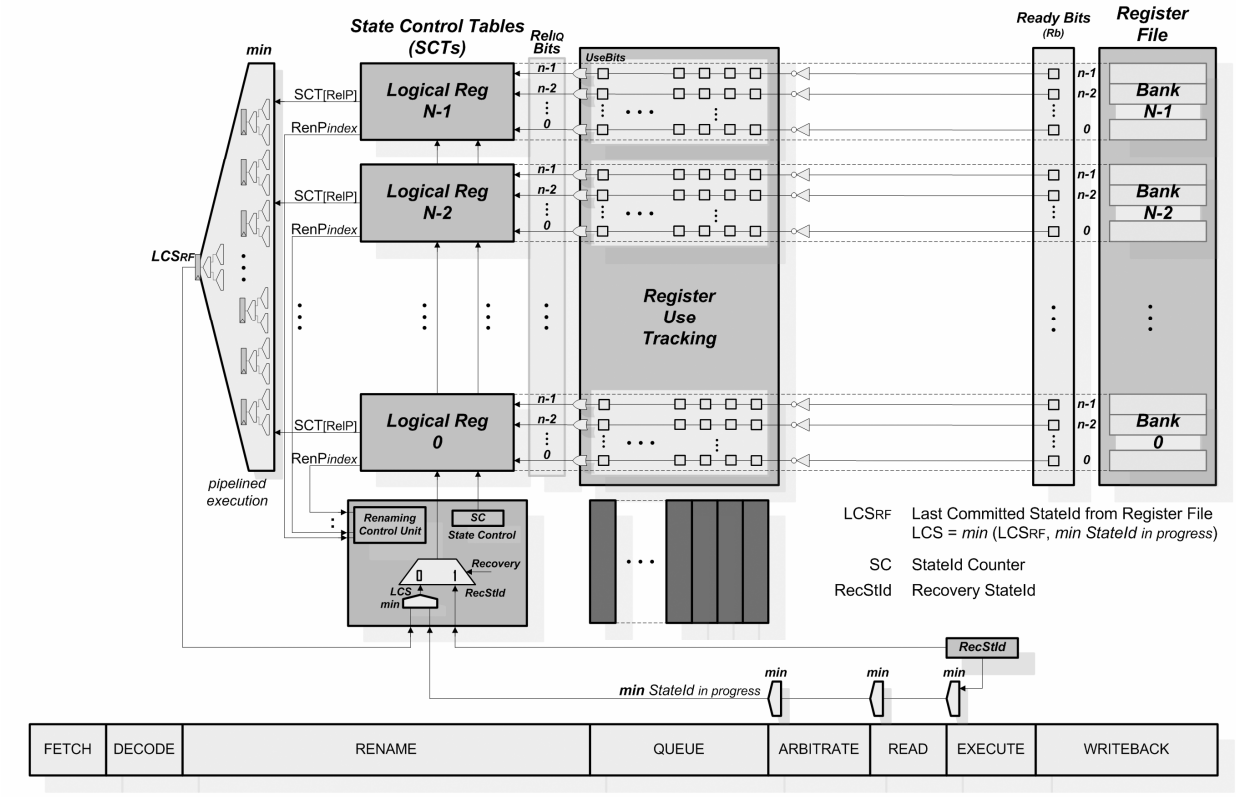

Figure 3. The MSP micro-architecture

- *Valid Bit (Vb):* specifies whether the entry is in use.

The control logic associated with each entry consists of:

- *StateId Range Comparator:* It compares the StateId of the entry and the next one with a StateId broadcast by the Global Control. A physical register is released and instruction committed if its StateId < LCS (unless it is the last such register in a bank). StateId range comparison is used to update the rename pointer.
- Logic to track 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 registers for committed instructions is a continuous process and instructions are committed in the StateId order.
- Recovery logic to receive a globally broadcast *Recovery StateId* and detect if a physical register needs to be released (see Sec. 5 for details). A register is released if its *StateId > Recovery StateId.*

Two 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 allocated physical register (SCT entry), which is 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 and the RenP index pair.
- *Release Pointer (RelP):* points at the first physical register in the bank that can not be released. The value in this register either has not been produced (Ready bit Rb=0) or has not been consumed by all its dependent instructions or some of the instructions associated with this StateId have *not* yet executed. RelP takes part in the global computation of the *Last Committed State* or *LCS = Min(RelPi)*, where $i \in [0, \text{NumLogReg-1}]$, as described in more detail below.

Any StateId in the bank/SCT such that StateId < RelP can *potentially* be committed and the corresponding registers released. The value in such a register has been produced (Ready bit Rb=1) and consumed by all its dependent instructions. All the instructions associated with this StateId have executed (without exceptions). *"can potentially be committed"* was used above instead of "can be committed" because StateId's are committed iff StateId < LCS.

Min(StateId[RelPi]) = LCS is the oldest entry in the processor that cannot yet be committed. It is computed every clock by the global control logic. Any StateId < LCS in any bank can be committed and its register released (unless it is the last renaming). Thus multiple StateIds across multiple banks

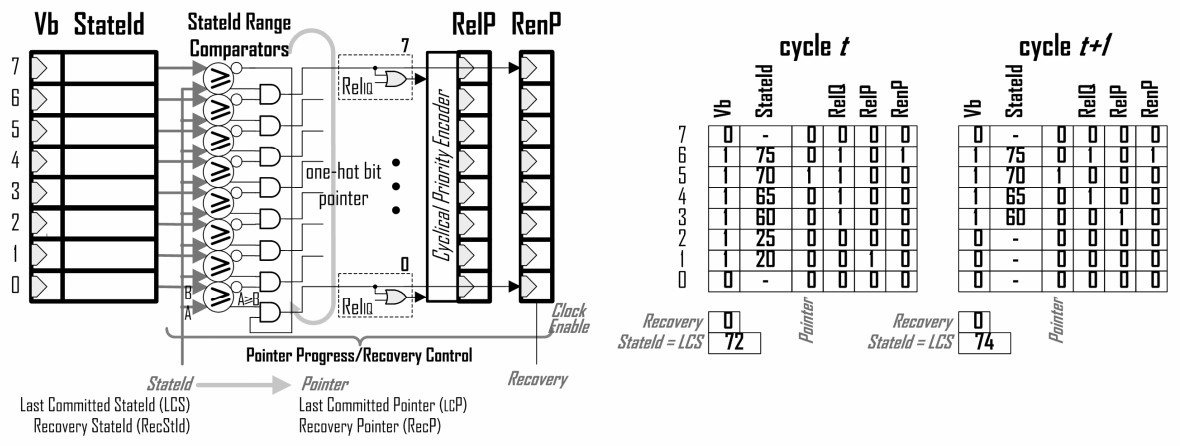

Figure 4. Local Management (release and recovery)

can be committed in the same cycle. Example in Fig. 4 shows pointer change in a cycle after LCS becomes 4.

In summary, local register management functions performed are:

a) Allocating a new physical register and renaming the corresponding logical register, accomplished by incrementing (shifting) the rename pointer RenP,

b) Source register renaming by encoding RenP and concatenating it with the logical register number (bank number),

c) Releasing a physical register, accomplished by setting the Valid bit to 0 for any physical register i such that SCT.StateId(i) < LCS,

d) Tracking uses of each register in a bank by dependent instructions

3.2.2 Global Management

The global management maintains the current state of the processor and determines the oldest non-committable StateId, LCS. This is accomplished by the following two functional blocks:

- *The StateId Counter (SC),* defines the current processor StateId. It is incremented for each decoded instruction that assigns a logical register.
- *The Last Committed StateId (LCS) unit,* computes the minimum (oldest) StateId of all SCT[RelP] entries every cycle. The LCS is the oldest state in the MSP that can not yet be committed. Thus any StateId's older than LCS *can be committed*. This may commit *multiple* older states.

Recall that several additional instructions may belong to one state and the state can only be committed when all of them have executed. Such instructions do not assign registers and thus cannot be tracked through the SCTs or through their register dependencies once they leave the instruction queue. This is why the state of instructions issued but still in the pipeline is tracked in Fig. 3.

A mechanism used to release processor resources of committed instructions/states, registers and Store Queue entries, is based on LCS. The logic in a local scope control of each logical register releases physical registers of entries with StateId < LCS. 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 smaller 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 pipelined to produce a new minimum. Latency of LCS computation is not a critical timing issue, even a 4-cycle LCS computation degrades performance by less than 1% compared to a 1-cycle computation. As for power, static 9bit comparators with low power consumption can be used.

A special condition occurs if (RenP=RelP and RelIQ[RenP]=0), i.e. all physical registers have been renamed, produced and consumed by all consumer instructions. The StateId of the SCT entry pointed by both RelP and RenP is not used in the computation of the LCS.

3.3 Renaming of Multiple Instructions per cycle

So far we have described several register management functions: allocation, renaming, release, and misprediction /exception recovery. Note that shadowing (or checkpointing) of the register map is not needed in the MSP, its function is performed by state recovery. The source operand lookup is performed by reading the RenP in a given SCT.

In the simplest case described above, the renaming process advances the rename pointer, RenP, by one in the required bank. However, the renaming process is complicated by the fact that multiple instructions may assign the same destination logical register in one clock 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 5% reduction in IPC. Therefore, the renaming logic described in this section allows up to four destination registers to be renamed per cycle, two of which can be the same logical register.

Fig. 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 a register 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 next 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. The current RenP bit vector value of each SCT, *RenPindex*, is sent to *Renaming Control Unit*, and used as base pointers to the corresponding source operands of instructions in the renaming cycle. The RAW dependences are resolved by increasing these pointers by the number of previous instructions which write in the same logical register (this control is very similar to traditional RAW dependency control). A port decoder identifies 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.

Lack of physical registers for renaming in a bank may cause a stall of all stages prior to Rename. The stall is detected by an SCT and 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. Note that this lack of space in a bank can be detected very early in the renaming cycle.

3.4 Tracking Register Use

To detect when the last use of a register occurs and also when all instructions associated with a register state have completed execution could be done with reference counters [9]. MSP proposes a different solution.

A bit vector *RelIQ* of the size equal to instruction queue size, is used to track dependents of a register and of all instructions belonging to the same state. 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 *vector* for each source register, *UseBit*. Note that this can be done within each SCT in

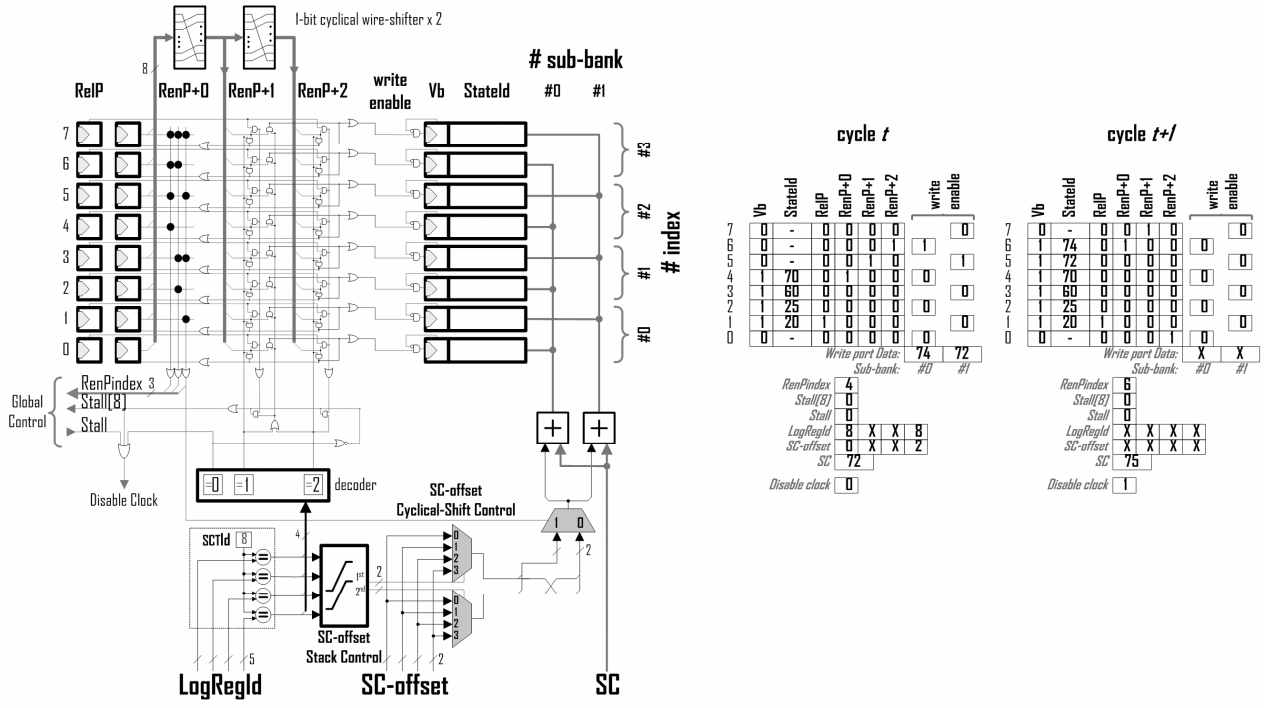

Figure 5. Local Management (renaming logic for one logical register)

TABLE I. PROCESSOR CONFIGURATION

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 | -
nt | 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	-	64k entries | 4 bits	-	-
LCS propagation delay	-	-	1 cycle	0 cycle
Int | Fp | LdSt units	4 | 4 | 2			
Branch predictor	gshare		TAGE	
Branch predictor parameters	PHT size: 64k		8 components	
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			

a distributed fashion.

The RelIQ bit vector is also 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 consumed their operands and complete execution without exceptions.

Finally, on branch misprediction or exception recovery all bits in a column of *RelIQ* vectors corresponding to the position of the cancelled instructions are reset.

3.5 The State Recovery Mechanism

The state recovery mechanism on a branch misprediction proceeds as following. The processor state is reset to the StateId of the branch. All instructions in the IQ following the Processor branch are squashed and their *Use* 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.

An exception is actually taken only when the instruction causing the exception is committable and all prior instruction have executed. Any younger instructions are cancelled. The Recovery StateId is set to the StateId of the instruction causing the exception or the StateId of the previous one if this instruction produced a new state. Similar to branch misprediction recovery, multiple instructions associated with a single state have to be dealt with correctly. A 5-bit id is assigned to instrunctions in the same state, in order, to achieve this. After the recovery is complete, the SC is set to the Recovery StateId and the *Recovery StateId* is disabled.

3.6 StateId Overflow

The StateId size is *log2(M) = m* bits, where M is the register file size. Thus the State Counter SC will eventually overflow. The MSP uses a *saturation bit*, *Sb*, added as the most significant bit to the *log2(M) bits of the* StateId to control the overflow. The SC is initialized to zero 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 saturation bit 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.

4 PERFORMANCE EVALUATION

The following architectures are evaluated and compared in this section:

- *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 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 and the same hierarchical store queue as CPR. It uses arbitration.
- *ideal MSP*. MSP with an infinite hierarchical store queue and an infinite, fully-ported register file.

The parameters of the four architectures are shown in Table 1, with many chosen to be identical to the CPR processor[1] in [2]. A notable difference with CPR are branch predictors used in this paper: *gshare* and *partially TAgged GEometric history length (TAGE)* [27] predictor. The former is an example of a simple, fast predictor and the latter of a complex but more accurate one.

The performance evaluation was conducted using a modified version of the execution-driven simulator SMTsim [17] and the SPEC CPU2000 benchmark suite [14]. The benchmarks were compiled with the Compaq C V5.8-015

[1] Even so, our CPR results should not be expected to the same as in [2] because we simulated a different ISA and used a different compiler and simulator.

compiler under Compaq UNIX V4.0 with the optimization option –O3. 300 million representative instructions per benchmark were simulated using input reference sets. These instructions were selected by analyzing the distribution of basic blocks per [12].

4.1 SPECInt Results

Fig. 6 shows the IPC of the four architectures described above using a 64K-entry gshare predictor. The *n*-SP processor is evaluated with *n* physical registers per logical register bank, $8 \le n \le 128$, to understand the impact of n on performance.

On *average*, MSP performance exceeds CPR's in all cases. The 8-SP architecture achieves a 5% average performance improvement. CPR does not use the arbitration stage and yet 16-SP+Arb achieves a 14% performance improvement. Further improvement from increasing n is relatively small. The performance of the 128-SP is basically identical to the ideal MSP. Performance of individual benchmarks for 8-SP varies with respect to CPR, it is only the 32-SP architecture that always has better performance than CPR.

The four architectures were also compared using the most accurate but possibly slower branch predictor – a very aggressive TAGE predictor. The results in Fig. 7 show that a branch predictor has a much bigger impact on CPR performance than on the MSP's. The 8-SP IPC average is now 10% lower than CPR and the 16-SP+Arb is 1% 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+Arb processor stall cycles from just three of the registers that contribute the most to performance loss. Even with 512 registers, the stalls can be very high as MSP exhausts physical registers in a bank.

4.2 SPECfp Results

The IPC results for the floating point benchmarks are shown in Fig. 8. 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 Fig. 8) on most-frequently used registers. In programs with very low stall cycles, such as *fma3d*, the 8-SP performance is better than that of CPR. In other cases CPR does better.

4.3 Reducing Register Stalls

Figs. 7 and 8 show a significant impact on MSP performance due to insufficient registers in a bank and the performance improvement obtained with an increase in the bank size. The main reason is a physical register allocation only within a logical register bank. This may happen in loops that use only a few registers. An 8-SP processor would stall after at most 8 iterations of a small loop. In a flat register file with traditional renaming this does not happen.

One can eliminate or significantly reduce this problem by two simple modifications of a program. One is loop unrolling and the other is a modification to the register allocation in a loop to avoid reusing the same register (even without

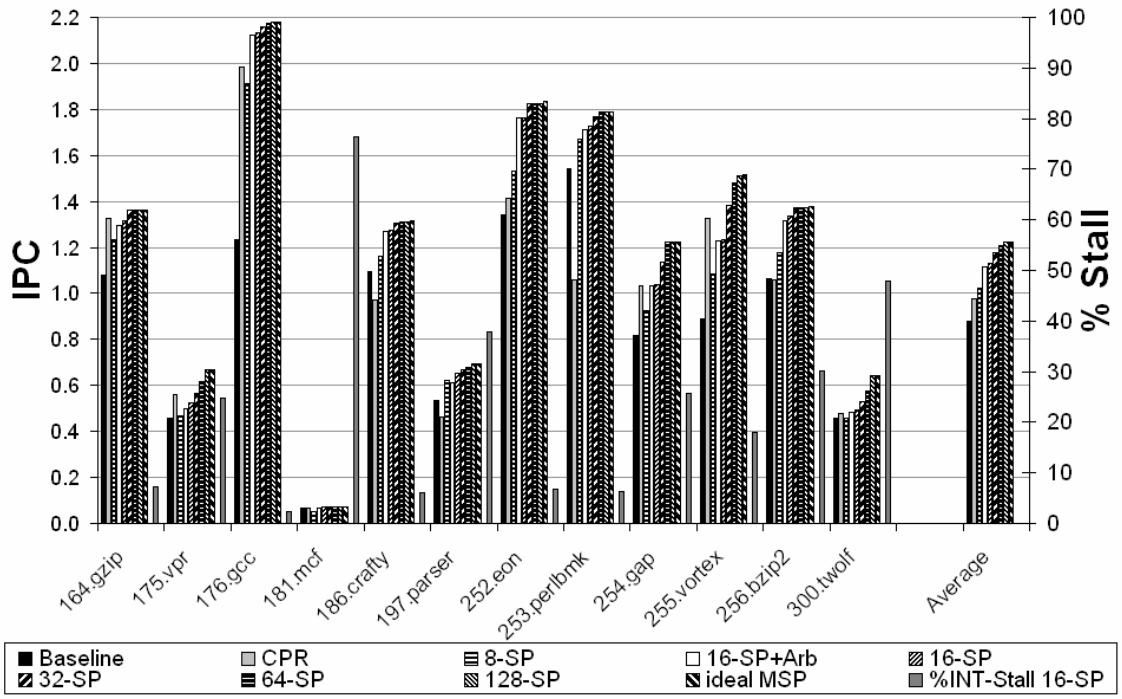

Figure 6. SPECint IPC with gshare and 16-SP stalls due to lack of registers

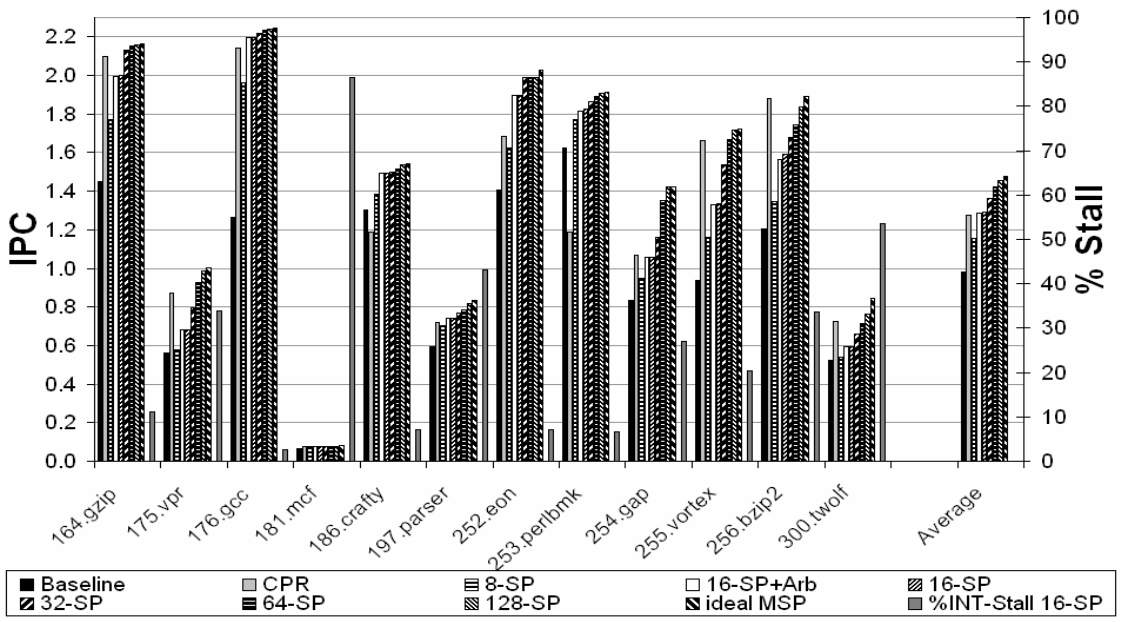

Figure 7. SPECint IPC with the TAGE predictor and 16-SP stalls due to lack of registers

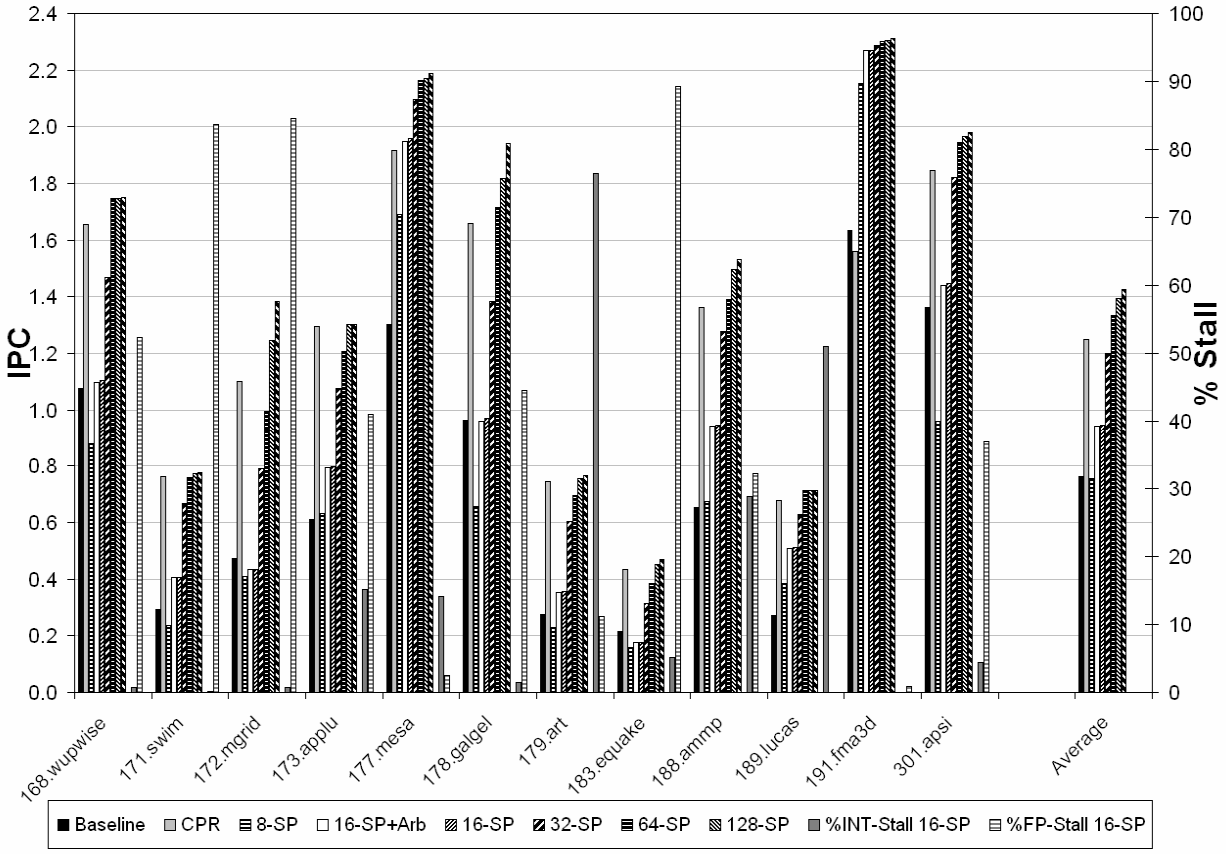

Figure 8. SPECfp IPC with the TAGE predictor and 16-SP stalls due to lack of registers

unrolling). Both of these can be easily implemented in a compiler. A small subset of benchmarks with high stalls was modified by hand changing 1 to 3 important loops per program.

Table 2 shows the performance of modified programs using the TAGE predictor. CPR uses a banked physical register file with 192 registers. After modifying only a few loops in some SPEC-traces, the 16-SP (with arbitration) has a 3% higher IPC, on average, than CPR for the entire SPECInt suite. We believe that when a compiler optimizes all loops in a program causing stalls the MSP performance improvement will be even higher. For SPECfp the performance of 16-SP+Arb is now close to CPR. The 8-SP performance is also much improved.

IPC for CPR with 256 and 512 physical registers (fully ported and without arbitration) and the TAGE branch predictor for SPECint benchmarks was also evaluated. CPR with 256 registers has a 1% IPC improvement and with 512 registers a 1.3% improvement over CPR with 192 registers. Given that 16-SP+Arb (with two modified programs) has a 3% IPC improvement for SPECint over CPR with 192 registers, the reason for MSP performance improvement is *NOT* its larger register file.

TABLE II. IPC FOR MODIFIED BENCKMARKS WITH TAGE BRANCH PREDICTOR

Benchmark function	Loops unrolled	% Execution time	Version	CPR	8-SP+Arb	16-SP+Arb	ideal MSP
256.bzip2 generateMTFValues	1	65	original	2.2	1.8	1.9	2.3
			modified	2.2	2.0	2.1	2.3
300.twolf new_dbox_a	3	19	original	0.6	0.5	0.6	0.8
			modified	0.6	0.6	0.6	0.8
171.swim calc3	0	25	original	0.7	0.3	0.5	0.7
			modified	0.7	0.4	0.7	0.7
172.mgrid resid	0	52	original	1.2	0.5	0.5	1.5
			modified	1.2	1.0	1.2	1.5
183.equake smvp	0	54	original	0.3	0.1	0.2	0.4
			modified	0.3	0.2	0.3	0.4

4.4 Reduction in Instruction Re-execution

Fig. 9 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. The results are presented for two different branch predictors. 16-SP+Arb executes, on average, 16.5% (9.5% due to precise recovery) fewer instructions than CPR using the gshare predictor. The reduction is 12% (7% due to precise recovery) with the TAGE predictor. Further reduction beyond the precise state recovery is a side-effect of the renaming mechanism. The reduction in executed instructions and the reduced execution time result in power savings.

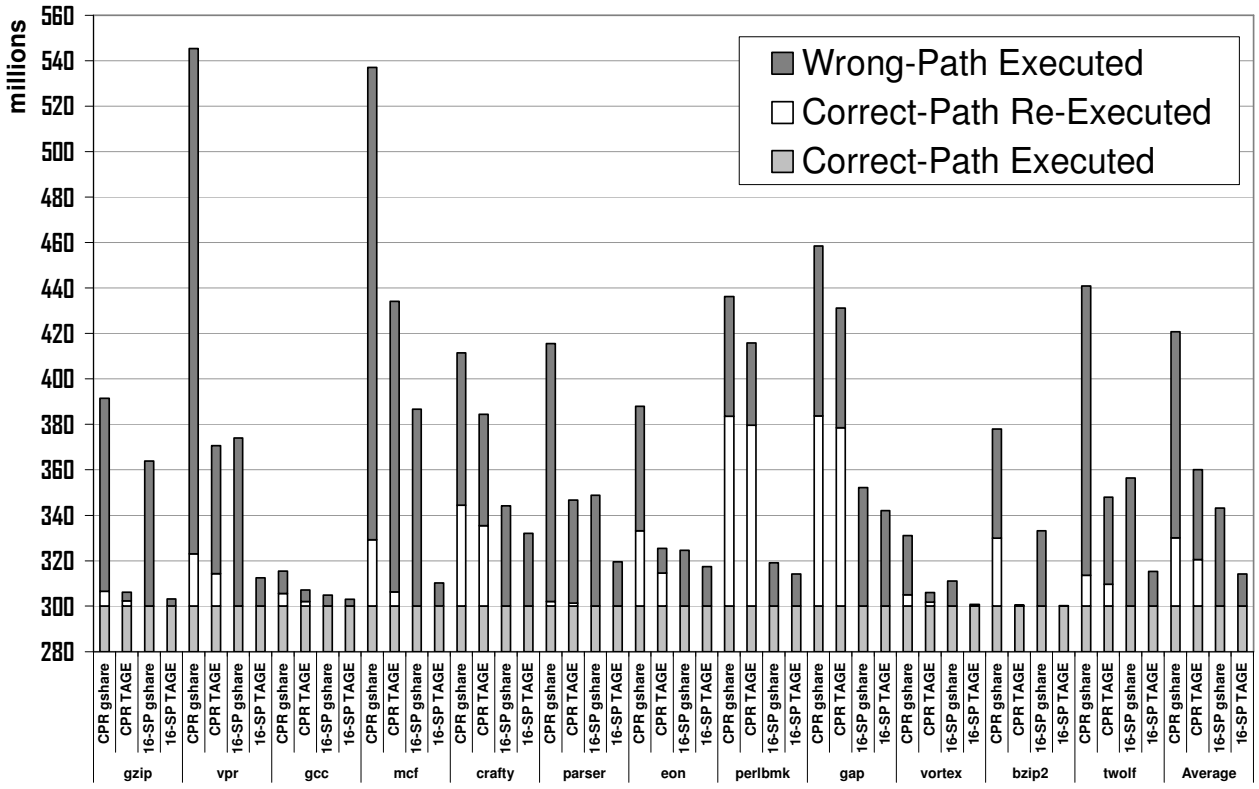

Figure 9. Total number of executed instructions for SPECint benchmarks

TABLE III. REGISTER FILE ACCESS POWER AND ACCESS TIME (MW | FO4)

	CPR 192 / 64bits per entry 4 banks 8Rd/4Wr ports per bank		CPR 192 / 64bits per entry 8 banks 8Rd/4Wr ports per bank		16-SP 512 / 64bits per entry 32 banks 1Rd/1Wr ports per bank	
Technology	***Write***	***Read***	***Write***	***Read***	***Write***	***Read***
65nm	4.75 | 1.06	4.50 | 5.51	2.75 | 1.06	2.65 | 5.51	2.05 | 0.85	2.10 | 4.44
45nm	3.30 | 1.29	2.60 | 6.11	2.10 | 1.29	2.10 | 6.11	2.00 | 1.11	1.65 | 5.92

5 HARDWARE ISSUES

This section discusses hardware complexity, area, power and access time issues arising in the MSP architecture.

5.1 Area Considerations

A banked register file can use 2 Read and 1 Write ports per bank with only a very minor increase in access conflicts [11, 7]. MSP uses 1rd/1wr port per bank because a given instruction needs at most one operand from a given bank – the latest renaming of a logical register. CPR, if banked, needs at least 2 Read /1 Write because a single instruction may need two different source operands from the same bank. The area of such a 512-entry register file is 0.1sq.mm while area of a 256-entry register file in CPR is 0.21sq.mm for 45nm technology (per CACTI 4.2 [29] but without banking).

MSP uses extra area for SCTs but they replace register renamer and some of the ROB and commit logic. Note that an 8 Read/4 Write port standard renamer is quite a large structure.

Finally, MSP adds the register use tracking matrix. It is a structure requiring 1 bit of storage per physical register for each instruction in the instruction queue. The matrix needs 3 Write ports and no read ports or output drivers. Each bit's output is permanently connected to the OR gate generating the RelIQ signal. Write operations and tracking control are completely independent between each sub-matrix (associated to a SCT).

5.2 Power Consumption and Access Time

A physical register file was designed and laid out for the 16-SP architecture. Each bank has 16 64b entries [25] and 1 Rd and 1Wr ports. The power and access time of one bank were evaluated using SPICE based on predictive technology models for 65nm and 45nm process. Similarly, 4- and 8-bank register files with 192 entries but fully ported banks for CPR were also evaluated.

Total access power was computed using the following equation [2], which includes leakage power of idle banks: $TAcc\_power = Acc\_power + (N - 1) \times Idle\_power$, where *TAcc_power* is total average power, *Acc_power* is a bank access power, *Idle_power* is idle bank power, and *N* is the number of banks.

[2] Power consumption of the address decoder is the same in all designs as they use a similar address decoder. It is not included here.

The results clearly show in Table 3 that the power consumption and access time (see footnote) of the larger 16-SP register file are lower than that of a banked CPR register file.

Of course much higher energy savings are achieved by the MSP due to executing fewer instructions.

6 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 number 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 uses 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* [1] is a multiple checkpoint 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 checkpointing 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.

Reference [18] proposed to stall decode while there are many outstanding and likely to be miss-predicted branches. Reference [19] used a simple confidence estimator to allocate checkpoints selectively to reduce power and maintain performance (it precedes CPR) [20] proposed to overlap recovery with renaming down the correct-path. Reference [22] proposed a virtual context architecture (VCA) to support both multithreading and register windows, providing higher performance with significantly fewer registers than a conventional machine and the idea of giving each logical register a FIFO queue of physical registers [28] has been previously used to make register reclamation easy in a very

different context for clustered architectures. Runahead+CPR was compared with CFP in [15] thus indicating Runahead's capabilities. Qualitatively, as branch predictors become better, the advantages of large-window processors (CPR, CFP, Kilo, MSP) over Runahead should increase. Reference [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]).

7 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. MSP uses a novel, scalable register management architecture, integrated with commit and register release. This includes a new approach to register renaming. Its banked register file can use just 1 Rd and 1 Wr port per bank significantly reducing its size and power consumption. MSP with the TAGE branch predictor achieves an average IPC increase of 3% compared to the CPR architecture with a 192-entry fully-ported register file and 14% using the gshare predictor. The performance of the MSP is affected by integer and f.p. register file stalls. We believe that these can be reduced or completely eliminated with compiler optimizations. MSP also executes 16.5% fewer instructions with gshare and 12% with TAGE, mostly due to precise state recovery. This and the lower power consumption in the register file make it more power efficient than CPR.

Acknowledgments

The authors would like to thank Alex Pajuelo and Oliverio J. Santana for their comments and advice in the preliminary stages of this work. This work was supported by the UPC research grant, by the Ministry of Science and Technology of Spain under contracts TIN–2004–07739–C02–01 and TIN–2007–60625, and by the Framework Programme 6 HiPEAC Network of Excellence (IST–004408) and by Framework Programme 7 HiPEAC 2 (IST-217068). Veidenbaum was supported in part by the National Science Foundation award CNS-0220069 for NSF/EU collaboration.

References

[1] A. Cristal, M. Valero, A. Gonzalez, and J. Llosa, "Large virtual ROBs by processor checkpointing," UPC-DAC-2002-43 Technical Report, Sept. 2002.

[2] H. Akkary, H. R. Rajwar, and S.T. Srinivasan, "Checkpoint processing and recovery: towards scalable large instruction window processors," MICRO-36, December 2003.

[3] M. Goshima, K. Nishino, T. Kitamura, Y. Nakashima, S. Tomita, and S. Mori, "A high-speed dynamic instruction scheduling scheme for superscalar processors," MICRO-34, 2001.

[4] S. Heo, K. Barr, M. Hampton, and K. Asanovic, "Dynamic fine-grain leakage reduction using leakage-biased bitlines," isca, p. 0137, 29th Annual International Symposium on Computer Architecture (ISCA'02), 2002.

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

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

[7] N. S. Kim, and T. Mudge, "Reducing register file ports using delayed write-back queues and operand pre-fetch," Proc. ICS-17, 2003.

[8] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas, "Cherry: checkpointed early resource recycling in out-of-order microprocessors," MICRO-35, 2002.

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

[10] S. Palacharla, "Complexity-effective superscalar processors," Ph.D Thesis, 1998.

[11] I. Park, M. D. Powell, and T. N. Vijaykumar, "Reducing register ports for higher speed and lower energy," MICRO-35, 2002.

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

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

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

[15] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. T. Upton, "Continual flow pipelines," ASPLOS-XI, 2004.

[16] J. H. Tseng, and K. Asanovic, "A speculative control scheme for an energy-efficient banked register file," IEEE Transactions on Computers, 2005.

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

[18] P. Akl, and A. Moshovos, "BranchTap: improving performance with very few checkpoints through adptive speculation control," Int'l Conference on Supercomputing, 2006.

[19] A. Moshovos, "Checkpointing alternatives for high performance, power-aware processors," Int'l Symposium on Low Power Electronics and Design, 2003.

[20] P. Zhou, S. Onder, and S. Carr, "Fast branch misprediction recovery in out-of-order superscalar processors," Int'l Conference on Supercomputing, 2005.

[21] A. Roth, "Physical register reference counting," IEEE Computer Architecture Letters, 2007.

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

[23] T. N. Buti, R. G. McDonald, Z. Khwaja, A. Ambekar, H. Q. Le, W. E. Burky, and B. Williams, "Organization and implementation of the register-renaming mapper for out-of-order IBM POWER4 processors," IBM Journal of Research and Development, 2005 Vol. 49, Number 1, p.167.

[24] I. Gonzalez, M. Galluzzi, A. Cristal, A. Pajuelo, O. J. Santana, and M. Valero, "The multi-state processor: ROB-free architecture with precise recovery," UPC-DAC-RR-2007-45 Technical Report, Sept. 2007.

[25] I. Gonzalez, A. Cristal, A. Veindenbaum, M. A. Ramirez, and M. Valero, "The MSP processor's register file timing and power evaluation," UPC-DAC-RR-2008-51 Technical Report, Sept. 2008.

[26] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," IEEE MICRO, April 1996.

[27] A. Seznec, and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," IRISA/INRIA/HiPEAC, 2006.

[28] R. Nair, and M. E. Hopkinks, "Exploiting instruction level parallelism in processors by caching scheduled groups," ISCA-24, 1997.

[29] D. Tarjan, S. Thoziyoor, and N. P. Jouppi, "CACTI 4.0," HP Laboratories Palo Alto, HPL-2006-86 Technical Report, June 2006.