SPARTAN: Speculative Avoidance of Register Allocations to Transient Values for Performance and Energy Efficiency

Deniz Balkan   Joseph Sharkey   Dmitry Ponomarev   Kanad Ghose
Department of Computer Science,
State University of New York,
Binghamton, NY 13902-6000
{dbalkan, jsharke, dima, ghose}@cs.binghamton.edu

ABSTRACT

High-performance microprocessors use large, heavily-ported physical register files (RFs) to increase the instruction throughput. The high complexity and power dissipation of such RFs mainly stem from the need to maintain each and every result for a large number of cycles after the result generation. We observed that a significant fraction (about 45%) of the result values are never read from the register file and are not required to recover from branch mispredictions. In this paper, we propose SPARTAN – a set of micro-architectural extensions that predicts such transient values and in many cases completely avoids physical register allocations to them. We show that the transient values can be predicted as such with more than 97% accuracy on the average across simulated SPEC 2000 benchmarks. We evaluate the performance of SPARTAN on a variety of configurations and show that significant improvements in performance and energy-efficiency can be realized. Furthermore, we directly compare SPARTAN against a number of previously proposed schemes for register optimizations and show that our technique significantly outperforms all those schemes.

Categories and Subject Descriptors
C.1 [**Processor Architectures**]: Other Architecture Styles – *Pipeline processors.*

General Terms: Performance, Design

Keywords: Register Files, Energy-Efficiency

1. INTRODUCTION

High-performance processor designs maintain a significant amount of speculative state in the form of the register values produced by the functional units along the predicted execution paths. In modern microarchitectures, this speculative state is typically embodied in physical register files (RFs), where a physical register allocated for the destination of an instruction is deallocated only when the next instruction writing to the same architectural (logical) register commits. Such a conservative register management guarantees that until all instructions between the two consecutive definitions of the same architectural register commit, the earlier definition is available and can be resurrected should the later definition be squashed as a result of a branch misspeculation, exception or interrupt. Intel's Pentium 4 [18], MIPS 10000 [44] and Alpha 21264 [20] implement the register files in this manner. While significantly simplifying the recovery to a precise state, this arrangement increases the register pressure and effectively mandates the use of larger RFs if pipeline stalls due to the lack of physical registers are to be avoided.

As the number of entries in the RF increases, the access to the RF (which can potentially limit the cycle time [4], [24], [22], [11], [33]) is likely to require multiple cycles. RFs with multi-cycle access delay increase the complexity of the bypass networks and also degrade the IPCs due to the lengthened branch resolution and load-hit speculation loops. Finally, large register files also dissipate significant amount of power. The power dissipated in the register file has been reported to account for 10% to 25% of the total chip power [2], [4]. The situation is further exacerbated in the SMT processors, where the pressure on the register file is increased as larger physical register files are needed to support multiple thread contexts.

An alternative to using large RFs is to use smaller number of registers, but manage them more effectively. A number of techniques for optimizing the RF usage were recently proposed. One set of solutions [24], [27], [30], [32] targets early register deallocation. In all of these schemes, however, each and every result value is still written into the register file and the validity of the physical register is used as one of the conditions for its early deallocation. Another set of solutions targets delayed register allocation. These schemes [15], [43] avoid tying up destination physical register between the time of instruction dispatch and instruction writeback by allocating physical registers only at the time of writeback and using separate virtual tags to maintain the data dependencies. Previous proposals have significant complexities as they require systematic tag re-broadcasts and associative searches on every reassignment of virtual tags to physical registers, i.e. for every instruction with a destination register. In this paper, we introduce an aggressive register management scheme that avoids the register allocations and the result writebacks to almost half of the generated result values *without* incurring many of the complexities inherent in the previous proposals.

The specific contributions and the key results of this paper are:

- We formally define *transient* values – results that are not read from the register file and are not required to recover from branch mispredictions. We show that 45% of all generated values can be classified as transient across the SPEC 2000 benchmarks.
- We propose *SPARTAN* - a set of microarchitectural mechanisms to predict transient values and avoid physical register allocations to them. We show that across the SPEC 2000 benchmarks, the accuracy of predicting such values is in excess of 97%. Compared to the previously proposed late register allocation mechanisms, our technique eliminates many of the design complications, such as the need to perform frequent tag re-broadcasts as well as expensive associative searches on the rename table and the issue queue for every register reassignment – these activities are never incurred in our proposal.
- We evaluate the impact of our mechanisms on the performance as well as the energy dissipations within the RF. We show that the aggressive nature of register management in SPARTAN results in significant IPC gains, as much as 28% on the average across SPEC 2000 benchmarks. The RF energy is reduced because the writes of *transient* values are avoided. Including dissipations in the additional logic required by our technique, such selective writeback of values saves as much as 35% (with 25% on the average) of the energy dissipated in the register file.
- We compare the performance of SPARTAN against some previously proposed schemes for register file optimizations and show that our mechanisms significantly outperform the previous solutions for the majority of the benchmarks as well as on the average.

The rest of the paper is organized as follows. We review the related work in Section 2. In Section 3, we formally define transient values and motivate the rest of the paper. In Section 4, we describe the implementation details of SPARTAN. Section 5 presents our simulation methodology. We present and discuss the simulation results in Section 6, and offer our concluding remarks in Section 7.

2. RELATED WORK

Several techniques have been proposed in the recent literature to reduce the energy requirements of the register files and improve the efficiency of register file usage. These can be broadly classified into several categories: minimizing the number of registers, reducing the number of register ports, using various register file caching schemes and multi-banked register files. Researchers have exploited the inefficiencies in register usage to reduce the number of registers in three major ways. One set of solutions delays the actual allocation of physical registers until the time that the result is written back, [15], [43]. These schemes avoid tying up destination physical register between the time of instruction dispatch and instruction writeback by allocating physical register only at the time of writeback and using separate tags to satisfy the data dependencies. The major drawback of the late allocation schemes is in the form of non-trivial increases in the datapath complexity due to the need to: (a) support several levels of register mapping tables, (b) perform various associative searches on the rename table and issue queue after the reassignment of mappings and (c) avoid potential deadlocks. Delayed physical register allocation was also used in [33] to reduce the conflicts over the write ports in a multiple-banked register file.

The second set of techniques aim at reducing the register file pressure by using the early deallocation of physical registers [12], [24], [27], [30], [32]. In all of these works, each and every generated result is written into the register file and validity of the register value is one of the conditions for the earlier register deallocation. The Physical Register Inlining technique of [24] embeds narrow-width results directly within the rename table and early deallocates corresponding physical registers. Energy reduction was not the goal of the previous proposals for the early deallocation of registers.

While it is conceivable that the techniques for late register allocation and early register deallocation can be used in conjunction with each other [29], the complexity of such a synergy is likely to be prohibitively high, as both classes of techniques involve significant additional logic. Furthermore, even if such a combination of techniques is implemented, the physical register will still have to be allocated for each instruction (possibly for just a few cycles), because the validity of the physical register is a necessary condition for its early deallocation in all previously proposed schemes. The third set of solutions reduces the number of registers through the use of register sharing [3], [19], [36], [41]. A unified framework for several techniques to collapse instructions from the dynamic stream and relink dependencies among them using register sharing was proposed in [34].

Alternative register file organizations (mainly using various forms of caching) have also been explored for reducing the access time (which goes up with the number of ports and registers), particularly in wire-delay dominated circuits [5], [11], [8]. In [26], register file usage was optimized using compiler support to exploit dead value information. A compiler-assisted early register release is explored in [48]. In [13], a technique to pack multiple narrow-width results into the same physical register is proposed to reduce register file pressure. Another technique to exploit narrow width operands to reduce the area, access time and energy consumption of the register file was proposed in [23]. The concept of non-blocking register file, where instructions waiting for long-latency events (such as caches misses) do not tie up physical registers, was introduced in [40]. The concept of partial value locality was exploited for reducing the register file power, area and delay in [16].

In [6], the dataflow mini-graphs (groups of instructions with certain properties) are processed in a datapath as a single instruction, thus the writeback of the intermediate (transient) values that connect only the instructions within the mini-graph is avoided. The technique relies on a binary rewriting tool to modify the executable and statically replace dataflow graphs that satisfy mini-graph criteria with handles. In this work, we propose a purely dynamic technique to identify transient values and avoid their writeback to the register file. Sassone *et. al.* [37] attempt to create *strands* in hardware using a trace-cache-like mechanism. Once the strands have been created, they enter the pipeline as a collapsed atomic operation.

3. MOTIVATION AND DEFINITIONS

It has been noticed by several researchers, that most of the register instances in a datapath are short-lived [14], [25], [35]. Following the work of [35], we define a value to be ***short-lived*** if the architectural register allocated as a destination of the instruction X has been renamed before the value generated by X is written back. We further define the instruction that renames a register allocated to hold a short-lived value as the ***renamer***. In our simulations of the SPEC 2000 benchmarks, about 86% of all generated register values were identified as short-lived.

We define a produced result value as ***transient*** if the following conditions are true:

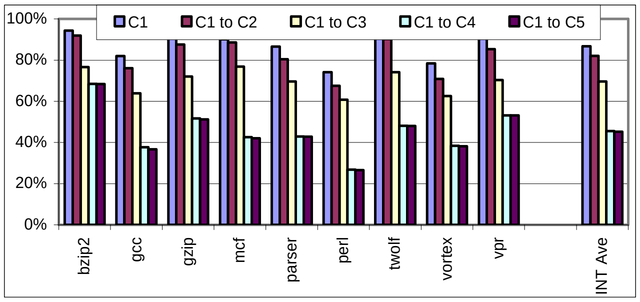
C1) The value must be short-lived.

C2) There must be no more than one instruction that consumes the value.

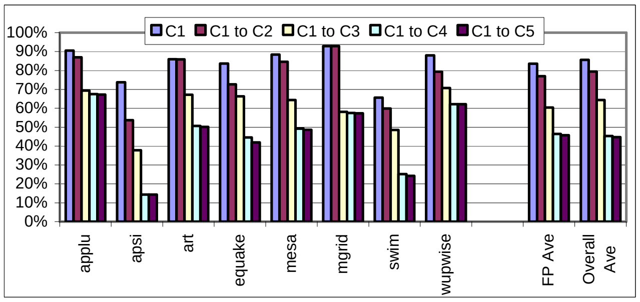
C3) The consuming instruction must issue before the value is produced – this ensures that the value is obtained off of the bypass network.

C4) There must be no branch instructions between the value-producing instruction and its renamer.

C5) The sole consumer of the value must not be subject to a replay caused by a load latency misprediction or a memory dependence misprediction.



(a)



(b)

Figure 1. Various statistics about short-lived values for integer (a) and floating-point (b) benchmarks.

Figure 1 shows the percentage of transient values across the execution of SPEC 2000 benchmarks. The details of our simulation framework are given in Section 5. The bars from left to right correspond to the cumulative percentages of the various conditions (C1 through C5) described above. For example, the leftmost bars show the percentage of all generated values when condition C1 is valid, the next set of bars shows the percentage of cases when both conditions C1 and C2 are satisfied and so on. The rightmost set of bars depicts the percentage of transient values, as all 5 conditions are satisfied. On the average across all benchmarks, about 45% of the produced results are transient.

This implies that almost half of the generated results are not read from the register file, are not needed to recover from a branch misprediction, and the consumers of these results are not subject to any memory replay traps. The only reason to store these values in the register file is to allow for the reconstruction of the precise state after exceptions or interrupts.

The goal of this paper is to introduce mechanisms that predict such transient values and completely avoid physical register allocations for them. The transient values are simply "dropped" from the datapath right after their generation. The overall scheme described in this paper is called SPARTAN - *SP*eculative *A*voidance of *R*egister allocations to *T*r*AN*sient values - to reflect the austere nature of using registers. In the next section, we describe the implementation details of SPARTAN.

4. IMPLEMENTATION OF SPARTAN

First, we discuss the predictability of transient values and show the prediction accuracies for SPEC 2000 benchmarks. Second, we describe mechanisms for maintaining data dependencies without performing physical register allocations. Third, we provide the details of the logic used for verifying the predictions. Finally, we show how the mispredictions are handled and the precise state is maintained.

4.1 Predicting Transient Values

Transient Value Prediction (TVP) (predicting if a value is transient or non-transient) can be implemented through the use of 1 bit for the I-cache entry of each instruction. These bits are maintained in a physically separate structure than the I-cache but are indexed in a similar manner to and in parallel with the I-cache. On an I-cache miss, the *default prediction of non-transient* is used. This prediction is revised when the result is actually computed. Of course, alternative prediction schemes that use prediction tables indexed separately from the I-cache can also be used, but we will show in Section 3.3 that our choice of predictor actually helps to simplify the process of detecting and handling mispredictions.

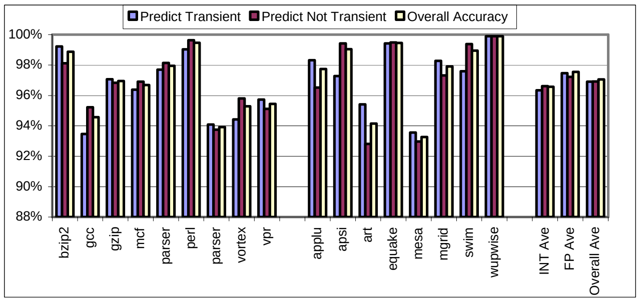


Figure 2. Accuracy of the Transient Value Predictor.

Figure 2 depicts the accuracy of TVP. Three sets of bars are depicted for each benchmark. The rightmost set of bars shows the overall prediction accuracy. The prediction accuracy is 96.6% for the integer benchmarks, 97.6% for the floating point benchmarks and 97.1% across all programs. The other two bars present additional statistics. The leftmost set of bars shows the percentage of cases where the value is predicted to be transient and the actual outcome matches the prediction. We isolated this scenario since the mispredictions occurring in this case will necessitate misprediction recovery actions (as described later in this section). As seen from the graphs, such mispedictions occur only in about 3.1% of the cases when the value is predicted as transient but turns out to be non-transient. The set of bars in the middle presents the accuracy of the predictions where the value is predicted to be not transient. Since the mispredictions occurring in these cases do not require any special handling (the value will simply be dropped), the performance is not impacted.

4.2 Handling Data Dependencies

If a value to be produced by an instruction is predicted to be transient, then instead of allocating a physical register to hold this value, a separate virtual tag is allocated and used to preserve the data dependencies. The total number of **virtual tags** is equal to the number of physical registers. In the rename table, the virtual tag is distinguished from the physical register identifier by the most significant bit of the address – the most significant bit is always "0" for the physical register addresses and "1" for the virtual tags. Finally, separate free lists are used for regular physical registers and the virtual tags.

If it is predicted that an instruction (say, instruction I) will generate a transient value, the destination architectural register of I is renamed to a virtual tag allocated from the free list of virtual tags (say, virtual tag V). The rename table is updated accordingly, appending "1" as the most significant bit, as explained above. If no free virtual tags are available but a free physical register is available, then a physical register is allocated. Subsequent instructions, which are data dependent on I, will use V (obtained from the rename table) as the wakeup tag for the corresponding source register. When instruction I is selected for the execution, its destination tag V is broadcast across the issue queue and all the dependent instructions, which use V as their source tag, are awakened.

4.3 Verifying the Prediction

Before the writeback, a TVP prediction made during the instruction fetching needs to be verified. If it is determined that the value produced by I is indeed transient and the prediction was correct, then the produced value is simply dropped and the virtual tag V is added to the free list of virtual tags. This is the ideal (and the most frequent) scenario. In Section 4.4 we describe mechanisms for handling TVP mispredictions. In the rest of this section, we describe the logic for validating the predictions (actually detecting transient values).

We first examine the hardware necessary to detect the first four conditions (C1 through C4) for transient values and show how it can be implemented with only 3 bits per physical register if the initial prediction is non-transient. For values which are predicted to be transient (and for which a virtual tag rather than a physical register was allocated), we then show how this requirement can be reduced to only one bit per virtual tag. Finally, we comment on how the condition C5 can be supported in both cases.

To verify the predictions for values predicted as non-transient, 3 bits per physical register are needed to verify the prediction by actually detecting transient values at the time of writeback. When a physical register is allocated, all the bits corresponding to this register are cleared. The first bit, which we call *Transient*, is used to drive the final decision for dropping the produced values. When an instruction is renamed, it sets the *Transient* bit corresponding to the previous mapping of its destination architectural register to one if no intervening branches were renamed between these two instructions. To detect the absence of intervening branches, another bit (called *Branch_Bit*) is used per physical register. When a physical register is allocated, its corresponding *Branch_Bit* is reset to zero. When a branch instruction is renamed, it gang-sets the branch bits of *all* physical registers to one. Thus, to account for the condition C4, the instruction only sets the *Transient* bit of its previous mapping (say, P) to one if the *Branch_Bit[P]* is set to zero. This logic therefore detects conditions C1 and C4 from Section 2.

The addition of one more bit (called *Consumer_Bit*) per physical register allows for the detection of the conditions C2 and C3. When a consumer of register P is renamed, the *Consumer_Bit[P]* is incremented. The circuitry is designed in such a way that this one-bit counter actually overflows into the corresponding *Branch_Bit*. Therefore, in cases where there are multiple consumers of the value, the value of the *Branch_Bit* will be set to 1, effectively preventing the value from being categorized as transient.

To determine if the sole consumer of a potentially transient value has issued by the time the value is written back, we again utilize the existing *Consumer_Bits*. When an instruction is selected for execution and moves to the stage(s) where it starts the register file access or obtains the source(s) via bypassing, the same source register address bits that are used for reading or bypassing are used to clear the *Consumer_Bits* associated with both of its source registers. Most contemporary and emerging datapaths have a multi-cycle delay between the issue and the execution stages, to permit this timing requirement to be met naturally.

In summary, to detect if the value produced by an instruction with destination physical register P is transient, the *Transient[P]* bit is checked one cycle before the producing instruction enters the writeback stage. If the bit is set, then the produced value is actually transient, it can be dropped without performing a writeback to the register file and the corresponding destination physical register can be deallocated immediately. Note that no additional manipulations with the extra bits need to be performed on a branch misprediction. The destination registers of all flushed instructions are deallocated and when the new allocations are made, the bits are cleared automatically. If the old mapping of a flushed instruction's destination is marked as *Transient*, then the instruction producing the value into this old mapping is also flushed (because the mispredicted branch precedes both the value-producing instruction and its redefiner). Otherwise, only the redefiner is flushed, but the value is not marked as Transient and will not be dropped anyway.

While similar hardware (3 bits per virtual tag) can be used to verify the predictions if the initial prediction is "transient", we now explain how this requirement can be reduced to only one bit per virtual tag. A one-to-one correspondence between the I-cache words and the prediction bits avoids aliasing in the predictor, and therefore the information built in the predictor can be used to glean the history of specific instruction and avoid repetitive checking of several conditions. Specifically, if an instruction was previously determined to produce a transient value (and it was not evicted from the I-cache), then during the execution of the next dynamic instance of this instruction, the conditions C2 (related to the number of consumers) and C4 (the absence of branches) will always be satisfied and they do not need to be checked. Furthermore, the checking of the condition C1 can also be avoided, as it normally only needed to check C2 and C4. Consequently, the only condition that needs to be checked dynamically for every instruction is C3. For this, we maintain one bit per virtual tag (called *Consumer_bits*). These bits are set to 1 when a virtual tag is allocated, and are cleared when the consumer of the value issues (analogous to the *Consumer_bits* for each physical register). If the *Consumer_bit* corresponding to the virtual tag of a value that was predicted to be transient is 0 when the instruction reaches the writeback stage, then the prediction was correct; otherwise, there was a misprediction. This optimization reduces the power consumption in the course of prediction verifications and reduces the percentage of cases where the results are predicted to be transient but turn out to be non-transient to only 2.1%. This is important, as only these types of misprediction require additional actions to guarantee correctness of the execution.

Finally, we address the support for condition C5. In processors that schedule instructions based on the load latency prediction [20], the *Consumer_Bits* are decremented speculatively at the time of scheduling. To support condition C5, the *Consumer_Bits* must be reset by the instructions that are replayed as part of the recovery process following a load-latency misprediction [20]. In addition, no value is dropped (even if it is determined as transient) during the time when such recovery takes place. To detect transient values in processors that use memory dependence prediction [10], we use a technique similar to what is described in [27]. Specifically, we ensure that the consumer of a transient value is older than the oldest store instruction with unknown address. As shown in Figure 1, the percentage of values identified as transient is reduced only slightly by imposing this additional condition. In our evaluations, we model both types of memory-related speculations.

4.4 Handling TVP Mispredictions

In this section, we explain how the TVP mispredictions can be handled without resorting to tag re-broadcasts, register re-mappings and/or rename table updates, as required by all previously proposed schemes for late register allocation. The basic idea is to rely on a small (we assume 6-entry in the rest of the paper) fully-associative buffer (called *TVB – Transient Value Buffer*) to temporarily hold values that were predicted to be transient, but were mispredicted.

Each TVB entry is composed of two fields: the virtual tag (as defined above) and the data. Whenever a TVP misprediction occurs, the result value of the mispredicted instruction, along with the virtual tag that was used in place of the corresponding destination physical register address, are written into a free slot in the TVB.

Every issued instruction can now obtain its source operands from one of the three different origins: a) the register file; b) the bypass network, or c) the TVB. For the consumers of transient values (for which a virtual tag and not a physical register was assigned), the choice is limited to the bypass (in case of correct prediction) and the TVB (in case of a misprediction). This access is performed analogously to traditional operand access, where the data can either be obtained off of the bypass network or directly from the register file. The small size of the TVB and its consequential low access time (compared to the register file) naturally permit the timing requirements to be met.

For accessing the TVB, the consumer uses its virtual tag as a search key and, on a match, reads the corresponding data. The TVB entry is deallocated immediately after the data is read by its sole consumer (all values in the TVB were predicted transient and satisfy condition C2 and therefore have only one consumer).

Two factors make it possible to maintain a very small TVB size: 1) the small percentage of TVP mispredictions, and 2) the short lifetimes of TVB entries (from writeback of the producer to the issue of the consumer). However rare, the cases where the TVB is full on a TVP misprediction have to be handled. In these very rare cases, we rely on checkpointing and simply rollback to the most recently established checkpoint. If such a rollback occurs, SPARTAN mode is automatically disabled until the next checkpoint is created to avoid multiple rollbacks to the same checkpoint if conditions re-occur and thus guarantee forward progress.

The key observation that makes this handling of TVP mispredictions efficient is that the number of such rollbacks is very small, making it possible to incur the rollback penalty when the TVB is full. In our simulations, the largest number of rollbacks were observer for *mgrid* benchmark (116537 rollbacks for 500M instructions), the rollbacks in the rest of the benchmarks were much less frequent. In fact, in some cases (*mcf, perl, equake, swim,gzi, and wupwise*), the rollbacks never occurred. In any case, the number of rollbacks can always be further reduced by using slightly larger TVB.

Notice that in SPARTAN, as opposed to the previously proposed schemes for the late register allocation (such as [15] and [43]),a virtual tag *never* has to be remapped to a true physical register, and no update of the issue queue or the rename table through expensive tag re-broadcasts and associative searches is *ever* needed. This is a direct consequence of the conscious effort to allocate virtual tags only to the transient values. In contrast, in [15] and [43] the virtual registers are *systematically* allocated to *all* value-producing instructions and they are all remapped to the physical registers after the instruction execution completes. This involves expensive tag re-broadcasts and additional associative searches within the issue queue and the rename table for *every dynamic instruction with a destination physical register.*

When some produced values are discarded right after their generation, traditional ROB-based mechanisms for restoring the precise state can no longer be used. Therefore, to maintain the precise state in the event of exceptions or interrupts, we rely on periodic checkpoints of the processor state. (We note here that branch mispredictions are handled just as in traditional designs due to the condition C4 as described in Section 2).

The periodic precise state of the register file is created in the following fashion. At any arbitrary cycle, the creation of a checkpoint can be initiated by disabling SPARTAN mode, forcing the allocations of physical registers to *all* newly renamed instructions with destination registers. This continues until all architectural registers are mapped to physical registers, i.e. none of the architectural registers is mapped to a virtual tag. (This can be easily done be keeping a counter that is incremented when an architectural register is mapped to a virtual tag and is decremented when a mapping changes from a virtual tag to a physical register. When the counter value is zero, the condition is satisfied). At this point, the youngest renamed instruction is marked as checkpoint initiator and when it commits, the actual register file and commit-time rename table checkpoints can be taken. Once the checkpoint initiator is found and marked, SPARTAN mode can be re-enabled immediately. If the checkpoint initiator is flushed out of the pipeline on a branch misprediction, SPARTAN mode is disabled again and the new initiator (from the correct path) is chosen. Notice that even when SPARTAN mode is disabled, the training of the transient value predictor still continues to increase prediction accuracies in SPARTAN mode.

If the checkpoints are created too frequently, then the benefits of SPARTAN will be reduced, as SPARTAN mode will be often disabled. However, if the intervals between consecutive checkpoints are too large, then a significant amount of cycles will be lost on the rollbacks caused by exceptions, interrupts, or some TVP mispredictions. Though extensive simulations, we established that the optimal checkpointing interval is 500 cycles, and we use this value in all our experiments.

We also simulated the effects of exceptional events, on the performance of SPARTAN. Our experiments indicated that unless exceptions occur exceedingly often, such as a page fault occurring once every 1000 memory operations (which is, of course, an unrealistically high rate), the performance is very close to that of the system with no exceptions.

The specific circuit implementation of register file checkpointing is not central to the ideas of this paper. One can either use a separate register file for this purpose (as in [27]), or embed the checkpoint within the register file itself, by backing up each bitcell with a shadow copy [12]. For our evaluations, we assumed the Checkpointed Register File (CRF) design proposed in [12], where each bitcell is backed up by a shadow cell. As shown in [14], such checkpointing can be implemented with minimal overheads in terms of area, power and delays. When the checkpoint is created, the contents of a bitcell are simply copied to the shadow cell. To recover, the contents of the shadow cells are copied back to the main storage. In our simulations, we fully accounted for these overheads of the checkpointed register file. In our baseline model, the traditional register file (without any checkpointing capabilities) was assumed.

To buffer a large number of store instructions between two consecutive checkpoints, we use the approach described in [27] and also used in a few others works. The values are stored within the local cache hierarchy, but their propagation to the main memory is avoided until it is safe to do so. Each cache line updated in this manner is marked *Volatile*, using one extra bit for each cache line. When a processor needs to rollback to a checkpoint, all cache lines marked *Volatile* are invalidated using the gang-invalidate signal. When the precise state is created (as described in the previous section), the Volatile bits are cleared. The idea of maintaining some speculative state in the cache hierarchy was also used in [17]. In our design, we handle the memory updates in the same manner as they are handled in [27]. We expect emerging high-end processors to rely on the use of such volatile bits for other aspects of performance, such as supporting transactional memory. A recent paper [47] also describes how to correctly incorporate caches with the volatile lines into a multiprocessor system.

When an exception occurs, the processor state is recovered to the previous checkpoint and the SPARTAN mode is disabled during the re-execution, forcing the register allocations to all values until the next checkpoint is created. This ensures that the precise state is available for each instruction during the re-execution and, should the exception re-occur, this state is first used to create a new checkpoint and is then used to handle the exception, after which the SPARTAN mode is re-anabled. Note that some exceptions, such as page faults, may not reoccur during the process of re-execution. For these cases, the SPARTAN mode is re-enabled after the next periodic checkpoint is formed.

5. SIMULATION METHODOLOGY

For estimating the energy savings and the performance gains achieved by using SPARTAN, we used a significantly modified version of the Simplescalar simulator [7] that explicitly models the issue queue, the reorder buffer, the load/store queue, the register renaming logic and other out-of-order execution mechanisms associated with a datapath where a unified register file is used. Register file is read after the instructions are issued.

Table 1. Configuration of the Simulated Processor.

Parameter	Configuration
Machine width	4-wide fetch/issue/commit
Window size	64-entry IQ, 64 entry LSQ, 128–entry ROB
Registers	Various sizes studied, as indicated
Function Units and Latency (total/issue)	4 Int Add (1/1), 1 Int Mult (3/1) / Div (20/19), 2 Load/Store (2/1), 4 FP Add (2), 1FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
L1 I–cache	32 KB, 2–way, 32 byte line, 1 cycle hit
L1 D–cache	32 KB, 4–way, 32 byte line, 2 cycles hit
L2 Cache	512 KB, 4–way, 128 byte line, 8 cycles hit
BTB	1024 entry, 4–way set–associative. Minimum branch misp. penalty – 10 cycles
Branch Predictor	Combined with 1K entry Gshare, 10 bit gl. hist, 4K entry bimodal, 1K entry selector
Memory	128 bit wide, 120 cycles
TLB	64 entry (I), 128 entry (D), fully associative

The studied processor configuration is shown in Table 1. For load-latency prediction, we used the load hit/miss predictor which was used for the Alpha 21264 processor [20]. A 5-bit saturating counter is used for each entry where the counter is incremented by 1 in case of hit and decremented by 2 in case of miss. Load instructions are predicted to hit in the cache if the most significant bit of the counter is 1. We also used the store set predictor described in [10] for speculating on memory dependencies.

We used 9 integer SPEC 2000 benchmarks (*gcc*, *gzip*, *parser*, *perlbmk*, *twolf*, *vortex*, *mcf*, *bzip* and *vpr*) and 8 floating point SPEC 2000 benchmarks (*applu*, *art*, *mesa*, *mgrid*, *swim*, *apsi*, *equake* and *wupwise*). We had difficulties compiling the other benchmarks (mostly those written in Fortran) in our simulation framework. Benchmarks were compiled using the Simplescalar GCC compiler (with –O4 optimizatons) that generates code in the portable ISA (PISA) format. Reference inputs were used for all the simulated benchmarks. The results from the simulation of the first 1 billion instructions were discarded and the results from the execution of the following 500 million instructions were used.

For estimating the energy dissipated in the course of accessing the register files, the event counts gleaned from the simulator were used, along with the energy dissipations, as measured from the actual hand–crafted VLSI layouts using industry-standard Cadence® design tools. CMOS layouts for the register files and the bit–vectors in a 0.18 micron 6 metal layer process (TSMC) were used to get an accurate idea of the energy dissipations for each type of transition.

6. RESULTS AND DISCUSSIONS

Figure 3 shows the IPC values for the baseline machine and the machine with SPARTAN scheme with 64 entry RFs using various sizes of the transient value buffer (TVB). As smaller number of TVB entries result in a high rate of rollbacks, SPARTAN with only one TVB entry performs worse than the baseline machine. When the number of entries in TVB is set to 2, SPARTAN outperforms the baseline for integer benchmarks but not floating point benchmarks. Starting from 4 TVB entries SPARTAN outperforms the baseline machine both in integer benchmarks and floating point benchmarks, but further analysis showed that this configuration does not scale well. SPARTAN scheme with a 6-entry TVB not only performs better than the baseline for both integer and floating point benchmarks but also scales well while still keeping the overhead minimal, therefore for the rest of our analysis we use a 6 entry TVB.

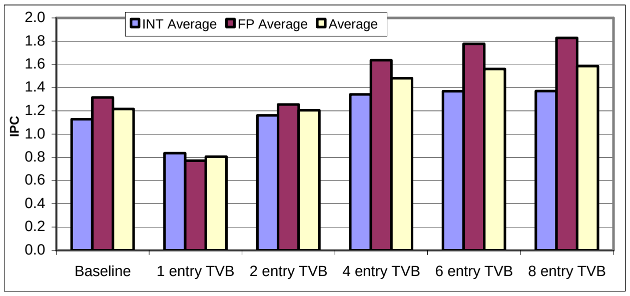


Figure 3. IPC values for the baseline versus the SPARTAN scheme with different number of TVB entries.

Figure 4 shows the IPC improvement achieved by using the SPARTAN scheme on a datapath with 64-entry physical register files compared to a datapath with the infinite number of physical registers.

The first set of bars in Figure 4 present the IPC performance of the baseline machine. The second set of bars in the figure depicts the IPC values obtained by using the SPARTAN scheme. The SPARTAN scheme achieves the IPC improvement of 28% on the average compared to the baseline machine, and some benchmarks show especially remarkable gains: 82.7% for *mgrid*, 76.6% for *bzip* ,and 57.8% for *wupwise*, and Finally, the last set of bars shown on Figure 4 depicts the performance of a machine with the infinite number of physical registers. As seen from these results, the machine with 64 physical registers using SPARTAN comes as close as 11.1% to the performance of the baseline machine with infinite number of registers.

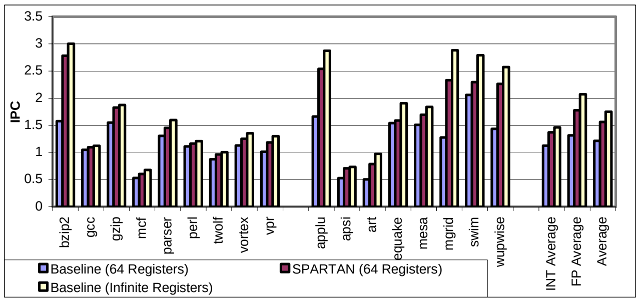
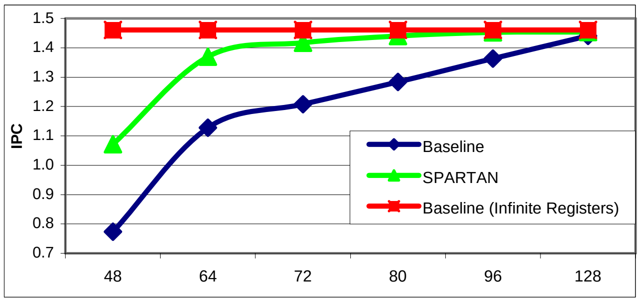


Figure 4. IPC improvement of SPARTAN over the baseline processor with 64-entry integer and floating point RFs compared to the processor with infinite number of registers.

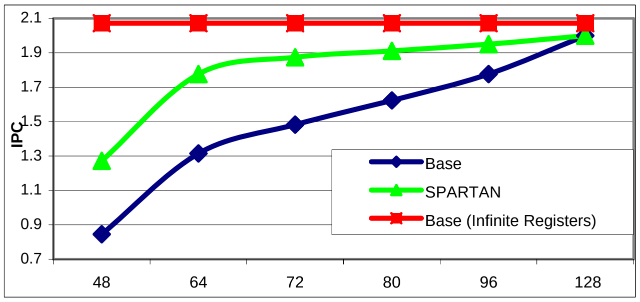
Figure 5 depicts the commit IPC values for the baseline machine, the SPARTAN scheme for register files ranging from 48 to 128 entries in each of the integer and floating point RFs. The commit IPC values are presented for the averages across SPEC 2000 Integer benchmarks (Figure 5 (a)), the averages across SPEC 2000 Floating Point benchmarks (Figure 5 (b)), and the overall averages (Figure 5 (c)). Similar trends can be observed in all three graphs. For the same number of physical registers, the SPARTAN scheme results in the following IPC improvements over the baseline machine on the average across all simulated benchmarks: 42.7% for 48 registers, 28% for 64 registers, 21.2% for 72 registers, 13.6**%** for 80 registers, and 7.8**%** for 96 registers. Note that with 128 registers, only 0.5**%** IPC improvement is possible, this is because the baseline with 128 registers already performs almost as well as the one with infinite number of registers, i.e. with 128 registers there are very few stalls due to the lack of physical registers.

We now compare the number of registers needed by the various schemes to achieve the same IPC performance target. As shown in Figure 5 (c), the performance of the baseline machine with 96-entry RFs is matched by the SPARTAN scheme with 64-entry RFs. Consequently, for the same IPC, the use of fewer registers with SPARTAN can reduce the wire delays and possibly decrease the cycle time, if the register file access lies on the critical timing path. The quantification of these additional potential advantages of SPARTAN is beyond the scope of this paper.

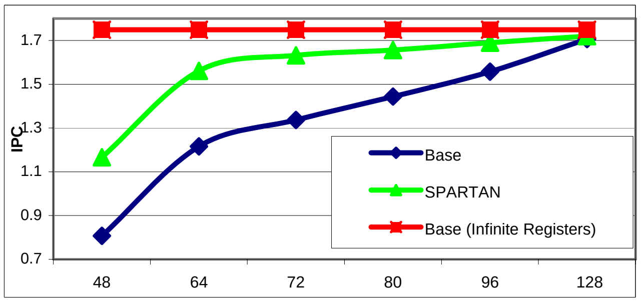
Figure 6 compares the performance of SPARTAN with some previously proposed techniques for optimizing register files. These alternative schemes were described in Section 2. In the figure, the first set of bars shows the performance of the baseline case. The second set of bars shows the IPCs of the scheme of [12], which deallocates the physical registers at the time of commitment or at the time of renaming the redefining instruction. We refer to this scheme as Ergin_ICCD to reflect the first author name and the publication venue. The third set of bars depicts the IPCs of the Virtual Physical Registers scheme of [15]. We refer to it as VPR in the rest of the paper. For the VPR scheme, we assume that 32 reserved registers are in use – the best configuration suggested in [15]. The next set of bars shows the performance of Physical Register Inlining [24], where narrow-width register values are stored directly within the rename table. Finally, the last set of bars shows the performance of SPARTAN. We have implemented all these techniques in our simulation framework and therefore can compare the results on an even footing.



(a)



(b)



(c)

Figure 5. IPC values for the baseline machine and the SPARTAN scheme with various register file sizes – (a) Integer averages, (b) Floating Point averages, (c) Overall averages.

On average, SPARTAN outperforms baseline by 28%, compared to 5.3% for Ergin_ICCD scheme, 7.7% for VPR and 10.2% for PRI. It is interesting to notice that SPARTAN outperforms the previous schemes on most of the benchmarks, but there are some exceptions. For example, PRI performs better than SPARTAN for *swim*. This is because a larger percentage of narrow-width values can be stored in the rename table for this benchmark, than the percentage of values that can be detected and predicted as transient. For *vortex*, VPR scheme performs better than SPARTAN and for *swim*, Ergin_ICCD scheme also outperforms SPARTAN. In all other cases, SPARTAN shows superior performance compared to other techniques. It is also interesting to observe either VPR or PRI perform better for various benchmarks with respect to each other.

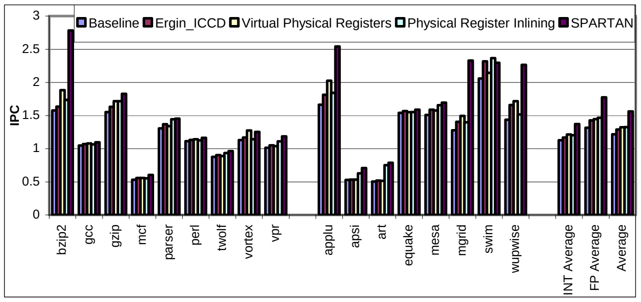


Figure 6. IPC comparison of the SPARTAN scheme against some of the previously proposed register optimization techniques

SPARTAN has particularly high IPC gains on *bzip, art, applu, wupwise* and *mgrid*. This is due to a high percentage of transient values and high transient value prediction accuracy on these benchmarks (see Figures 1 and 2 for comparison).

6.1 Energy Considerations

We first present the energy implications of SPARTAN on the register file, ignoring the dissipations in the auxiliary structures, such as the additional bit-vectors needed for prediction verification, as described in Section 4.3. We then take the dissipations of those components into consideration.

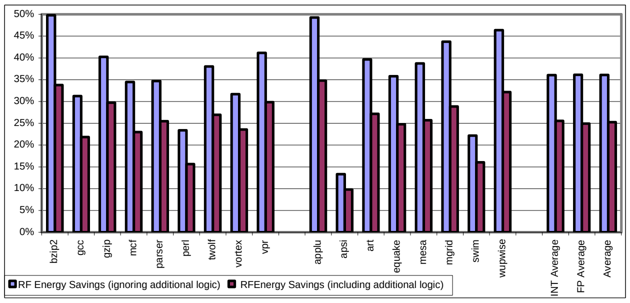


Figure 7. Percentage of energy savings within the RF.

First, we consider the reduction in the overall energy dissipations (or energy per task). Figure 7 presents the energy reduction achievable within the register file if the SPARTAN scheme is used. The comparison is given for a datapath with 64 integer and 64 floating point registers. There are two bars depicted in the figure, the bar on the left presents the energy savings within the register file itself when the dissipations in the additional logic required by SPARTAN are ignored. On the average, SPARTAN reduces the RF's energy consumption by 36% - this number also accounts for small overhead of using shadow bitcells in SPARTAN's register files. The energy reduction on individual benchmarks ranges from 50% for *bzip* to 13% for *apsi*. In the presented results, we assumed that the energy reduction is only a consequence of the smaller number of writebacks. The bar on the right depicts the energy reduction in the RF, if the dissipation in all additional bit-vectors and arrays required by the SPARTAN scheme are taken into consideration (the additional energy dissipated in the caches due to the presence of Volatile bits in the D-cache and the prediction bits in the I-cache is also accounted for). Even with the additional energy dissipation, there is still a reduction of 25% in the RF energy. The overall processor energy savings depends on the percentage of energy attributed to the RF. The range of these savings will be between 2.5% and 7% (considering that RFs contribute between 10% and 25% to the overall processor energy, as mentioned in the introduction).

To understand why the reduced number of writebacks has such a significant impact on the register file energy dissipation, it is useful to examine the numbers presented in Table 2. Although the total number of reads from the register file is higher than the number of writes, the write energy per access is larger than the read energy per access by about 1.8 times, as detailed in Table 2. The circuit simulations of the actual full custom register file layouts reveal that the energy dissipated in the course of driving the write lines plus the energy dissipated in the course of changing the contents of the SRAM bitcell significantly exceeds the energy dissipation of sense amps and prechargers used during the read accesses to the register file (Table 2). In the interests of low power, we precharged the read bit lines to Vdd/2.

Table 2. Energy dissipation within the RF components for 64-entry RFs

	Read Energy (fJ)	Write Energy (fJ)
Decoder	724	724
Word Select	1242	1242
SenseAmp	14592	-
Precharger	9184	-
Bitcell	3456	8800
Write Driver	-	42473
Total	29198	53239

Finally, we also evaluated the energy savings within the RF, if SPARTAN machine is configured to provide the same IPC performance as the baseline processor. As detailed earlier in the section, SPARTAN with 64-entry RFs provides the same IPC performance on the average as the baseline machine with 96-entry RFs. However, higher energy savings can be achieved comparing these configurations, as in addition to having fewer writebacks, a smaller RF size also further decreases the energy per access. On the average, the energy savings (as well as the power savings since the performance is the same) are 66% in this case if the additional logic is ignored and 46% if the additional logic is accounted for.

We do not directly compare the energy advantages of SPARTAN with the previously proposed schemes for optimizing register files, because energy reduction was not the goal of those techniques. With those schemes, it is possible that the additional structures and/or data movements required to support the complicated register management will outweigh the energy advantages of having somewhat smaller register files. The energy advantages of SPARTAN come from avoiding a significant fraction of writebacks to the register file, which is a unique feature of our solution.

In summary, SPARTAN achieves higher performance and lower energy consumption per instruction (or per task) at the same time. It is possible that the power consumption may increase on some configurations due to much faster execution time (because more work is done per cycle). However, the desired power / performance targets can always be achieved using DVS techniques.

7. CONCLUDING REMARKS

We introduced the notion of transient values and showed that 45% of all the results produced in a typical superscalar datapath are transient in nature. We also showed that the transient values can be predicted as such with a very high accuracy (more than 97% on the average across the simulated benchmarks). We presented SPARTAN - a microarchitectural technique that completely eliminates register allocations and avoids register writes of almost half of the produced values. The key results of our mechanisms are as follows:

- For the same register file size, SPARTAN improves the performance by 42.7% for 48 registers, 28% for 64 registers, 21.2% for 72 registers, 13.6% for 80 registers, and 7.8% for 96 registers over the baseline machine on the average across the simulated SPEC 2000 benchmarks.
- For a processor with 64-entry RFs, SPARTAN also reduces energy by 25%, when the energy dissipated by the additional logic needed by the scheme is taken into account.
- With only 64-entry RFs, SPARTAN matches the performance of a baseline machine with 96-entry RFs and results in 46% savings in the RF energy and power over that machine when all additional logic is taken into account.

Finally, we also showed that SPARTAN outperforms several recently proposed schemes for register file optimizations.

8. ACKNOWLEDGMENTS

We thank Oguz Ergin for his contributions during the early stages of this work. We also thank Ronny Ronen, Antonio Gonzalez, Pavel Vasek and the anonymous reviewers for their valuable comments on earlier versions of this paper. This research was supported in part by the National Science Foundation, award numbers CNS 0454298 and EIA 9911099, and by the Integrated Electronics Engineering Center at SUNY Binghamton..

9. REFERENCES

[1] Akkary, H., et. al.., “Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors”, *in the Proceedings of MICRO-36, 2003.*

[2] Azevedo, A., et.al., “Profile-based Dynamic Voltage Scheduling using Program Checkpoints in COPPER Framework”, *in the Proceedings of* DATE, 2002.

[3] Balakrishnan, S., Sohi, G., “Exploiting Value Locality in Physical Register Files”, *in the Proceedings of MICRO-36, 2003.*

[4] Balasubramonian, R.,et. al.., "Reducing the Complexity of the Register File in Dynamic Superscalar Processor", *in the Proceedings of MICRO-34, 2001.*

[5] Borch, E., Tune, E., Manne, S., Emer, J., "Loose Loops Sink Chips", *in the Proceedings of HPCA-8, 2002.*

[6] Bracy, A., et.al., “Exploiting Data-Flow Mini-Graphs in Superscalar Processors”, *in the Proceedings of MICRO-37, 2004.*

[7] Burger, D. and Austin, T. M., "The SimpleScalar tool set: Version 2.0", *Tech. Report, Dept. of CS, Univ. of Wisconsin-Madison, June 1997.*

[8] Butts J. A., G. Sohi "Use-Based Register Caching with Decoupled Indexing", *in the Proceedings of ISCA-31 2004*

[9] Butts, A., Sohi, G., "Characterizing and Predicting Value Degree of Use", *in the Proceedings of. MICRO-35, 2002.*

[10] Chrysos G., J.Emer, "Memory Dependence Prediction using Store Sets", *in the Proceedings of ISCA-25, 1998.*

[11] Cruz, J-L. et. al., "Multiple-Banked Register File Architecture", *in the Proceedings of ISCA-27, 2000.*

[12] Ergin O., et.al., "Increasing Processor Performance through Early Register Release", *in the Proceedings of ICCD, 2004*

[13] Ergin O., et.al., "Register Packing: Exploiting Narrow-Width Operands for Reducing Register File Pressure", *in the Proceedings of MICRO, 2004.*

[14] Franklin, M., et. al.., "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors", *in the Proceedings of MICRO-25.*

[15] Gonzalez, A., Gonzalez, J., Valero, M., "Virtual-Physical Registers", *in the Proceedings of HPCA-4, 1998.*

[16] Gonzalez, R. et. al. "A content Aware Register File Organization", *in the Proceedings of ISCA-31, 2004.*

[17] Gopal, S., Vijaykumar, T.N., Smith, J., Sohi, G., "Speculative Versioning Cache", *in the Proceedings of HPCA-4, 1998.*

[18] Hinton, G., et.al., "The Microarchitecture of the Pentium 4 Processor", *Intel Technology Journal, Q1, 2001.*

[19] Jourdan, S.,et. al. "A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification", *in the Proceedings of MICRO-31*

[20] Kessler, R.E., "The Alpha 21264 Microprocessor", *IEEE Micro, 19(2) (March 1999), pp. 24-36.*

[21] Kim I., M.Lipasti, "Understanding Scheduling Replay Schemes", *in the Proceedings of HPCA-10, 2004.*

[22] Kim, N., Mudge, T., "Reducing Register Ports Using Delayed Write-Back Queues and Operand Pre-Fetch", *in the Proceedings of ICS-17, 2003.*

[23] Kondo M. and Nakamura H. "A Small, Fast and Low-Power Register File by Bit-Partitioning", *in the Proceedings of .HPCA-11, 2005.*

[24] Lipasti, M., et.al., "Physical Register Inlining", *in the Proceedings of ISCA-31, 2004.*

[25] Lozano, G. and Gao, G., "Exploiting Short-Lived Variables in Superscalar Processors", *in the Proceedings of. MICRO-28, 1995.*

[26] Martin, M., Roth, A., Fischer, C., "Exploiting Dead Value Information", *in the Proceedings of MICRO-30, 1997.*

[27] Martinez, J., et. al.., "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors", *in the Proceedings of MICRO-35, 2002.*

[28] Monreal, et.al., "Delaying Register Allocation Through Virtual-Physical Registers", *in the Proceedings of MICRO-32, 1999.*

[29] Monreal, T., et.al., "Late Allocation and Early Release of Physical Registers", *IEEE Transactions on Computers, October 2004.*

[30] Monreal, T., Vinals, V., Gonzalez, A., Valero, M. "Hardware Schemes for Early Register Release", *in the Proceedings of ICPP-02, 2002.*

[31] Morancho E., et.al., "Recovery Mechanism for Latency Misprediction", *in the Proceedings of PACT-10, 2001.*

[32] Moudgill, M., et al. "Register Renaming and Dynamic Speculation: An Alternative Approach", *in the Proceedings of MICRO-26, 1993.*

[33] Park, I., Powell, M., Vijaykumar, T., "Reducing Register Ports for Higher Speed and Lower Energy", *in the Proceedings of MICRO-35, 2002.*

[34] Petric V., et. at. "RENO: A Rename-Based Instruction Optimizer", *in the Proceedings of ISCA-32, 2005.*

[35] Ponomarev, D., et. al."Reducing Datapath Energy Through the Isolation of Short-Lived Operands", *in the Proceedings of PACT-12, 2003.*

[36] Roth A. et. al "Register Integration: A Simple and Efficient Implementation of Squash Reus", *in the Proceedings of MICRO-33, 2000.*

[37] Sassone, P. et. atl., "Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication", *in the Proceedings of MICRO-37 , 2004*

[38] Sethumadhavan, S., et. al., "Scalable Hardware Memory Disambiguation for High ILP Processors", *in the Proceedings of MICRO-36, 2003.*

[39] Smith, J. and Pleszkun, A., "Implementation of Precise Interrupts in Pipelined Processors", in ISCA-12, 1985.

[40] Srinivasan S., et.al., "Continual Flow Pipelines", *in the Proceedings of ASPLOS 2004.*

[41] Tran, N., et.al., "Dynamically Reducing Pressure on the Physical Register File through Simple Register Sharing", *in the Proceedings of ISPASS-2004.*

[42] Tseng, J., Asanovic, K., "Banked Multiported Register Files for High Frequency Superscalar Microprocessors", *in the Proceedings of ISCA-30, 2002.*

[43] Wallase, S., Bagherzadeh, N., "A Scalable Register File Architecture for Dynamically Scheduled Processors", *in the Proceedings of PACT-5, 1996.*

[44] Yeager, K., "The MIPS R10000 Superscalar Microprocessor", *IEEE Micro, Vol. 16, No 2, April, 1996.*

[45] Yingmin L., et. al, "Performance,Energy, and Thermal Considerations for SMT and CMP Architectures", *in the Proceedings of HPCA-11, 2005.*

[46] Yoaz, A.,et.al. "Speculation Techniques for Improving Load-related Instruction Scheduling", *in the Proceedings of ISCA-26, 1999.*

[47] Kirman M. et. al., "Cherry-MP: Correctly Integrating Checkpointed Early Resource Cycling in Chip Multiprocessors", *in the Proceedings of MICRO-38, 2005.*

[48] Jones T. et al., "Compiler Directed Early Register Release", *in the Proceedings of PACT-14, 2005.*