

# Lazy Retirement: A Power Aware Register Management Mechanism

Guillermo (Eli) Savransky, Ronny Ronen, Antonio Gonzalez

Intel Corporation

{guillermo.savransky, ronny.ronen, antonio.gonzalez}@intel.com

## Abstract

*In this paper we describe "Lazy Retirement" - a power-aware improvement to the Intel's P6 family microarchitecture. Lazy Retirement significantly reduces the energy and power involved in register retirement. Lazy Retirement delays the copy from the physical register file (ROB) to the architectural (real) register file (RRF) until it has no choice and the physical register has to be re-used. In many cases, a new retired instruction invalidates such register before it is needed to be copied. Overall, Lazy Retirement eliminates most of the register copy operations involved in register retirement - saving energy while incurring no performance degradation. Alternatively, the reduction in register accesses can be used to decrease the number of register ports - thus allowing even faster, less complex, and more energy efficient register file - with a minimal performance loss.*

*Results: Lazy Retirement eliminates about 75% of retirement-related register copies in typical integer code. Lazy Retirement enables the reduction in the number of retirement-related register file ports from 3 to 2 with practically no performance penalty or even from 3 to 1 with very minor performance penalty.*

## Keywords

Out of Order Scheduling, Register Retirement, Energy Saving, Power-aware Microarchitecture.

## 1. Introduction

Power is becoming a first order consideration in designing new processors. We soon expect to see that the processor maximum cycle time, hence frequency, will be limited by thermal constraints. That is, while every individual circuit will be capable of running quite fast (given enough voltage is supplied), the processor itself will not be able to run at this high frequency because the generated power would be too high to be cooled in an affordable manner. This is especially true for processors intended for the mobile segment where limited cooling potential reduces the power budget, and for servers, to reduce the total energy consumption of server farms and maximize the MIPs per cubic meter.

To address this issue, microarchitects seek ways to reduce overall energy and power consumption. Attempts are being made to invent both new paradigms for power-aware microarchitectures as well as to make existing micro-architectures

more power-aware. This paper is an effort in the latter direction. The main idea in these attempts is to reduce overall energy with minor or no impact on performance. When a processor frequency is thermally limited, saved energy can be used to increase frequency and hence performance.

This trend is quickly developing - we have seen quite a few attempts being made during the last years in this area. The attempts can be classified into several families - shutting down inactive elements, caching of already done work, and smart reduction of some of the work.

Shutting down inactive elements attempts to identify units or part of structures that are not needed and reduce their energy consumption by either gating their clock or even shutting them off completely. The Pentium® 4 processor [Gunt01] use aggressive clock gating to inhibit the clock to units that are known not to be active at a certain cycle. Other studies have been conducted attempting to closely monitor the active size of various arrays - such as caches [Albo99] and instruction Queues [Buy00][Fol01] in order to deactivate portions of them that are not needed during a certain period. Smart algorithms are used to identify the inactive portion and the duration of inactivity. The energy saved is said to justify the minor performance penalty incurred.

Caching of already done work has been studied and practiced mainly for the front-end of Intel's IA32 processors. The trace cache [Hint01], the micro-operation cache [Sol01] and various flavors of basic block caches [Blac99, Jour00] decode micro-operations once and store them in an intermediate structure, to eliminate expensive re-fetching and re-decoding of these variable length, tough to decode instructions.

The last class aims at smartly reducing the work needed to accomplish a certain task. In many cases, past designs did not pay attention to power consumption and preferred simplicity over power saving. In particular, information was moved and re-written redundantly. An example for such saving potential followed the observation that many of the moved and operated upon data contain bytes with trivial values (e.g. 0 or 1). It was suggested to encode these bytes in few flags and move/operate only the non-trivial bytes and these flags [Cana00]. The technique presented in this paper belongs to this class of ideas.

In this paper we introduce a mechanism called *Lazy Retirement* that makes the register file of the Intel's P6 microarchi-

ecture family more power-aware by eliminating most of the register copy operations involved with register retirement. This mechanism can be implemented with no performance degradation. Alternatively, the reduction in number of register accesses can be used to reduce the number of register ports in order to achieve even faster, less complex, and more energy efficient register file with only minor performance degradation.

The remainder of the paper is organized as follows. Section 2 describes the Lazy Retirement mechanism – structures and algorithms. Section 3 attempts to explain why Lazy Retirement actually works. Section 4 describes the experimental methodology and brings the experimental results. Section 5 estimates the expected energy saving. Finally, we conclude in section 6.

## 2. Lazy Retirement

### 2.1 Background

Register renaming is a key component of dynamically scheduled processors in order to remove register name dependences among the instructions in-flight and hence increase the parallelism. Several renaming schemes are currently in use [Sima00]. These schemes differ in the place where speculative values are stored and the pipe stage when operands are read.

In this paper we focus on those schemes that have an architectural register file (also referred to as Real Register File or RRF) to that store committed values and a separate buffer to store non-committed values. This concept is used, among others, in the Intel P6 microarchitecture [Gwe95] and the PowerPC 620 [Lev95]. In both cases, the buffer for speculative register values is managed as a circular FIFO structure, which enables the physical register file to use a cyclic buffer semantics for allocation and deallocation of registers. As a drawback, such designs must copy the registers from the speculative to the architectural register file for every retired instruction that modifies a register

The main difference between the two schemes is that the buffer used by the P6 microarchitecture can be regarded as part of the reorder buffer (ROB) whereas the PowerPC 620 has a buffer that is smaller than the ROB and is referred to as rename buffers.

The mechanism presented in this work is evaluated for a P6-like microarchitecture, which has the buffer for speculative register values integrated into the ROB, but it works for any scheme that has a separate buffer for non-committed values that must be copied on retirement. Even the cyclic allocation/deallocation scheme is not a requirement of the proposed mechanism although all current implementations (ROB and rename buffers) use it.

In the rest of the paper the terms physical register file and reorder buffer (ROB) are used interchangeably.

### 2.2 The idea

The proposed *Lazy Retirement* mechanism allocates the physical registers cyclically – every new instruction that requires a register<sup>1</sup> gets the next sequential register. The new mechanism still uses a separate register file to store architectural values of registers, but it does not always copies committed values into the architectural register file. Instead of copying, when an instruction that writes to a register is committed, the physical register assigned to it becomes part of the architectural state.

Of course, it is possible that a physical register that contains an architectural value has to be allocated for a new instruction. Only in this case its value is copied to the separate architectural register file, which holds a place for every architectural register.

The next sub-section describes the Lazy Retirement mechanism in details. It presents the associated structures and algorithms as well as the handling of exceptions and branch mispredictions.

### 2.3 The details

**Hardware structures.** Figure 1 depicts the structures involved with the Lazy Retirement mechanism. Bold rectangles identify changes relative to the traditional P6 microarchitecture. The mechanism extends the use of the ROB to hold also some of the non-speculative register values. For every register in this structure, there is also an added valid bit that indicates whether the register contains valid architectural data. This bit is relevant only for free entries.

The architectural register file (RRF) is also used. It has a location for each architectural register, although, in the new mechanism, some entries may contain stale data.

The renamer has a speculative register map table that does not require any change with respect to the original map table of the P6 microarchitecture. For every architectural register, the table points to the last physical register mapped to it. This register can be either in the ROB or in the architectural register file. It is usually implemented as a field that points to an entry in the ROB, and an additional *is-in-RRF* bit.

To keep track of the place in which the last non-speculative copy of every architectural register is stored, an additional register map table is used. This table is similar to the one used by the renamer, but is modified when instructions commit, or on exceptions and branch mispredictions, as is

---

<sup>1</sup> In the P6 microarchitecture, every instruction is allocated a physical register.

described below. Like the register map table, each entry contains a pointer to the ROB and an *is-in-RRF* flag. We refer to this table as the *lazy map table*.

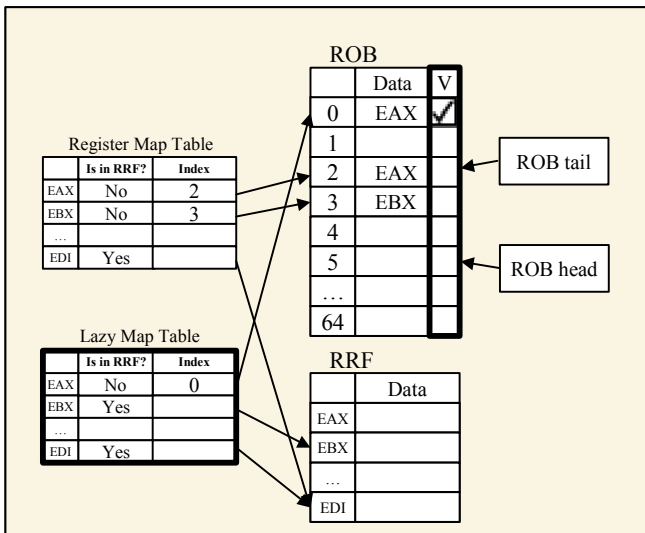


Figure 1: A general diagram of the proposed mechanism. In the shown example, there are three registers currently allocated to instructions, the second entry contains the last speculative value of the EAX register, while the entry number zero have the architectural value.

**Algorithm.** Figure 2 shows the steps for retiring an instruction. First, it is checked whether the instruction has a destination register. Some instructions, such as branches and stores, do not modify any register. In this case, the valid bit of the register is set to zero and nothing else is done.

Otherwise, the retirement logic checks whether the destination register is one of the registers for which Lazy Retirement is applied. An implementation may decide not to apply lazy retirement to registers that are infrequently modified (such as the IA32 segment and control registers), since in most cases they will end up being copied to the RRF anyway. Thus, these registers are always copied to the RRF at retirement.

If the register is candidate for Lazy Retirement, it now becomes the latest committed copy of the register, so its valid bit must be set to one. In addition, the previous latest committed copy must be invalidated if it is in the ROB. This is done by reading the pointer stored in the lazy map table entry corresponding to the destination register. If the entry points to an entry in the ROB, the corresponding valid bit is cleared.

Finally, this entry of the lazy map table is set to point to the retired ROB entry and the *is-in-RRF* bit set to zero.

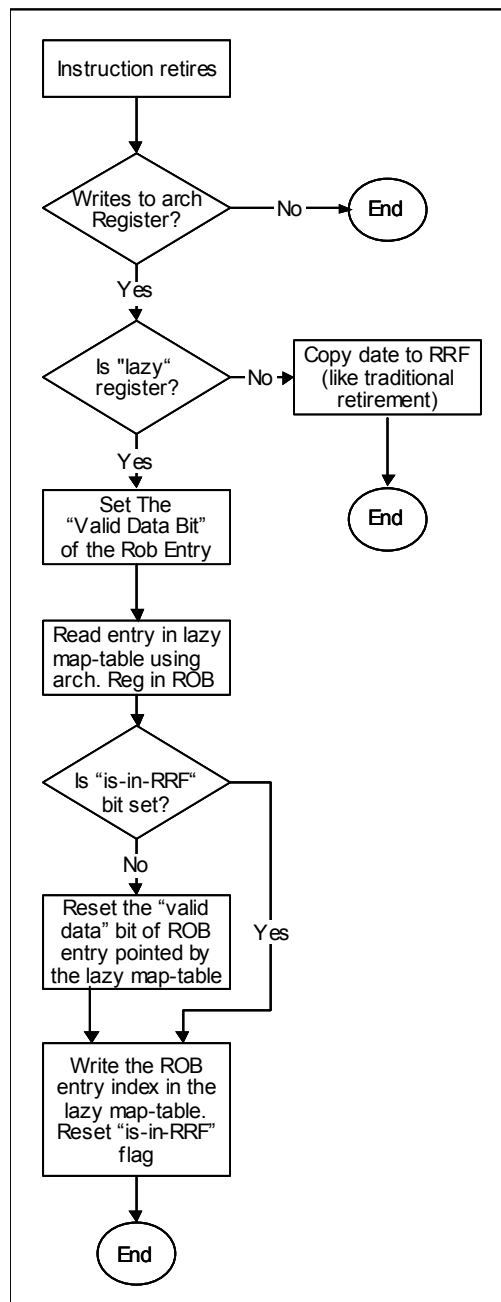


Figure 2: The steps for retiring an instruction.

Figure 3 shows the steps for allocating a physical register to an incoming instruction. First, the valid bit of the physical register is checked to see if it contains a valid architectural value. If it does, the value is copied to the RRF, and the valid bit is set to zero. In addition, the *is-in-RRF* bit in the corresponding entry of the lazy map table is set to one. If this was the latest mapping of the corresponding architectural register, the *is-in-RRF* flag of the register map table is also set.

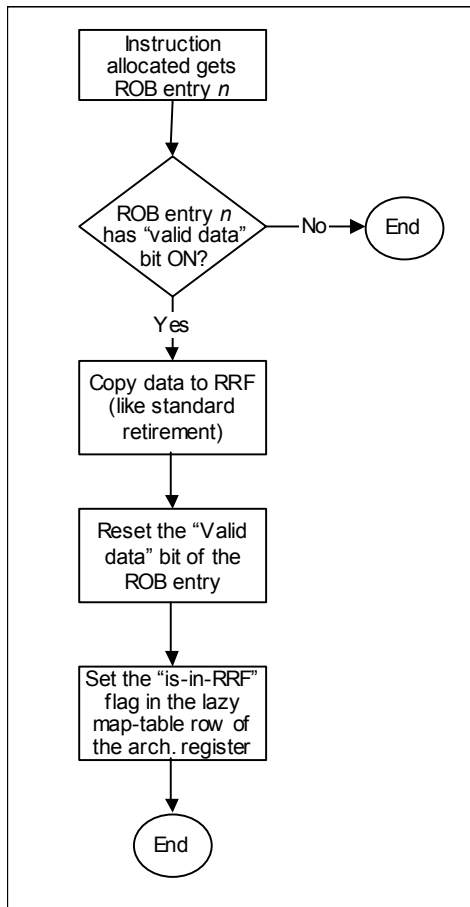


Figure 3: A flow diagram of the algorithm used when a physical register (in the ROB) is allocated to an incoming instruction.

Note that the valid bit in the ROB is used only to check at allocation time whether the entry has a valid architectural value. This could alternatively be done by checking if the lazy table has an entry whose contents matches the physical register ID, for instance using a CAM. This would avoid completely the need for the valid bits. However, CAM structures are very power hungry.

Another functionality that must be provided is the recovery in case of a branch misprediction or another exception that affects the control flow of the program. The P6 architecture waits until the faulting instruction retires. At this point, all that is needed is to set the *is-in-RRF* flag of all the entries in the register map table to indicate that the latest values of all registers are in the RRF. In the case of Lazy Retirement, some committed values may still be in the ROB. The exact location of every architectural register is stored in the lazy map table. Two alternative solutions are proposed. The first one consists of copying the content of the lazy map table to the register map table. The second solution consists of copying the architectural registers that are in the ROB to the RRF, by walking through the lazy map table and looking for

registers with the *is-in-RRF* flag on. The latter solution incurs in an extra energy penalty because of the copy of the values from the ROB to the RRF.

## 2.4 Related Research

Lazy Retirement relies on the observation that the lifetime of an architectural register, especially in applications compiled for the Intel IA32 ISA, is short. Several studies have been conducted in the past to exploit this phenomenon in order to either gain performance or to save energy and power.

Lozano and Gao [Loz95] have proposed a similar concept, which consists of an extension to the register allocation mechanism that allows not to assign short-lived variables to locations in the register file. Instead, short-lived variables are confined to locations in the reorder buffer using the features provided by the register renaming mechanism. While the concept is similar, the goal and the mechanism are quite different. The goal is to reduce register pressure by avoiding the allocation of architectural registers for short-lived variables. The mechanism is not transparent – it assumes architecture extensions and requires compiler support. The compiler identifies the short-lived variables for which architectural register need not be assigned.

Hu and Martonosi [Hu00] also take advantage of the facts that many register values are very short-lived and, based on that, have proposed a microarchitecture mechanism to save energy and power. They propose to buffer the result between the functional units and the register file in a "holding tank" named Value Aging Buffer (VAB). They show that most of the accesses to register values can be serviced from the VAB. The VAB saves energy since it is smaller than a typical register file and since it allows to reduce the number register file ports. A power-aware VAB implementation (as proposed in [Hu00]) has an inherent performance loss since it requires serial lookup – an element is first searched in the VAB, and if not found it has to be looked up in the full register file. Lazy Retirement avoids this slowdown by keeping track of the location of all values.

Postiff et al. [Post01] have proposed a mechanism for renaming schemes based on rename buffers that allow the data in a rename buffer to be accessible even after the buffer has been released and before it is re-allocated by a new instruction. That mechanism does not reduce the retirement activity and just focuses on reducing register file access time.

### 3. Why Does it Work?

As shown before, Lazy Retirement avoids the copying of a register when, by the time the allocator needs a ROB entry, the architectural register last assigned to that entry has already been overwritten.

Two conditions are necessary for this to happen:

1. The reorder buffer must not be completely full. In fact, the bigger the window of unallocated entries in the ROB, the higher the probability of avoiding a register copy.
2. Registers must be short-lived. This means that the architectural registers must get new values quite frequently. This is of course related to the program structure, the compiler implementation, and the characteristics of the involved ISA. The precise meaning of the word ‘frequently’ depends on the size of window of unallocated entries mentioned above, as is elaborated in the next paragraphs.

The graph in Figure 4 shows empirically that the first condition is satisfied.

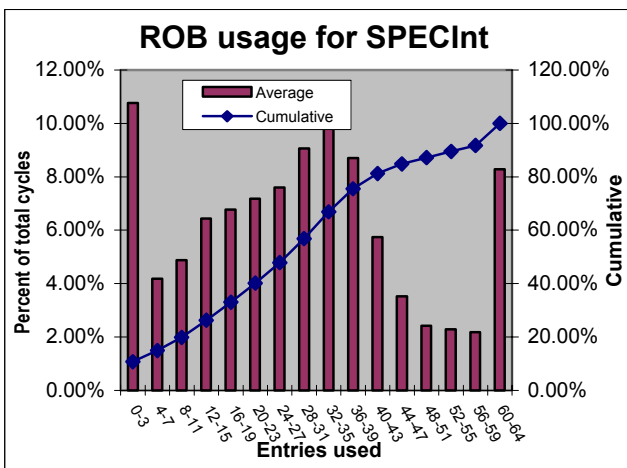


Figure 4: Reorder buffer usage. It can be seen that 80% of the time the ROB has more than 20 free entries.

The graph shows that the reorder buffer is completely full during only 8% of the cycles. Moreover, it can be seen that for more than 80% of the cycles, 20 or more ROB entries are unallocated. The low usage of the reorder buffer can be attributed to a relatively narrow front-end, branch mispredictions and their corresponding pipeline refill time, instruction cache misses, and the shortage of other resources – such as reservation station entries or load/store buffers.

As for the register lifetime, we will present three hypothetical scenarios and show that in every one of them the lifetimes of register values, in an integer code, are short enough to expect a significant copy savings.

In the first scenario, the average register lifetime is maximized. This corresponds to a code sequence in which architectural registers are overwritten in a round-robin fashion (i.e., between two consecutive writes to the same register, every other register is also written). In this case, if the empty part of the ROB is smaller than the number of registers in the ISA, all register values will need to be copied to the RRF. Otherwise, no copy will be needed<sup>2</sup>.

The second scenario assumes that the register reuse pattern is random with a uniform probability. Every new instruction uses a register as a destination that is chosen among all registers with the same probability. In this case, the probability that a copy is not needed when a new instruction is allocated is given by the following expression, where  $n$  is the number of architectural registers and  $k$  is the number of empty entries in the ROB:

$$P(n,k) = 1 - (n-1)^k/n^k$$

The graph on Figure 5 depicts the above function for different values of  $n$ . For instance, for 8 and 16 architectural registers, 9 and 19 free ROB entries, respectively, will suffice to avoid 70% of the copies.

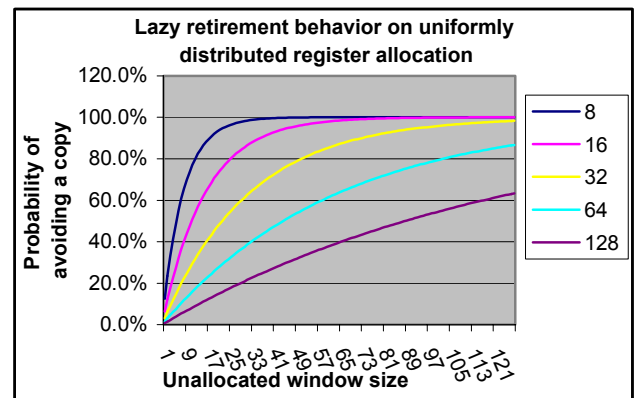


Figure 5: The probability of avoiding the copy of the next allocated register as a function of the number of free ROB entries and number of architectural registers

The third scenario is a loop with a short body that uses the same few registers for every iteration. As an example, consider the loop in Figure 6, which is part of the SPECInt2000 Perl benchmark. For a certain dynamic instruction stream of the program, this loop runs 1,643,558 (not neces-

<sup>2</sup> For simplicity of the explanation, we assume here that all instructions have a destination register. In practice, only about half of them do, so any reference in this section to the number of empty entries in the ROB should be interpreted as the number of empty entries previously used by an instruction with a destination register, which in average implies an increase by a factor of around 2.

sarily contiguous) iterations and represents 32% of the dynamic instructions.

```

0x420390:  addl  $-1,  %ebp
0x420393:  addl  $-1,  %ecx
0x420396:  movb  (%ebp), %dl
0x420399:  movb  %dl,  (%ecx)
0x42039b:  addl  $-1,  %edi
0x42039e:  jnz   .-14
    
```

Figure 6: Extract from the Perl trace. It is a six-instruction loop, which provides 32% of the dynamic instructions in the trace.

The loop uses only 4 registers: *ebp*, *ecx*, *edi* and *dl* (the lower 8 bits of *edx*). Figure 7 shows how the ROB looks like while running this loop. In this case, if there are more than six free entries in the ROB, no copy to the RRF is necessary at all. In general, if the number of instructions in a loop is smaller than the number of unused entries in the ROB, no copy operation is needed.

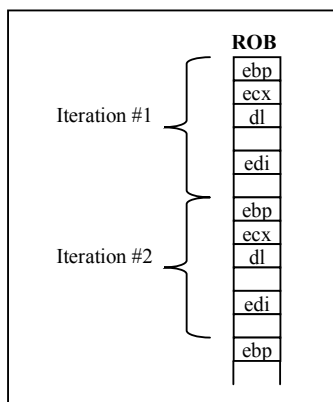


Figure 7: A picture of the reorder buffer when running the code of Figure 6. It can be seen that every iteration uses six ROB entries.

To summarize, we have shown several scenarios in which a relatively small number of free ROB entries allows Lazy Retirement to provide big savings in the copy activity. Combined with the experimental data that shows that the ROB normally has a significant number of empty entries, the potential gains of this mechanism is high. Next section quantifies these gains through detailed simulation.

## 4. Performance Evaluation

### 4.1 Experimental Framework

The design was modeled in an internally developed P6-like microarchitecture performance simulator. The simulator mimics an IA32 processor, in particular, it models:

- 8 architectural integer registers.
- 12 temporary integer registers, used for communication between micro-operations inside a single IA32 instruction.
- 64 ROB entries
- 24 RS entries.
- Capable of retiring 3 instructions per cycle.

The simulator processes instruction traces. Each trace consists of 30 million consecutive IA32 instructions. Traces include both user and kernel activities. Results are reported for 26 traces grouped into 4 suites:

- SpecInt: 12 traces from the SPECint2000 benchmark.
- WinSt99: 4 traces of from Winston99 benchmark.
- Smark98NT: 7 traces from SYSmark32 benchmark.
- MM99 & other: 3 traces.

### 4.2 Performance Results

The most important benefit of the Lazy Retirement mechanism is the reduction in the number of values that are moved from the reorder buffer to the architectural register file. Figure 8 shows the average number of copied values per retired instruction for both the standard and the Lazy Retirement mechanisms. We can observe that the lazy scheme generates a significantly lower number of copies for all the benchmarks. In particular, the standard scheme requires between 0.43 and 0.6 copies per instruction depending on the benchmark, and 0.5 copies on average. On the other hand, the lazy scheme generates between 0.07 and 0.19 copies, and 0.13 on average. Overall, the lazy scheme eliminates about 74% of the copies required by the standard scheme without any detriment in performance.

This important reduction in activity translates directly into energy savings since the dynamic energy consumed by the retirement stage is mostly determined by the number of values that are copied from the reorder buffer to the architectural register file. The lazy scheme requires an additional map table to identify the last committed value associated to each architectural register. This table is accessed in the rename stage and thus consumes additional energy. However, this table has very few (8 for general propose register and 12 for temps) and small (7 bits) entries.

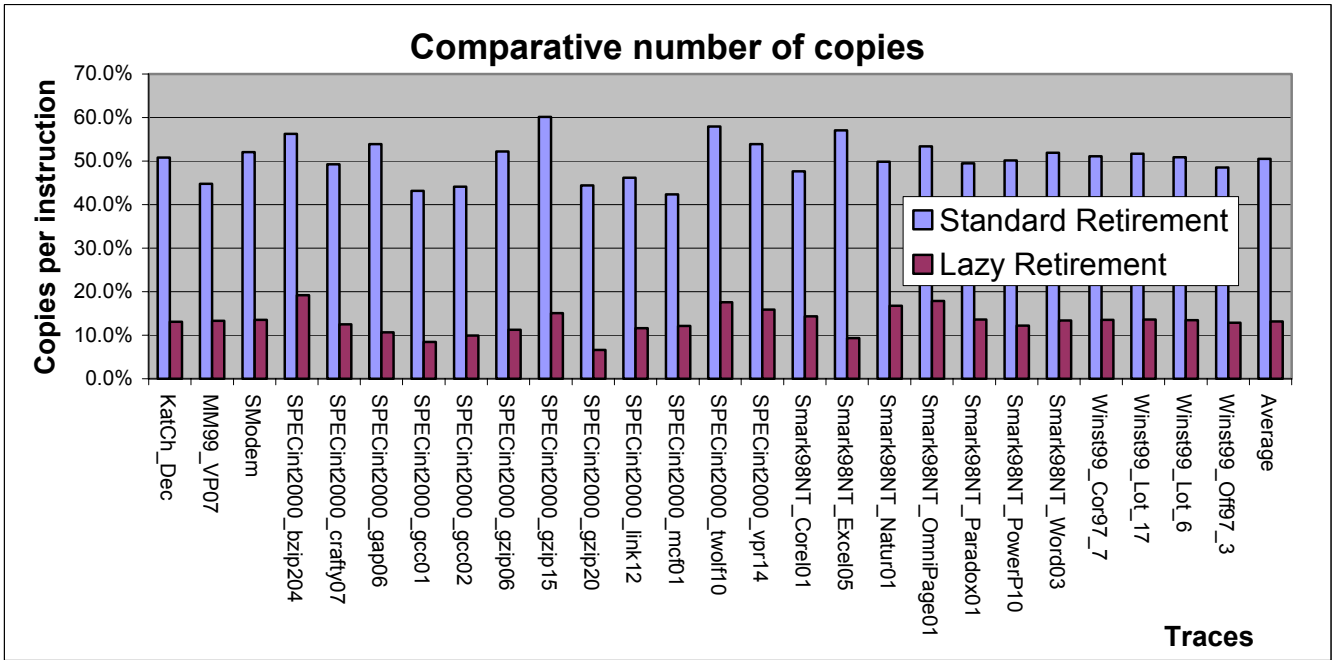


Figure 8: the average number of copied values per retired instruction for both the standard and the Lazy Retirement mechanisms.

The reduction in number of copies results also in a significant decrease in the pressure on the ports of both the reorder buffer and the architectural register file. Figure 9 shows the distribution function of the number of ports required by each scheme. We can observe that the lazy scheme uses more than 2 ports for less than about 0.3% of the cycles and more than one port just 2.5% of the cycles. This suggests that one could reduce the number of ports from 3 to 2 or even from 3 to 1 with minimal or no impact on performance. On the other hand, for the standard scheme, more than one port is required 12.5% of the cycles.

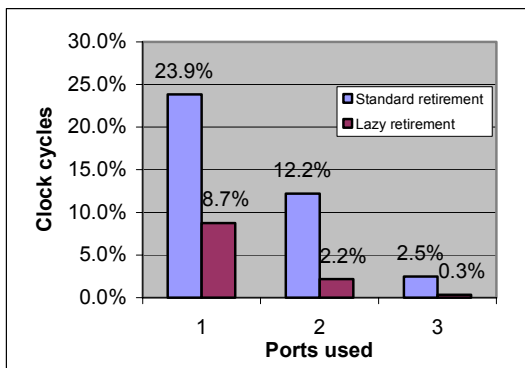


Figure 9: The distribution function of the number of ports

## 5. Energy Saving Estimates

The presented idea reduces drastically the number of copies from the ROB to the RRF. This comes at the price of a new array, which has multiple ports, and is read and updated many times. To understand the tradeoff and to prove that the idea actually reduces the consumed energy, we used a derivative of the CACTI [Joup99] tool modified to model register files. The energy consumed by the ROB, the RRF and the lazy map table were calculated for read and write operations. The numbers of accesses extracted from the performance simulator were used to get total energy, both for the traditional P6 mechanism and for Lazy Retirement.

The modeled ROB has 64 entries, 32 bits per entry, 6 read ports for sources, 3 read ports for retirement, and 3 write ports for write-back. The RRF has 20 entries (8 general propose and 12 temporary registers), 32 bits per entry, 6 read ports for sources and 3 write ports for retirement.

The lazy map table has 20 entries (one per architectural register), 7 bits per entry (6 to point to one of the 64 entries in the ROB and one for the "register-in-RRF" bit). It has 3 read and 3 write ports used when instructions are retired and 3 write ports used when a register is really copied.

Figure 10 shows the energy consumption of the retirement related activity using Lazy Retirement relative to the correspondent activity in the traditional P6 microarchitecture. It can be seen that on average, Lazy Retirement uses only 39% of the retirement energy consumed by the traditional P6 mechanism. From that, 21% is used by the ROB, 5% by the RRF and 13% is burnt by the new lazy map table.

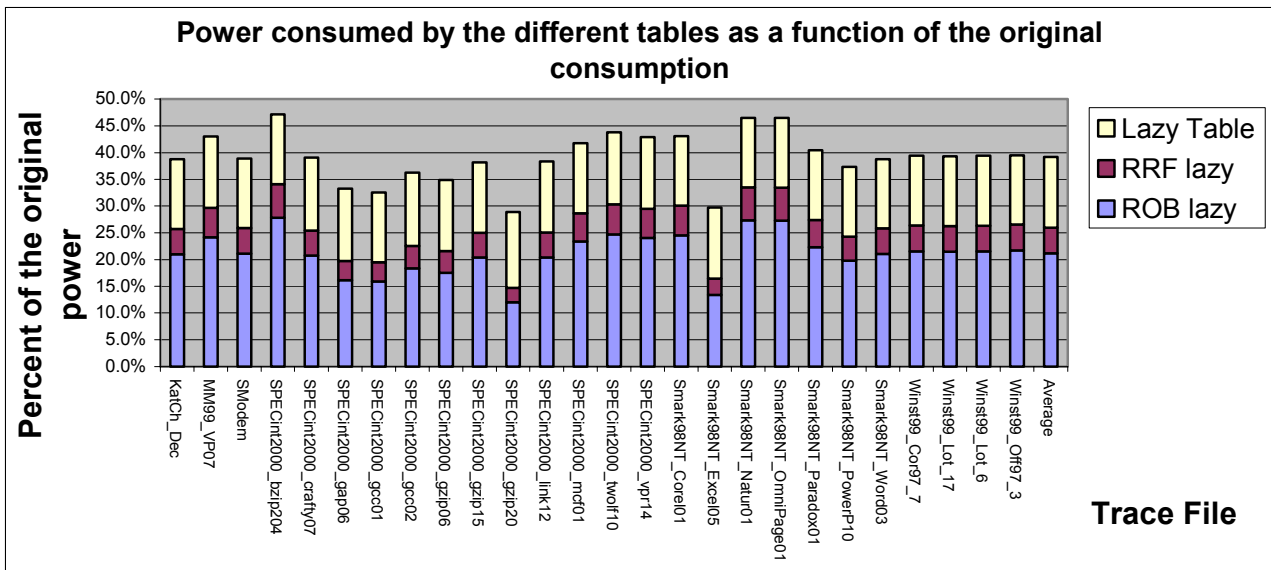


Figure 10: The estimated energy consumption of both the P6 micro-architecture and the presented lazy retirement mechanism

Another source for energy saving associated with Lazy Retirement is the reduction in the number of the register map table updates. In the P6 microarchitecture, every time a register is copied from the ROB to the RRF, the register map table is accessed and is possibly updated to reflect the change. The update is done only in the case when the retired register is the youngest copy of an architectural register. This is required to correctly rename new incoming instructions which consume retired registers. This mechanism is expensive in terms of power and complexity, since the physical register being copied is not necessarily the last incarnation of its architectural register. Lazy Retirement, by eliminating many of the ROB to RRF copies, reduces the number of these register map table updates and saves energy. The energy saving associated with the register map table is not quantified here, since it is expected to be somewhat smaller and is more difficult to estimate.

## 6. Conclusions and Future Work

In this paper we presented Lazy Retirement – a power-aware improvement to the existing P6 microarchitecture register retirement mechanism. We showed that Lazy Retirement can eliminate most (about 75%) of the register copies involved with traditional register retirement. We also showed that Lazy Retirement can allow the reduction of register file ports with only minimal performance impact, thus providing additional power saving potential. Our power analysis shows that we can expect about 60% power saving in the register retirement related power consumption.

The research of the Lazy Retirement mechanism can be enhanced by identifying more opportunities to eliminate register copy. Examples:

1. Do not copy a register to the RRF if the register is allocated to an incoming instruction that has no destination (e.g., a store, jump, or compare operations).
2. Delay the copy until the ROB entry is actually written. By that time a future instruction that writes the same register may have retired, thus saving the copy (similar to virtual renaming [Gonz98]).
3. Do not copy the ROB entry even when it is written, if the ROB entry update is eventually silent. That is, the new value is the same as the existing value.
4. In some situations, it may be beneficial to stall the allocation of registers to incoming instructions to further reduce register copies.

## 7. References

- [Albo99] D.H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation", MICRO 32, November 1999.
- [Blac99] B. Black, B. Rychlik, and J. Shen, "The Block-based Trace Cache," ISCA 26, May 1999.
- [Buy00] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D.H. Albonesi, "An Adaptive Issue Queue for Reduced Power at High Performance", Workshop on Power-Aware Computer Systems, (ASPLOS), November 2000.
- [Cana00] R. Canal, A. González, and J.E. Smith, "Very low power pipelines using significance compression", Micro-33, December 2000.
- [Fol01] D. Folegnani, A. González, "Energy-Effective Issue Logic", ISCA-28, July 2001.
- [Gonz98] A. González, M. Valero, "Virtual Physical Registers". HPCA-4, February 1998.
- [Gwe95] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design", Microprocessor Report Vol. 9, No. 2, February 1995.

- [Gunt01] S.H. Gunther, F. Binns, D.M. Carmean, J.C. Hall, "Managing the Impact of Increasing Microprocessor Power Consumption", Intel Technology Journal, February 2001, <http://developer.intel.com/technology/itj/q12001.htm>
- [Hint01] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, P. Roussel, "The Microarchitecture of the Pentium® 4 Processor", Intel Technology Journal, February 2001, <http://developer.intel.com/technology/itj/q12001.htm>
- [Hu00] Z. Hu, M. Martonosi, "Reducing Register File Power Consumption by Exploiting Value Lifetime Characteristics", WCED'00 (ISCA-27), June 2000
- [Joup99] G. Reinman, N. Jouppi, "An Integrated Cache Timing and Power Model", Compaq WRL, Summer 1999. <http://www.research.compaq.com/wrl/people/jouppi/CACTI.html>
- [Jour00] S. Jourdan, L. Rappoport, Y. Almog, M. Erez, A. Yoaz, R. Ronen, "eXtended Block Cache", HPCA 6, January 2000.
- [Lev95] D. Levitan, T. Thomas, P. Tu. "The PowerPC 620 Microprocessor: A High Performance Superscalar RISC Microprocessor", CompCon 95, 1995.
- [Loz95] L.A. Lozano, G.R. Gao, "Exploiting short-lived variables in superscalar processors", Micro 28, December 1995.
- [Post01] M. Postiff, D. Greene, S. Raasch, T. Mudge, "Integrating superscalar processor components to implement register caching", ICS'01, June 2001.
- [Sima00] D. Sima, "The Design Space of Register Renaming Techniques", IEEE MICRO, Vol. 20, No. 5, September/October 2000.
- [Solo01] B. Solomon, A. Mendelson, D. Orenstien, Y. Almog, R. Ronen, "Micro-operation cache: a power aware frontend for variable instruction length ISA", ISLPED'01, August 2001.