Register Packing:
Exploiting Narrow-Width Operands for Reducing Register File Pressure

Oguz Ergin[*]
Intel Barcelona Research Center
Intel Labs, UPC, Barcelona, Spain
oguzx.ergin@intel.com

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

Abstract

A large percentage of computed results have fewer significant bits compared to the full width of a register. We exploit this fact to pack multiple results into a single physical register to reduce the pressure on the register file in a superscalar processor. Two schemes for dynamically packing multiple "narrow-width" results into partitions within a single register are evaluated. The first scheme is conservative and allocates a full-width register for a computed result. If the computed result turns out to be narrow, the result is reallocated to partitions within a common register, freeing up the full-width register. The second scheme allocates register partitions based on a prediction of the width of the result and reallocates register partitions when the actual result width is higher than what was predicted. If the actual width is narrower than what was predicted, allocated partitions are freed up. A detailed evaluation of our schemes show that average IPC gains of up to 15% can be realized across the SPEC 2000 benchmarks on a somewhat register-constrained datapath.

1. Introduction

Modern superscalar processors use sizable physical register files to support large instruction windows for exploiting available code parallelism. A free physical register is allocated to hold a result of any new instruction with a destination register. This register 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 misprediction, an exception or an interrupt. Many recent superscalar processors, such as the Intel's Pentium 4 [9], MIPS 10000 [20] and Alpha 21264 [12] 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 register files if pipeline stalls due to the lack of physical registers are to be avoided.

The problem is exacerbated in processors that support large instruction windows: the access time of the large register file can force the use of a slower clock. Additionally, as higher issue widths are used to support large instruction windows, the number of read and write ports on the register file increases commensurately, again increasing the physical dimensions of the register file, slowing it down further. As a result, physical register files with a multi-cycle access time may become a necessity. Consequently, complex multi-stage bypass networks may be needed to avoid the performance degradation associated with the "holes" in the availability of the instruction source operands [5]. In addition, as the register file access stages are within a branch misprediction loop [3], the performance may degrade due to the increased branch misprediction latency. Finally, large register files also dissipate more power. The power dissipated in the register file can be anywhere between 10% and 25% of the total chip power [2], [23]. The situation is further exacerbated in the SMT processors, where the pressure on the register file is increased and larger physical register files are needed to support multiple thread contexts.

An alternative to building large register files is to use smaller number of registers, but manage them more effectively. Researchers have generally exploited the inefficiencies in register usage to reduce the number of registers by using late register allocation [7, 19, 33], early deallocation [14, 15, 16, 24] and register sharing [4, 11, 18]. In this paper, we propose alternative mechanisms for reducing the register file pressure. Our techniques are based on the observation that a large percentage of instructions produce narrow-width results. Such operands/results require

[*] This work was done when Oguz Ergin was at the State University of New York at Binghamton

fewer than the maximum number of bits available in a register for their storage. This situation can be exploited by packing multiple narrow-width results into the same register, thus reducing wastages in the register file and resulting in higher register file utilization. Such optimization can be especially attractive in the context of a 64-bit datapath, where 64-bit wide physical register files are used and unless the produced result is a double-precision floating point value (or, less frequently, a long integer value), significant register wastages occur.

The main contributions of this paper are as follows:

- We propose *register packing* – a set of microarchitectural techniques, both deterministic and predictive, to pack multiple narrow-width register values into a common physical register. Our technique results in 15% performance improvement on the average across Spec 2000 benchmarks, for a processor with somewhat register-constrained datapath configuration.
- We evaluate several register reassignment schemes and also analyze several mechanisms for handling possible width mispredictions and avoiding deadlocks.

The rest of the paper is organized as follows. We discuss the distribution of the produced result widths and also study the predictability of the result widths in Section 2. Section 3 describes the general considerations involved in packing of multiple results into a common physical register. Our techniques for reducing register pressure are described in Sections 4 and 5. Our simulation methodology is described in Section 6, followed by the simulation results in Section 7. We review the related work in Section 8 and offer our concluding remarks in Section 9.

2. Motivations

It has been well documented in the recent literature that many operand and result values in a datapath have narrow width [27, 31, 24]. In this section, we analyze the data width characteristics of the SPEC 2000 benchmarks that were used in this work. We also study the physical register file utilization based on the bit-occupancy of the individual registers and finally explore the predictability of the result widths using some additional bits in the I-cache.

First, we define a narrow-width value. The width of a value is the position of the first zero (or one) bit, such that all the bits in the more significant positions are also zeroes (or ones). The value with a width smaller than the full width of the datapath (32-bit or 64-bit typically) is then called a narrow-width value. One can obviously define several classes of narrow-width values – for example, those that can be defined using 8 bits, 16 bits, 24 bits etc (more on this in Section 3).

Figure 1 shows the width distribution of the generated register values, both committed and speculative. On the average, about 40% of all values can be represented using just 16 bits. Obviously, if a full-sized 64-bit register would be used to store each such result, significant inefficiencies in the register file usage would occur. Another 45% of all values can be represented using 32 bits. Only about 15% of the generated values require 64-bit storage within the register file to represent the result. Results shown in Figure 1 suggest that significantly better use of a register file is possible if narrow width results can be packed within a single 64-bit physical register. This observation has motivated this work, as well as the works of [27, 31, 24].

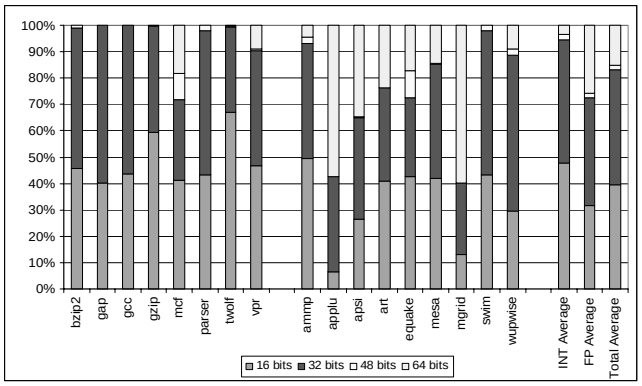

Figure 1 – Width Distribution

The second observation that motivated this work is that the widths of the produced results are highly predictable, as was also noted in [31]. For this study, for each instruction in the I-cache, we stored two additional bits signifying the width of the result produced by this instruction during its last dynamic instantiation. The 2 bits allow us to distinguish 4 possibilities: a) a result was less than 16 bit-wide, b) result was between 16 and 32 bits-wide, c) result was between 32 and 48 bits-wide and d) result was greater than 48 bits-wide. Figure 2 depicts the percentage of cases (captured by the two bits kept in the I-cache) where a dynamic instance of a static instruction generated a result within the same width class as the previous dynamic instance of the same static instruction. For each benchmark, 2 bars are presented. The left bar shows the percentage of cases where a dynamic instance of a static instruction generated a result exactly within the same width class as the previous instance of the same static instruction whereas the bar on the right shows the percentage of cases where a generated result is either within the exact same width class or has a larger width. This simple cache based last-width class prediction achieves an average prediction accuracy of 94% when width overpredictions are treated as mispredictions (left bar) and 98% when they are not (right bar).

High predictability of data widths was also shown in [31] where the data-width predictors similar to the load value predictor and the bimodal saturating counter branch predictor were evaluated. The prediction accuracies presented in [31] are in line with the results of Figure 2. The specific width predictor design is not the central part of

this paper, the important thing to notice is that the data widths are highly predictable and this predictability can be efficiently exploited using a variety of predictors.

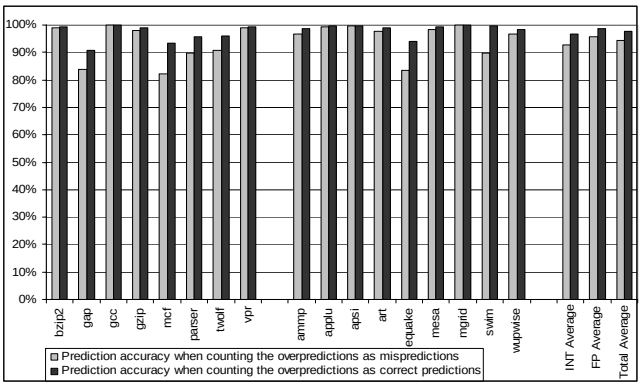

Figure 2 – Width Prediction Accuracy

In the next section, we formally define the width classes and describe the general issues in maintaining multiple results within a common physical register.

3. Packing Multiple Results into a Register

All results which are stored in physical registers are grouped into several "width" classes, say C_1, C_2, .. C_n, with associated bit widths w_1, w_2, .. w_n. A result belongs to the width class C_1 if the number of significant bits in the result is w_1 or lower; otherwise it belongs to the class C_k, if the number of significant bits of the result are within $w_{k-1} + 1$ and w_k. Multiple results can be stored within a single physical register of width D bits (D = 32 or 64 in contemporary designs) as long as the sum of the widths associated with the various classes of the stored results is less than or equal to D. In theory, a more efficient use of the physical registers is possible if a result class is associated with each possible value of the width from 1 through N (i.e., if n = N). However, such an organization grossly complicates the management of free "slots" within physical registers and also introduces delays in addressing a result stored within a register as a combination of the physical register address and specifiers for the size (or class) and the location of the result within the physical register.

3.1 Operand Access and Partition Management

The addressing of a narrow-width result, along with possibly other narrow results within a common physical register, as well as the management of the free regions within such registers, are considerably simplified if the number of classes are limited. Further simplifications and additional efficiencies result if the classes are integer multiples of a single byte or of a half word. To see this, consider a 64-bit physical register, i.e., D = 64 (We will use the example of a 64-bit register throughout this section to illustrate the basic concept behind our schemes; the discussion can be extended to other widths as well.). The result classes in this case of a 64-bit register are C_1, C_2, C_3 and C_4, which have the associated data widths of 16, 32, 48 and 64, respectively. This register can hold up to:

- 4 distinct results belonging to class C_1
- 2 distinct results belonging to class C_2
- 1 distinct result belonging to class C_3 or C_4
- 2 distinct results in class C_1 and another in class C_2
- 2 distinct results, one in class C_1 and in class C_3

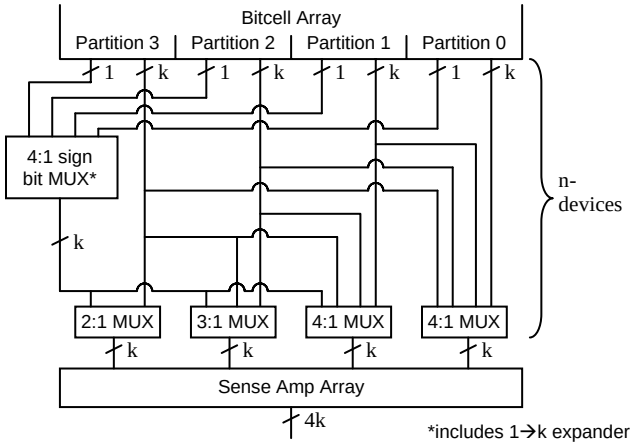

Figure 3 –Data Steering Logic

For this 64-bit register, one can also use a 4-bit mask to identify where a result is stored within a register. Bits in this mask correspond to the four consecutive 16-bit fields within the 64-bit register. The use of this mask bit to specify the locations of the results is another advantage – it is not necessary to use contiguous fields to hold parts of the same result within a physical register. For example, the fields used by a result belonging to class C_2 (i.e., any result with 17 to 32 significant bits) can be possibly indicated by the following bitmasks:

- 1100, 0110, 0011 (results stored within two contiguous 16-bit parts)
- 1010, 1001, 0101 (results stored within non-contiguous 16-bit parts)

The ability to store a result within the non-contiguous fields of a physical register improves the efficiency of using the space within the register file. The use of the four classes described above also simplifies the data steering circuitry needed to access a register for a result and also simplifies the sign extension logic that is needed to "expand" a narrow-width operand to the full width before commencing any operations on the data. A result is now addressed with a combination of two entities: a physical register address and a bit-mask (called *parts*) that identifies the part fields used for storing the result.

Figure 3 depicts an example of the data steering logic required for collecting results that can be stored in registers

that have 4 partitions each. Each partition is k-bits wide. The multiplexers in this logic are inserted on the bit lines leading into the sense amp array from the bitcell array implementing the registers. Such multiplexers can be typically implemented using n-transistors (in a CMOS implementation), as is routinely done in some RAMs that isolate the bitlines for fast sensing or in RAM designs that use column multiplexing. The logic of Figure 3 also includes a sign bit extension facility, as shown, that copies the sign bit into the most significant bit positions of narrow-width results. The multiplexers are controlled using the parts bit used to address a stored result (in conjunction with a register address). The multiplexers are thus turned on well before sensing is enabled. The propagation delay of the data steering logic (the delay of 2 n-transistor pass devices, at most) is almost absorbed by placing this logic ahead of the sense amp array. The inclusion of this logic grows the effective area of the register file slightly.

3.2 Managing Free Regions within a Register

We continue with the above example of a 64-bit datapath that has 64-bit physical registers. At any time, four free lists, FL_1 through FL_4, are used. FL_k (k = 1, 2, 3 or 4) lists all the physical registers that have exactly k 16-bit parts of free space, not necessarily contiguous. Entries in these free lists are thus specified by two entities: a physical register address and a bit-mask (*free_parts*) that indicates the locations of the free byte field(s). Note that *free_parts* mask for entries in FL_4 is unnecessary. Note also that a physical register is listed only within a single free list.

Storage Allocation: If, at any time, a free space of n parts is needed to store a narrow-width operand, the free list FL_n is searched. If an allocation is possible, that allocation is made for the storage of the result. If, on the other hand, FL_n is empty, the requested storage is looked for in FL_k, where k > n. If there are multiple candidates for FL_k, one can either use a rotating fit, a best fit or a worst-fit algorithm to determine the free list that would be used for the allocation (we used the best fit algorithm throughout the rest of this paper, where the first free list that is checked for performing a new register allocation is the one that corresponds to the size of the actually calculated result – say FL_k. On a failure to allocate a new register from this list, the free list FL_{k+1} is checked and so on. Of course, to insure that these activities can be performed within one cycle, all these free lists can be searched in parallel). After the allocation is performed, the *free_parts* field of the register used for the allocation is updated and an entry is set up in a possibly different list for the remaining portions of the register. For example, consider the scenario, where FL_1 is empty and the other free lists are non-empty. If register space for a single part is needed, let's assume that we use a best-fit algorithm and use a register from FL_2, say register p, with a *free_parts* mask of 0101. To perform this allocation, we remove this entry from FL_2 and allocate the last part field of p. A new entry is then made in FL_1 with the value (p, 0100).

Storage Deallocation: The process of deallocating the part fields within a physical register has analogous, but complementary steps. The challenge in this case is to quickly locate the entry for the register, if any, and move it to another free list after updating the *free_parts* mask bits. This process can be facilitated if the free lists are kept sorted by the physical register address. One can also imagine alternative implementations of the lists and the lookup, including the use of indirection entries or hash addressing – such discussions are not central to this paper. We believe that techniques exist for the fast lookup and management of these free lists, with the described allocation and deallocation semantics.

4. Conservative Packing

Our first approach towards dynamically packing multiple results into a single register is very conservative in nature and assigns a full-width register for the destination register of an instruction at the time of register renaming. Again, we use the example for the 64-bit physical registers.

4.1 Main Differences with the Conventional Design

Following the scheduling of the instruction producing a register value, its dependent instructions in the issue queue are awakened by the broadcast of the destination register id as wakeup tag. When a result has been computed for the instruction, the following steps are performed concurrently:

- The result is forwarded to dependent instructions in the usual manner.
- The number of significant bits in the result (and thus the class to which the result belongs) is determined.
- If the result requires all of the part fields in the allocated physical register, no further actions are necessary and the writeback to the register file proceeds normally.
- If the result requires fewer parts, an allocation is made using the free lists FL_3 through FL_1, if possible. If an allocation is not possible, the result is stored within the required part fields of the originally allocated physical register and the unused portion of that register is added to the appropriate allocation list. The new allocation made for the narrow-width result, be it from the originally allocated full-width register or from a different physical register (obtained from FL_3 through FL_1) is noted (i.e., saved in a latch). The result is also saved in a latch.

In the cycle following the writeback, if the result was reallocated to a different register or a portion of the originally allocated full-width register, the following steps are performed concurrently:

- A special tag broadcast is made to notify the still-waiting dependents of this instruction, if any, in the issue queue that instead of the original physical register address they were supposed to use to access this result, they have to use the newly-assigned address. The issue queue logic has to be modified to permit this update and such modifications are described later. At the end of this update, each issue queue entry would have the information of what parts of which physical register have to be read to supply the source operands for instruction execution.
- The result is written into the newly allocated region of the register file.
- The ROB entry of the instruction whose result was moved to the newly-allocated register is updated to reflect this update. This requires the addition of write ports to only the part of the ROB that has the destination information.
- The ROB entry of the instruction that will release the reallocated register has to be updated to reflect the new assignment (if the instruction has passed the rename stage).
- If the destination architectural register was not renamed, the rename table entry is also updated to reflect this new allocation. An interlocking logic is needed to ensure that instructions that are reading the rename table entry in the cycle they are updated get the most recent value of the entry.

When the rename table is fully checkpointed on branches, the update to the rename table from the writeback stage has to be propagated to all of the checkpointed copies that map to the originally allocated physical register. As noted in [24], such updates can be performed in the background. If the rename table update performed at the time of renaming the instruction's destination saves the old mapping in the ROB entry of the instruction, the update to the rename table entry from the writeback stage on a register reallocation can be performed in isolation, without any need to update any other copies. Of course, in this latter scheme, recovering from branch mispredictions requires walking back the ROB entries serially. In our simulations, we assumed that a full checkpoint of the rename table is created on branches.

The steps described above basically permit out-of-order execution to continue correctly when the destination physical register is re-allocated to suit the width of the result. Note also that this scheme does not suffer from any deadlocking because of the lack of a register for the width-based reallocation. Furthermore, the reallocation does not tie up any new full-width registers for the dispatch. The reallocation simply makes (further) use of an already-allocated full-width register. The full-width registers that are freed up as a result of the reallocation simply ease the register file pressure and improve the IPC.

Note also that all dependent instructions that were issued before the result was written back, pick up the result off of the bypass network using the originally allocated (full-width) physical register address; these instructions are functionally unaffected by the reallocations in progress. When the instruction reallocates a register, the original register that was allocated to this instruction at the time of renaming is added to the free list immediately after the re-broadcast of the new register tag across the issue queue. Thus, the subsequent reassignment of the freed register does not create any problems and any special considerations for allocating a register.

4.2 Datapath Changes

The obvious additions to the datapath include the free list management logic as well as the sign extension and byte multiplexing logic as described earlier. We also need to augment the issue queue (IQ) logic and the tag buses to permit dependent instructions in the IQ to be notified of the reallocation of the destination register as follows.

First, the source register fields of the IQ entries are widened to include the *parts* bit-mask (Section 3.2). If the r-bit physical register addresses are used and if we use the same 64-bit datapath example, this drives up the width of the source field entries from r to r+4. At the time of dispatch, the parts bits are initialized along with the register address in the source specifier fields.

Second, the tag bus is widened to support the broadcast of the new register specification (register address plus 4 *parts* bits), along with the address of the original register plus an additional line (*normal/update*) that indicates if it is a normal broadcast or a broadcast for updating source addresses in matching IQ entries on a reallocation. Thus, if we have a r-bit physical register address, the width of the tag bus line goes up from r to 2*r+5 (2*r is due to the fact that both the old address and the new address have to be re-broadcasted in the course of tag updates). This, as explained below, does not widen the comparators used within the IQ entries: the comparators still monitor only the original r-bit lines. For a normal broadcast used to wake up dependent instructions in the issue queue, the physical register address of the originally allocated full-width register is broadcasted on the originally present r-bit bus lines and the *normal/update* line is driven to a value corresponding to normal. The IQ comparators behave exactly as they do in the original design for this value on the normal/update line. For an update, the *normal/update* line is driven to a value corresponding to update, the physical register and the new register address specification are driven on the bus lines. IQ entries matching the original source address simply update the source address fields, overwriting the original physical register and parts bits. This requires the latches holding the source address within the IQ entries to be of a master-slave type.

Note also that the increase in the width of the wakeup bus can be limited by using a pair of existing tag buses to broadcast the old and the new addresses in the course of update broadcasts. However, as the update broadcasts (which actually require two sets of buses) are frequent in *Conservative Packing*, this can result in significant competition for the existing buses and lead to performance degradation. We evaluate some of these tradeoffs in the results section later in the paper.

4.3 Number of Tag Buses and Logic Details

Up to a maximum of two tag broadcasts are needed for every result produced that has a register as a destination: one for the wakeup and one for broadcasting the necessary updates on a reallocation. In theory, one thus needs to have two sets of tag buses. In reality, the tag buses are underutilized [42]. In a normal M-way superscalar machine that dispatches up to M instructions per cycle, up to M tag buses are needed to maintain the full throughput. However, not all of these tag buses are utilized because the IPCs are typically much lower than M and some instructions, such as stores and branches, do not have to broadcast their destination tags simply because these instructions have no destination. One can thus use the existing set of M tag buses to support the wakeup broadcast and the update broadcast without any performance penalty. In fact, as we show in the results section, increasing the number of tag buses is unnecessary, at least when width prediction is in use.

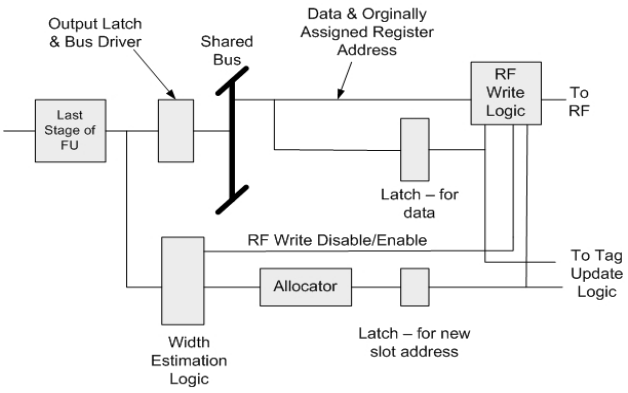

Figure 4 – Writeback Logic Details

Some additional explanations are also due on the concurrent steps of the writeback stage and the concurrent steps executed in the cycle after writeback. Figure 4 depicts the logic necessary to implement these steps. As the result and the originally-assigned slot address are driven over a shared bypass bus, the width estimation logic estimates the required slot width for the result. If the result requires only a portion of the register, this logic disables the register file write in the current (i.e.,the writeback) cycle. We estimate that the bus delays permits the write disable control to hold off the register file write before it actually commences. Simultaneously, the width estimation logic invokes the allocator to make a new allocation (register + parts) for the result. The core of the width estimation logic consists of four parallel arrays of NAND gates that combine the full-width result produced by the FU against four byte masks to determine the number of significant bytes needed to hold the result. As shown in Figure 4, the new slot address needed for a write into the register file as well as for the required broadcast for the tag updates in the next cycle, is saved in a latch. The sign padding adjustment needed to remove extraneous significant bits in the result for this write is part of the RF write control logic.

5. Speculative Packing with Width Prediction

As indicated in Section 2, the number of significant bits in the result produced by an instruction that targets a register is highly predictable. *Speculative Packing* exploits this fact to perform a register allocation based on the predicted width in advance and improve on *Conservative Packing* in a number of ways.

5.1 Main Differences with Conservative Packing

In *Speculative Packing*, width predictions are implemented through the addition of 2 bits for the I-cache entry of each instruction. If an instruction is decoded to be one that produces a result into a register, the two associated bits fetched from the I-cache along with the instruction indicates the predicted class of the result as one of C_1, C_2, C_3 and C_4 (for the 64-bit datapath used as an example). The addition of two bits to the fields for each of the instructions in a cache line may require modifications to the optimized design of the cache data array RAM macros, as writes to these bits can take place from the logic that detects the actual result width. An alternative and one that is more desirable, is to use a separate, independent array of prediction bits common to all of the cache ways. For an S-way instruction cache that has Q instructions per line, this implies that this array of predictions bits will have 2*S*Q bits in each row. The identity of the way providing the instruction line on an I-cache hit can be used to extract the relevant prediction bits from the row that was read out using the set index. The default prediction used for the width is that the result is predicted to be a full-width instruction. This prediction is revised when the result is actually computed. Of course, alternative prediction schemes, using prediction tables outside of the I-cache, can also be used, as in [31]. The complete evaluation of these techniques is beyond the scope of this paper.

One can also avoid using width predictors and instead rely on the explicit width specifications typically provided by the ISA. For example in the Alpha ISA, the instruction opcodes can be easily used to distinguish between 32-bit

and 64-bit instructions (i.e. addq vs. addl, ldq vs. ldl, etc.) However, such information can only distinguish two result classes. The use of the width predictor allows for the differentiation among several width classes at any level of granularity.

The main difference of *Speculative Packing* from *Conservative Packing* stems from the fact that a destination register is allocated based on the predicted length of the result at the time of renaming the instruction. The modifications to the instruction renaming/dispatch stage activities are as follows:

- Use the predicted result class to allocate a destination register
- Record the specifier consisting of a physical register address and a *parts* bit mask for this destination in the rename table and within the ROB entry for the instruction

Instruction dispatch stalls if an allocation is not possible (as in the base case design). The tag broadcast for the wakeup of dependent instructions in the IQ now uses a physical register address as well as the *parts* bits for the register allocated based on the prediction. This implies, following the notation used earlier that each tag bus has 2*r + 9 lines.

When a result has been calculated, the writeback stage implements the following steps in parallel:

- If the predicted width is higher than the width of the allocated result repository, an update broadcast, as in *Conservative Packing*, is performed to update the parts field of dependent instructions. Additionally, if the corresponding architectural register was not renamed, the rename table entry for the destinations is updated with the new value of the *parts* bits. Simultaneously, the unused portion is deallocated, as described in Section 3.2. A possible alternative to this step will be to do nothing and simply waste the unused byte fields. This, of course, requires the result to be sign extended to the allocated length before it is written into the register file.
- If the predicted width is smaller than that of the computed result, an allocation attempt is made to find a result repository of the correct width. If this allocation is successful, an update broadcast and a possible rename table update is performed as described in the last step. If the allocation fails, then we have the potential for a deadlock.

Deadlocking can occur in the scenario described above if the failure to allocate a repository for the result holds up instruction commitment and the release of free space in the register file. This situation is analogous to what is encountered in a scheme where register allocations are delayed till the result is generated [7, 19, 33]. However, the important difference is that in register packing, the situation leading to a potential deadlock can occur only on a rare occasion of a width misprediction when no appropriately sized physical register part is available in the free lists. Since the deadlocks occur very infrequently in our scheme, sophisticated deadlock management mechanisms are not required and simple techniques works very well as we detail later in the paper.

The logic necessary for handling the tag updates in *Speculative Packing* is very similar to the logic described in Section 4.3, Figure 4. The width estimation logic of Figure 4 is augmented to simply detect a width misprediction of a more general nature (The original logic of Figure 4 simply detected if the result was not full-width). As in the case of *Conservative Packing*, the FUs simply supply a full-width result. The writing of the significant parts into the assigned register is part of the RF write logic. Width mispredictions are detected by comparing the number of 16-bit slots in the original allocation against the number of 16-bit slots actually required by the generated result.

5.2 Avoiding Deadlocks

Several possible solutions exist for avoiding the possible deadlocks when an instruction whose result width was mispredicted cannot obtain an appropriately sized physical register part from the free lists.

Flush Younger Instructions (FYI): The simplest possible scheme for avoiding a deadlock as described earlier is to flush all instructions prior to the one for which a repository allocation on a width misprediction failed. Instruction execution resumes with the dispatch of the instruction for which the misprediction occurred, assuming that the result is a full-width one (for simplicity). One can, of course, remember the actual width and use that as the width predicted on restarting. This solution will not have a large performance penalty as long as the prediction rate is very high, which is indeed the case.

Steal From Younger (SFY): A potentially more efficient solution, but one that is clearly more complicated, is to locate the most recently dispatched instruction, say I, that was allocated a repository of the required width or a higher width and reassign all or part of this repository to the instruction whose width was under-predicted. All instructions following I are flushed from the pipeline and instruction dispatch resumes from instruction I. This, in some sense, is similar to “stealing from the younger” scheme described in [33] in the context of handling deadlocks with late register allocation.

Other solutions of varying complexity are possible, including a variation of the last one, where a minimum number of full-width registers are kept reserved for allocations on a width under-prediction, resorting to *SFY* only when the number of free reserved registers drops to zero. As shown in the results section, none of these complications are necessary and *SFY* does not provide any noticeable performance benefits compared to *FYI*, mainly because the actions involved in either solution are performed very infrequently, as in most cases the

appropriately sized register part is available on a width misprediction.

5.3 Main Benefits

Speculative Packing makes two expected improvements over *Conservative Packing*. These are as follows:

In *Conservative Packing*, an update broadcast is always needed when the result falls into the classes C_1 through C_3 (Another way of looking at *Conservative Packing* is to think of it as a variation of *Speculative Packing*, with a prediction that the result is in class C_4). In *Speculative Packing*, the update broadcast is mandated whenever a result's width is unpredicted (An update broadcast can be avoided in *Speculative Packing* on width-overpredictions). Thus, assuming the width distributions as given earlier, and the high likelihood of predicting a result's width, significantly fewer update broadcasts are required in *Speculative Packing*.

Conservative Packing locks up a full width register allocated to an instruction till at least the width of the result is known. *Speculative Packing*, on the other hand, allocates register portions based on the predicted width. Again, given a high width prediction ratio and the distributions of the data widths, *Speculative Packing* keeps more register portions available than *Conservative Packing*.

Parameter	Configuration
Machine width	4-wide fetch, 4-wide issue, 4 wide commit
Window size	64 entry issue queue, 64 entry load/store queue, 128–entry ROB
Function Units and Latency (total/issue)	4 Int Add (1/1), 1 Int Mult (3/1) / Div (20/19), 2 Load/Store (2/1), 2 FP Add (2), 1FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
L1 I–cache	32 KB, 2–way set–associative, 64 byte line, 1 cycles hit time
L1 D–cache	64 KB, 4–way set–associative, 64 byte line, 2 cycles hit time
L2 Cache unified	2 MB , 8–way set–associative, 128 byte line, 6 cycles hit time
BTB	4K entry, 2–way set–associative
Branch Predictor	Combined with 1K entry Gshare, 8 bit global history, 4K entry bimodal, 1K entry selector
Memory	256 bit wide, 80 cycles first part, 1 cycle interpart
TLB	32 entry (I) – 2-way set-associative, 128 entry (D) – 16-way set associative, 12 cycles miss latency

Table 1. Configuration of the Simulated Processor

6. Simulation Methodology

Our simulation environment was developed from scratch in C++, and includes a detailed cycle-accurate pipeline simulator (our code is rooted in the Simplescalar simulator [1], but has very little resemblance to it in the final version). To target programs, this environment exactly replicates Linux 2.6 on an Alpha 21264 at the system call and ISA level. All benchmarks were compiled with gcc 3.3.3 for the Alpha 21264 instruction set (compiler options: -O99 -funroll-loops -mcpu=ev67 -mtune=ev67 -mfix -mcix -mbwx -mfloat-ieee-fno-trapping-math). These options deliver the maximum possible optimization the compiler is capable of. The programs were then linked with the stock Linux glibc 2.3.3 for Alpha, compiled with the same options. For this study, we use 17 SPEC 2000 benchmarks; we had difficulty compiling the other benchmarks in our environment, mainly those that are written in Fortran 90 or C++. The results from the simulation of the first 1 billion instructions were discarded and the results from the execution of following 200 million instructions were used. Reference inputs were used for all simulated benchmarks. Table 1 shows the processor configuration used.

7. Experimental Results

7.1 Evaluation of Conservative Packing

We first evaluate the performance of *Conservative Packing*. Figure 5 shows the commit IPCs for 4 different situations. The leftmost bar shows the IPC of a 4-way baseline machine as defined in Section 6. The next bar depicts the performance of *Conservative Packing* where 8 tag buses are used. Since we simulated a 4-way machine, 8 tag buses are always sufficient to avoid any collisions between the regular tag broadcasts and the tag re-broadcasts performed during the re-assignment of registers. In this case, 4 of these tag buses are essentially reserved for tag rebroadcasts and have to be wider than the normal tag buses. Compared to the baseline case, the performance is increased by more than 14% on the average across the benchmarks, ranging from 40% (*art*) to 1% (*ammp*). Such a low performance increase in *ammp* is not surprising since *ammp*'s performance is predominantly constrained by a large number of D-cache misses, diminishing any gains due to the register file optimizations.

The next bar shows the performance of the system where both the tag re-broadcasts and the regular tag broadcasts share the same 4 tag buses. In this case, the priority is always given to the tag re-broadcasts and the selection of the instructions which are not able to obtain the access to the tag bus for the regular tag broadcast are delayed. One can see a fairly drastic performance degradation compared to the case where the number of tag buses is unrestricted. On the average, performance loss is 8% compared to the configuration with 8 tag buses. Compared to the baseline case, there is still 6% performance improvement on the average. It should also be noted that some of the benchmarks (*bzip2*, *gcc*, *parser*, *ammp*) exhibit worse performance than even the baseline case. This is obviously a consequence of the fact that delaying the execution of some instructions, even by one cycle, is very critical to the performance of these benchmarks.

The rightmost bar of Figure 5 shows the performance of *Conservative Packing* when all tag re-broadcasts are

handled within a single cycle and the rest of the pipeline is stalled during that cycle. Such an arrangement avoids complications associated with arbitrating for the tag buses between the re-broadcasts and the regular tag broadcasts. Unfortunately, this results in an average performance degradation of 27% compared to the base case, and is therefore not an attractive option. Such a large performance drop is expected, as the number of the tag re-broadcasts in *Conservative Packing* is very significant. The reduction of the number of the tag re-broadcasts is exactly what has motivated *Speculative Packing*, where totally different trade-offs occur, as we detail in the next subsection.

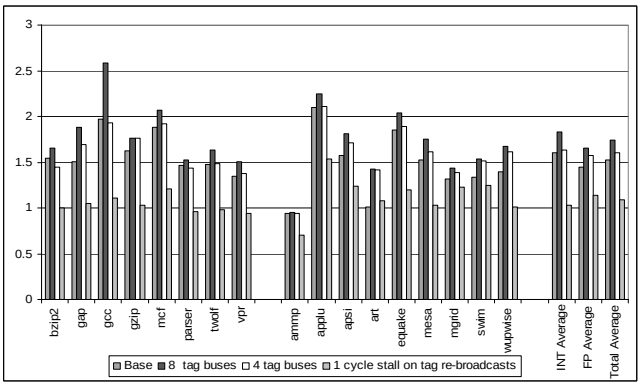

Figure 5 – Performance of Conservative Packing

7.2 Evaluation of Speculative Packing

Figure 6 shows the performance of *Speculative Packing* for the configurations detailed in the previous subsection (base case, 8 tag buses, 4 tag buses and 1 cycle stall on the tag re-broadcasts). In all cases, we assumed that on a width misprediction and the subsequent failure to allocate a required register partition (because of the absence of such in the free lists) all instructions following the mispredicted instruction (and including that instruction itself) are flushed and the fetching restarts from the mispredicted instruction using the actually computed result width as a prediction (which will always be correct). While it may seem that handling potential deadlocks in such a fashion could result in a significant performance loss, this is not the case because the prediction accuracy is high. Later in this subsection, we also evaluate alternative mechanisms for avoiding deadlocks.

For the case where 8 tag buses are used, the performance is increased by more than 16% on the average across the benchmarks, ranging from 46% (*art*) to 4% (*ammp*) compared to the baseline case. When 4 tag buses are shared among the tag re-broadcasts and the regular tag broadcasts, the average performance is still 15.5% higher than in the baseline case. One can see that in *Speculative Packing*, there is little difference between using 8 tag buses and sharing 4 tag buses. Again, this is a consequence of the high width prediction accuracy, representing a marked difference from what we observed in *Conservative Packing*. Even when the pipeline is stalled for 1 cycle during the tag re-broadcasts, the performance of *Speculative Packing* does not significantly degrade – it is still 10% higher than the performance of the base case on the average, although some benchmarks (*mcf*, *parser*, *ammp*, *equake*) exhibit a slight performance degradation. Unsurprisingly, these are the benchmarks (except *ammp*, which is dominated by D-cache misses) which have relatively lower width prediction accuracy (Figure 2).

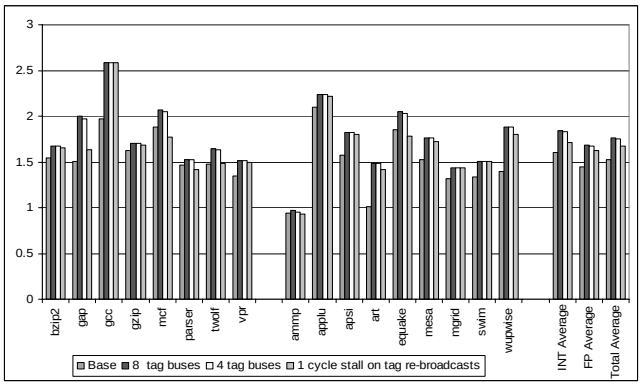

Figure 6 – Performance of Speculative Packing

Figure 7 presents a comparison of two deadlock avoidance schemes (*FYI* and *SFY*, Section 5.2). The first scheme flushes all the instructions following the mispredicted instruction while the second scheme tries to find a younger instruction to which an appropriately sized physical register part was allocated. That part is then "stolen" from that younger instruction and only the instructions that follow the younger instruction are replayed. As seen from the figure, there is virtually no performance difference between the two schemes. The average IPC difference is only 0.3% (with a maximum of 1.9% for *swim*). At first glance, this result seems to be counterintuitive. To understand why this is indeed the case, we need to closely examine the actions that transpire after a width misprediction and a subsequent failure to allocate a required register repository, and analyze the associated penalties.

These penalties come from two sources. First, a precise processor state (in particular, the state of the rename table) has to be reconstructed and, second, the instructions following the mispredicted instruction, or the instruction from which a register was stolen, have to be re-fetched and re-executed. We assume that the shadow copies of the rename table are created at every branch, thus to reconstruct a precise state of the rename table, all we need to do is to apply the modifications performed by the instructions between the mispredicted instruction (or the instruction from which the register was stolen) and the most recent preceding branch to the shadow copy of the rename table created for that branch. Obviously, the latency of this operation only depends on the distance between the

instruction in question and the prior branch. Whether the instruction in question is a mispredicted instruction itself or a younger instruction from which a register was stolen does not impact this latency. We actually verified in our simulations that this latency is about the same in both cases, as one would expect. Therefore, the only difference between the two schemes comes from potentially replaying a smaller number of instructions if stealing from the younger is used. However, as the number of times that such replays are needed is very small, there is almost no impact on the IPCs. For these reasons, the additional complexities involved in implementing *SFY* are not justified to support register packing.

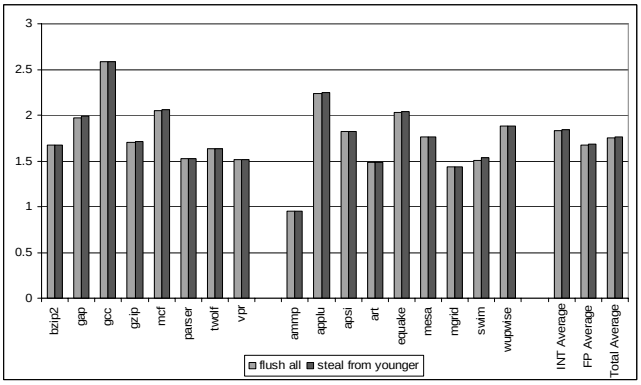

Figure 7 – Comparison of Deadlock Avoidance Techniques

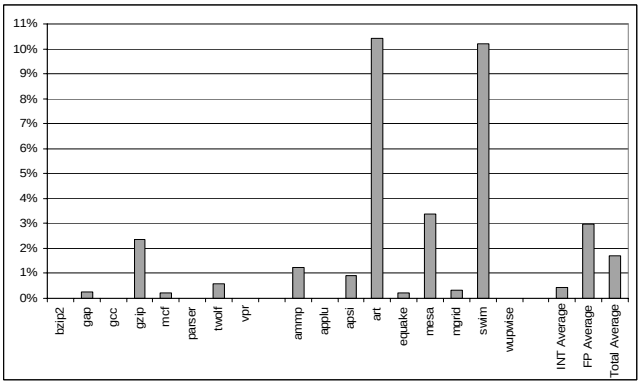

Figure 8 – Percentage of Width Mispredictions when Required Physical Register Part cannot be found in the Free Lists

Figure 8 shows the actual percentage of width mispredictions when required physical register part cannot be found in the free lists. This percentage is below 3.5% for all benchmarks with the exception of *art* and *swim*. Coupled with the high prediction accuracy, results of Figure 8 can be used to explain why the possibility of a deadlock is miniscule in our schemes and why there is almost no difference in the bars presented in Figure 7. The only two benchmarks that have a higher percentage of cases when a width misprediction could result in a deadlock are *art* and *swim*. For *swim*, the width prediction accuracy is relatively lower (about 90%) so there is some difference in the bars of Figure 7 for swim. On the contrary, the width prediction accuracy for *art* is more than 97%, so despite the relatively high percentage of cases where a register cannot be found in the free lists, the resulting impact of the deadlock avoidance mechanisms on the IPC is almost negligible.

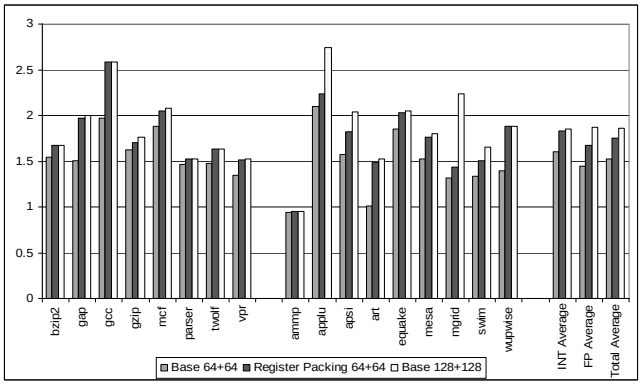

Figure 9 – Comparison of Register Packing and Doubling the Number of Physical Registers

Finally, Figure 9 presents the comparison between using register packing (we assumed *Speculative Packing* where 4 tag buses are shared) and simply doubling the number of physical registers in the base case. The leftmost bar shows the performance of the baseline case with 64 integer and 64 floating point registers. The middle bar shows the performance of register packing with 64 integer and 64 floating point registers when *Speculative Packing* is used. Finally the rightmost bar shows the performance of a baseline machine with 128 integer and 128 floating point registers. For integer benchmarks, the performance of register packing comes to within 0.8% of the performance of the baseline case with 128 integer and 128 floating point registers. For floating point benchmarks, the performance difference between register packing and the baseline case with 128 registers is about 10%. In fact, for FP benchmarks, register packing realizes slightly more than half of the speedup achieved by simply doubling the number of registers. This is primarily due to the benchmarks where a high percentage of generated values are double precision floating point values requiring full 64 bit registers to store them (*applu*, *apsi*, *mgrid*, refer to Figure 1 for details). On the average across all benchmarks, doubling the number of registers increases the IPC of the baseline case by about 22%, while the use of register packing increases the IPCs by 16%. In other words, register packing realizes 73% of the maximum performance improvement achievable by doubling the number of registers, with 94% for the integer benchmarks and 56% for the floating point benchmarks. The performance difference between the baseline case with 128 registers and register packing with 64 registers is about 6% on the average across all benchmarks, 0.8% for the integer benchmarks and 13% for the floating point benchmarks.

8. Related Work

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 [19, 7, 33]. Delayed physical register allocation was also used in [17] 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 [14, 15, 16, 24, 41, 45]. In [46], a combination of early deallocation and late allocation was used to completely avoid register allocation for a large number of instructions. The third set of solutions reduces the number of registers through the use of register sharing [4, 11, 18].

Replicated [12] and distributed [21, 22] register files in a clustered organization have been used to reduce the number of ports in each partition and also to reduce delays in the connections in-between a function unit group and its associated register file. 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, 3, 2, 26]. In [25], register file usage was optimized using compiler support to exploit dead value information. Asymmetrically ported register file was proposed in [43].

Techniques based on the value range analysis were extensively used in high-level code transformations [35, 36, 37]. In [28], the information about the operand sizes was used to rewrite the source code such that each data type has an associated width component. Several techniques were also proposed to directly exploit the fact that many operands and results in a datapath have narrow width. Most of these techniques propose optimizations for power efficiency [28, 29, 30]. In [28], a scheme for encoding significant zeros is exploited and investigated for power reduction in scalar pipelines. In [29], the presence of zero bytes was exploited for reducing the cache energy consumption. A software-controlled operand gating is proposed in [30], where the ISA is extended to include the opcodes that specify operand widths. In [39, 40], narrow width operands were exploited to reduce the power requirements of a value predictor.

Several researchers proposed compiler optimizations and architectural techniques to exploit the narrow-width operands for performance by packing multiple operations together to execute on the same FU. In [27], Brooks et.al. proposed a technique to detect the widths of the instructions to be executed and pack them so that they can be executed at the same time using the wide ALU. Similar mechanism was proposed in [31], but instead of detecting the data width deterministically before scheduling, the width is predicted. [32] proposed a similar approach for a VLIW-style machine. Compiler support to synthesize SIMD instructions from basic block statements was proposed in [34]. While these techniques exploit the narrow operands (and in some cases use the width prediction) for packing multiple operations on the ALU, we use the same motivations for packing multiple results within a common physical register. Our schemes can be very well used in conjunction with the techniques of [27] and [31].

In [24] Lipasti et al introduced a technique for reducing register file pressure that exploits significance compression [28]. In their technique, narrow width results are stored in the rename table entry itself. The work of [24] and this work offer two distinct solutions for exploiting narrow width operands for easing the register pressure.

A compiler based solution for packing multiple sub-word values into a single register in embedded processors was proposed in [44].

9. Concluding Remarks

We proposed microarchitectural techniques to pack multiple narrow-width results into a common physical register. Our first scheme, a conservative technique, allocates a full-sized physical register for every dispatched instruction and later reassigns a smaller-width register partition to this instruction, if the result turns out to be narrow-width. The drawback of this scheme is in the need to frequently re-broadcast the destination register tags when register reassignments are performed. Our second technique, a predictive scheme, avoids this complication by predicting the result width at the time of instruction renaming and allocating just the right register partition to hold the result. On a rare occasion of width mispredictions, the same tag buses used for the regular tag broadcast can be also employed to re-broadcast the new register tags to the dependent instructions still waiting in the instruction queue.

For a 4-way processor with 64 integer and 64 floating point registers, the predictive scheme with the tag bus sharing achieves 15% average IPC improvement across simulated Spec 2000 benchmarks. This gain is achieved with reasonable additional datapath complexities and without any increase in the number of tag buses.

10. Acknowledgements

We thank Matt Yourst for his help in developing the simulation environment. We would also like to thank Aneesh Aggarwal, Matt Yourst, Joseph Sharkey and the anonymous reviewers for their valuable comments on this paper. This work was supported in part by DARPA through contract number FC 306020020525 under the PAC-C program, the NSF through award No. EIA 9911099.

11. References

[1] Burger, D. and Austin, T. M., "The SimpleScalar tool set: Version 2.0", Tech. Report, Dept. of CS, Univ. of Wisconsin-Madison, June 1997 and documentation for all Simplescalar releases (through version 3.0).

[2] Balasubramonian, R., Dwarkadas, S., Albonesi, D., "Reducing the Complexity of the Register File in Dynamic Superscalar Processor", in *Proc. of MICRO-34*, 2001.

[3] Borch, E., Tune, E., Manne, S., Emer, J., "Loose Loops Sink Chips", in *Proc. of HPCA*, 2002.

[4] Balakrishnan, S., Sohi, G., "Exploiting Value Locality in Physical Register Files", in *Proc. of MICRO-36*, 2003.

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

[6] Franklin, M., Sohi, G., "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors", *in Proc. of MICRO-25*, 1992.

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

[8] Hu, Z. and Martonosi, M., "Reducing Register File Power Consumption by Exploiting Value Lifetime Characteristics", in *Workshop on Complexity-Effective Design (WCED)*, 2000.

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

[10] Jaleel A. and Jacob B. "In-line interrupt handling for software-managed TLBs." *in Proc. of ICCD-19*, 2001.

[11] Jourdan, S., Ronen, R., Bekerman, M., Shomar, B. and Yoaz, A., "A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification", *in Proc. of MICRO-31*, 1998.

[12] Kessler, R.E., "The Alpha 21264 Microprocessor", *in Micro*, 19(2), 1999.

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

[14] Martinez, J., Renau, J., Huang, M., Prvulovich, M., Torrellas, J., "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors", *in Proc. of MICRO-35*, 2002.

[15] Moudgill, M., Pingali, K., Vassiliadis, S., "Register Renaming and Dynamic Speculation: An Alternative Approach", *in Proc. of MICRO-26, 1993.*

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

[17] Park, I., Powell, M., Vijaykumar, T., "Reducing Register Ports for Higher Speed and Lower Energy", *in MICRO*, 2002.

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

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

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

[21] Canal, R., Parserisa, J.M., Gonzalez, A., "Dynamic Cluster Assignment Mechanisms", *in Proc. of HPCA-6*, 2000.

[22] Farkas, K., Chow, P., Jouppi, N., Vranesic, Z., "The Multicluster Architecture: Reducing Cycle Time Through Partitioning", *in Proc. of MICRO-30*, 1997.

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

[24] Lipasti, M., et.al., "Physical Register Inlining", *in Proc. of ISCA, 2004.*

[25] Martin, M., Roth, A., Fischer, C., "Exploiting Dead Value Information", *in Proc. Of MICRO-30, 1997.*

[26] Butts, A., Sohi, G., "Use-Based Register Caching with Decoupled Indexing", *in Proc. of ISCA, 2004.*

[27] Brooks, D. and Martonosi, M., "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance", *in Proc. of HPCA, 1999.*

[28] Canal R., Gonzales A., and Smith J., "Very Low Power Pipelines using Significance Compression", *in Proc. of MICRO-33, 2000.*

[29] Villa, L., Zhang, M. and Asanovic, K., "Dynamic Zero Compression for Cache Energy Reduction", *in MICO 2000.*

[30] Canal, R., Gonzalez, A., Smith, J., "Software-Controlled Operand Gating", in *Proc. of the Intl. Symp. On Code Generation and Optimization*, 2004.

[31] Loh, G., "Exploiting Data-Width Locality to Increase Superscalar Execution Bandwidth", *in MICRO-35*, 2002.

[32] Nakra, T., et.al., "Width Sensitive Scheduling for Resource Constrained VLIW Processors", *Workshop on Feedback Directed and Dynamic Optimizations, 2001.*

[33] Monreal, T., et.al., "Delaying Physical Register Allocation Through Virtual Physical Registers", *in Proc. of MICRO-34 1999.*

[34] Larsen, S., Amarasinghe, S., "Exploiting Superword Level Parallelism with Multimedia Instruction Sets", in *Proc. of the Conferenc on Programming Language Design and Implementation*, 2000.

[35] Budiu, M., et.al., "BitValue inference: Detecting and Exploiting Narrow Bitwidth Computations", *in Proceedings of EuroPar 2000.*

[36] Patterson, J., "Accurate Static Branch Prediction by Value Range Propagation", *in Proc. of PLDI, 1995.*

[37] Stephenson, M., et.al., "Bitwidth Analysis with Application to Silicon Compilation", in Proc. of PLDI, 2001.

[38] Cao, Y., "A System-Level Energy Minimization Approach Using Datapath Width Optimization", *in Proc. of ISLPED, 2001.*

[39] Sato, T., Arita, I., "Table Size Reduction for Data Value Predictors by Exploiting Narrow Width Values", *in Proc. of ICS, 2000.*

[40] Loh, G., "Width Prediction for Reducing Value Predictor Size and Power", *in the 1st Value Prediction Workshop (Held in conjuction with ISCA-30), 2003.*

[41] Ergin, O., Balkan,D., Ponomarev, D., Ghose, K., "Increasing Processor Performance Through Early Register Release", *in Proceedings of ICCD, 2004.*

[42] Aggarwal, A., Franklin, M., Ergin, O., "Defining Wakeup Width for Efficient Dynamic Scheduling", *in ICCD 2004.*

[43] Aggarwal A. and Franklin M., "Energy Efficient Asymmetrically Ported Register Files" *in Proc. of ICCD-21. 2003.*

[44] Tallam S. and Gupta R., "Bitwidth Aware Global Register Allocation", *In Proc. of* SIGPLAN-SIGACT, *2003.*

[45] Balkan D., Ergin O., Ponomarev D., Ghose K. "Selective Writeback: Improving Processor Performance and Energy Efficiency", *in Proc. of IBM p=ac2 conference, 2004.*

[46] Balkan D., Ponomarev D, Ghose K. "Predicting, Detecting and Exploiting Transient Values", *in the 2nd Value Prediction Workshop (held in conjunction with ASPLOS XI), 2004*