Improving Performance and Reducing Energy-Delay with Adaptive Resource Resizing for Out-of-Order Embedded Processors

Houman Homayoun[†], Sudeep Pasricha[†], Mohammad Makhzan[‡], Alex Veidenbaum[†]

[†]Center for Embedded Computer Systems
University of California, Irvine, CA
{hhomayou, sudeep, alexv}@ics.uci.edu

[‡]Department of Electrical and Computer Engineering
University of California, Irvine, CA
mmakhzan@uci.edu

Abstract

While Ultra Deep Submicron (UDSM) CMOS scaling gives embedded processor designers ample silicon budget to increase processor resources to improve performance, restrictions with the power budget and practically achievable operating clock frequencies act as limiting factors. In this paper we show how just increasing processor resource size is not effective in improving performance due to constraints on achievable operating clock frequency. In response we propose two adaptive resource resizing techniques *L2RS* and *L2ML1RS that adaptively resize resources by exploiting cache misses*. Our results show a significant performance improvement and overall energy-delay reduction of on average 9.2% (upto 34%) and 3.8% respectively across SPEC2K benchmarks for *L2ML1RS*. Applying *L2RS* resulted in 6.8% performance improvement (upto 24%) and 4.6% energy-delay reduction. We also present the required circuit modification to apply these techniques which shown to be minimal.

Categories and Subject Descriptors C.1.1 [Processor Architectures]: Single Data Stream Architectures; C.4 Performance of Systems

General Terms Performance, Design

Keywords Architecture, Performance, Energy-Delay, Out-of-Order Embedded Processor, Resource Resizing

1. Introduction

High performance embedded applications in the multimedia, networking, imaging and high-end consumer application domains are becoming increasingly prevalent in the market today. Since these applications have high performance requirements, there has been a gradual shift towards using more complex out-of-order superscalar embedded processors to meet performance goals. Examples of such embedded microprocessors include the IBM PowerPC 750FX [11] [22], NEC's VR5500 and VR77100 Star Sapphire [4] processors which have an on-chip L2 cache, along with a large instruction queue (IQ), reorder buffer (ROB) and register file (RF).

With CMOS technology scaling into the Ultra Deep Submicron (UDSM) region, hundreds of millions of transistors can already be integrated on a single System-on-chip (SoC). While this gives designers ample silicon budget to increase processor resources to improve performance, restrictions with the power budget and practically achievable operating clock frequencies act as limiting factors that prevent increases in processor resource sizes. As we will see in this work, while increasing the size of processor resources such as the reorder buffer (ROB), instruction queue (IQ), and register file (RF) can deliver a higher IPC (instructions per cycle), the negative impact such a resource increase has on achievable operating frequency can result in an overall performance degradation in terms of execution time. This is due to the fact that the access times to the muti-ported RF and IQ (in the bypass stage) are one of the most critical timing factors that determine the achievable processor operating frequency [1]-[3] [5]. We further study the possibility of adaptively resizing these units. Our study is based on the observation that after an L2 cache miss or multiple L1 cache misses one of the ROB/IQ/RF completely fills up and becomes the performance bottleneck. To reduce such fill up occurrences and hence improve performance we study the effect of increasing size of ROB/IQ and RF at the cost of increasing their access time. We study two possible approaches to overcome this problem – either lowering the processor operating clock frequency or pipelining access to these units. Our result show that none of these techniques resulted in a noticeable performance improvement (and in fact result in performance degradation for most of the studied cases) and reveals that the negative impact of applying pipelining or frequency scaling as a result of processor resource upsizing on the performance becomes dominant over the positive effect of resource upsizing on reducing the fill up rate and performance improvements.

We further study how these occupancy rates vary during a period where at least one L2 or at least two DL1 cache misses are pending (we refer to this as *cache miss period*) compared to when these cache miss scenarios do not occur (we refer to these periods as *normal periods*). Based on the results, we propose adaptive resource resizing and using aggressive resource upsizing only during a cache miss period. During the normal period, resources are kept at their typical size and as such they can operate at the normal operating frequency. During a cache miss period we increase the resource sizes which in turn results in an increase in their access times as well. To meet frequency targets, we use pipelining on these upsized resources.

Our adaptive resource scaling proposes two techniques – *L2RS* and *L2ML1RS*. In *L2RS* we start with resources at their typical size and

LCTES'08 June 12–13, 2008, Tucson, Arizona, USA.

will only increase their size when we encounter an L2 cache miss. In *L2ML1RS* we apply the scaling on a single L2 cache miss or when at least two DL1 cache misses are pending. In both techniques the resource size is returned to their normal size once all of cache misses are serviced and the scaled up part has no data. We also present the required circuit modification to realize these techniques. Our proposed circuit modification is applied to a unified register file and in contrast to costly banking or clustering techniques it comes with minimal hardware modification; adding one pass transistor per register file bitline. The results show a significant performance improvement and an overall energy-delay reduction of on average 9.2% and 3.8% respectively across SPEC2K benchmarks for *L2ML1RS*. Applying *L2RS* resulted in 6.8% performance improvement and 4.6% energy-delay reduction.

The rest of the paper is organized as follows: related work is described in Section 2. In Section 3 we presented the motivation for proposed architectural techniques. Section 4 describes our proposed architectural technique and the required circuit modification. Experimental results are presented in Section 5. Finally, in Section 6 we offer concluding remarks.

2. Related Work

There has been a lot of research that proposed altering the structure of the register file (RF), reorder buffer (ROB) and instruction queue (IQ) for improving performance. One set of techniques uses localities of communication to split the microarchitecture into distributed clusters, each containing a subset of the RF, ROB and IQ [1] [6]-[9] [18]. The different functional units in a cluster can service different requests in parallel as long as inter-operation value communication is within a cluster; a penalty occurs when a value is passed from one cluster to another. A critical issue in the design of such systems is the heuristics used to map instructions to clusters. These schemes have the potential to scale to larger issue widths but require complex inter-cluster control logic to map instructions to clusters and to handle inter-cluster dependencies.

Alternatively, other approaches retain a centralized microarchitecture, but partition the processor units such as the RF [3] [10] and IQ [14] [17] to reduce access time and energy dissipation. Partitioning the register file into multiple banks for instance reduces the ports on the partitions, which reduces the pre-charging and sensing times and the related energy dissipation. But these reductions come at the cost of value multiplexing and port conflict problems. The major drawback of all these banking techniques in general is in the complexity that speculation adds and more specifically the complexity they introduce on handling the coherency in register caches and banking conflicts. This added complexity becomes even more critical for embedded processors that work in resource-constrained environments.

There have also been some techniques to dynamically resize the ROB [12] [16], IQ [14]-[16] and RF [13] at runtime to reduce energy dissipation. None of the above schemes exploit the L1 and L2 cache misses for resource adaptation. The technique that comes closest to our work is [5], which performs early register de-allocation based on L2 misses to improve performance. This is accomplished by speculatively committing the load-independent instructions and deallocating the registers corresponding to the previous mappings of their destinations, without waiting for the cache miss request to be serviced. The early deallocated registers are then made immediately available for allocation to other instructions, thus improving the overall processor throughput. However, such a scheme increases complexity since it requires additional resources such as bit vectors to identify the sources and the destinations of the load-dependent instructions, additional bits in the ROB and possibly a backup register file to store de-allocated values. In contrast, we propose a much simpler circuit level approach that dynamically adapts the ROB, RF and IQ on cache misses to achieve significant performance improvements.

3. Motivation

A load instruction miss in the cache (DL1 or L2) prevents any dependent instructions from being issued in a processor. The dependent instructions fill up the reorder buffer, the instruction queue, register file, and/or the load and store queues (*LQ/SQ*) until the miss returns. In this section we briefly study the status of the *ROB*, *IQ* and *RF* during such a scenario for our out-of-order 2-way embedded processor similar to the IBM PowerPC 750FX architecture [11] with a separate IL1 (level 1 instruction cache) and DL1 (level 1 data cache) of 32KB with access time of 2 cycles and a unified L2 (level 2 cache) of size 256KB with an access time of 12 cycles. The miss penalty of accessing the main memory is 60 cycles. The size of ROB, IQ and RF is 24, 12 and 32 respectively.

At every cycle, up to two new instructions are dispatched to the *ROB* and up to two physical registers are allocated out of the pool of free registers. To allocate new instructions, the processor also releases up to two committed-instruction physical registers and their *ROB* entries at every cycle. This is done in program order to enable precise interrupt handling. When a cache miss occurs, the load instruction that caused the miss stays on top of the *ROB* and doesn't allow the subsequent instructions to be committed. As a result, the subsequent instructions occupying the *ROB* and RF cannot be released until the miss returns. This gradually increases *ROB* and *RF* occupancy, and reduces the processor issue rate. The same scenario occurs for the LQ/SQ and IQ – the subsequent dependent instructions to the load with a miss cannot be issued due to data dependency. Such instructions reside in the IQ until the miss returns. Accordingly the IQ occupancy increases as the processor fills it with up to two instructions per cycle. Due to data dependencies however, very few out of these can be issued.

Given the long cache miss service time (12 cycles for DL1 and 60 cycles for L2 in our architecture), the above scenario can happen quite frequently. For the case of an L2 miss, due to the long service time, either the *ROB*, RF, LQ/SQ or *IQ* can completely fill up with subsequent instructions and the processor ends up being stalled (issue rate of 0) until the miss is serviced. We refer to this as *scenario I*. In the case of a DL1 miss, the service time is much smaller than that for an L2 miss, and it is less possible that any of *LQ/SQ*, *ROB* and *IQ* (all referred to as buffer) fills up completely before the cache miss is serviced. Note that when one DL1 cache miss occurs, its dependent instructions cannot be issued and all the subsequent instructions cannot be committed as discussed above. This reduces the issue rate and increases the occupancy of the aforementioned buffers. In the presence of many pending DL1 cache misses, the impact on issue rate would be large and the occupancy of queues will increase significantly. We refer to this case where multiple DL1 misses are pending as *scenario II*. We refer to the period during which one or more L2 miss/misses and/or at least two DL1 misses are pending as *cache miss period*. We refer to the rest of program execution time as *normal period*. Figure 1 illustrates the status of the ROB occupancy in our architecture for the scenarios described above, during a cache miss period and for a normal period (non-scenario I occurs when there is no pending L2 cache miss and non-scenario II occurs when there is none or

one pending DL1 cache miss exist in the pipeline). As can be seen from the figure, the ROB occupancy during *scenario I* increases significantly compared to its occupancy during normal period for all integer (INT) SPEC2K benchmarks. The maximum increase occurs for the *gcc* benchmark, for which the ROB occupancy doubles (increases by more than 100%). Across all floating points (FP) benchmarks we observed a smaller increase. The average increase across all benchmarks is around 20%. For *scenario II* the ROB occupancy increases but less than for *scenario I*. The average ROB occupancy increase is 14%. This difference is due to the smaller latency penalty of DL1 cache miss compared to an L2 cache miss.

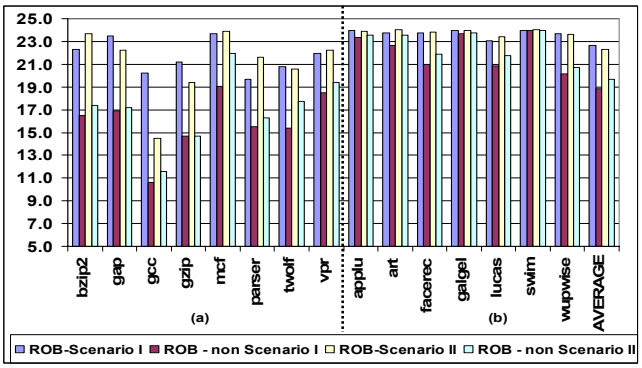

Figure 1. Case study: ROB occupancy variation during scenario I and scenario II compared to normal period. (a) INT benchmark (b) FP benchmark

In figure 2 we report the frequency of ROB saturation (i.e., how often it becomes completely full) during *scenario I* and *scenario II*. It should be noted that the ROB can also potentially fill up completely due to TLB misses, but for this work we only consider cache misses. The results from the figure show that the ROB is filled completely for a significant portion of a program execution time, 45% of the time on average. Interestingly, in FP benchmarks, this rate is larger than for INT benchmarks, due to the fact that FP benchmarks have higher L2 and DL1 miss rates compared to INT benchmarks. Another explanation for the ROB filling up significantly more often in FP benchmarks compare to INT benchmarks is that FP instruction operation latency is higher than the INT operation latency.

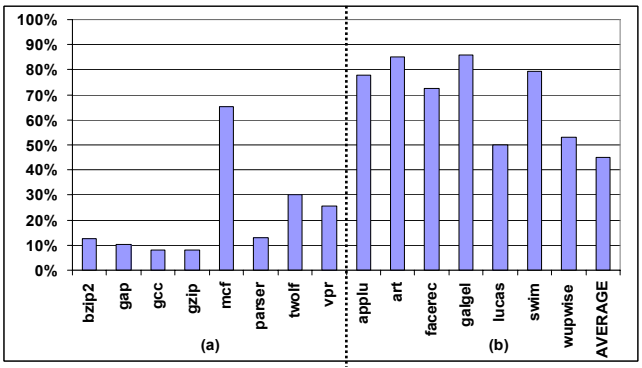

Figure 2. Frequency of ROB saturation due to L2 cache miss or at least two DL1 cache misses. (a) INT benchmark (b) FP benchmark

3.1 Impact of Increasing Resource Sizes

Intuitively, from the results presented above, it can be inferred that increasing the size of the ROB, as well as the IQ and RF will prevent them from filling up completely as frequently and potentially improve performance. It should be noted that the sizes of these buffers have to be scaled up together; otherwise the non-scaled ones would become a performance bottleneck.

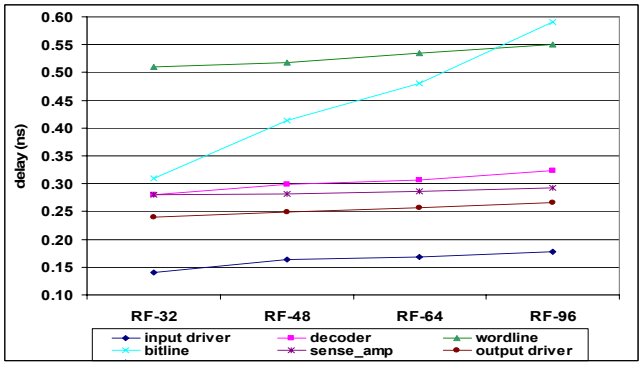

Figure 3. Break down of RF component delay with increasing RF size

Typically, the size of the RF is a limiting timing factor that determines the maximum operating frequency of a processor as discussed in several other works [1] [2] [3] [5]. Increasing the size of the RF increases its access time. This is mostly due to increase in the length of the bitline. Figure 3 shows the delay breakdown among the various components of the RF, for 32, 48, 64 and 96 entry RF configurations. Results are shown for a single read operation, with delays calculated using a modified version of CACTI4 [19]. A clear trend seen in this figure is the significant increase in bitline delay when the size of register file increases. This can be explained as follows: The signal propagation delay of bitline is relative to its equivalent capacitance. The equivalent capacitance on the bitline is $C_{eq} = N * diffusion\ capacitance\ of\ pass\ transistors + wire\ capacitance$ (usually 10% of total diffusion capacitance) where N is the total number of rows. As the number of rows increases the equivalent bitline capacitance also increases and since the propagation delay on the bitline is relative to RC_{eq}, the propagation delay approximately increases with the number of rows. The propagation delay for the remaining RF components increases only slightly with an increase in RF size, as shown in the figure.

It is thus clear that increasing RF size increases its access time. A similar increase in access time occurs for the ROB and IQ. As a result, the achievable operating frequency of the processor is reduced when resource sizes are increased. To reduce access time it is possible to apply banking or clustering techniques to improve RF access time, as has been proposed for high performance processors [10] [3] . However banking or clustering the RF is a costly solution, due to the significant complexity that is introduce in handling banking conflicts and coherency in RF banks. Such complexity can be prohibitive, especially in the resource constrained environments in which embedded processors operate.

The increase in access time for upsized resources leads to two main implementation choices for designers: *(i)* reducing operating clock frequency, and *(ii)* maintaining operating clock frequency by pipelining resources. In the next two sections, we explore these two techniques in more detail, to analyze their impact on performance.

3.2 Impact of Reduced Operating Clock Frequency

Table 1 presents three different processor configurations as a case study to see the effectiveness of increasing the size of resources (RF, ROB and IQ) to reduce the occurrences of stalls and hence potentially improve processor performance.

Table 1. Reduction in operating clock frequency with resource upsizing

Processor Configuration	Conf. 1 baseline	Conf. 2 intermediate	Conf. 3 aggressive	Conf. 4 upper bound
RF size	32	48	64	96
ROB size	24	32	48	64
IQ size	12	24	32	48
RF access time (ns)	1.76	1.92	2.03	2.19
Operating Freq (MHz)	560	520	490	450

As explained earlier and consistent with several previous works, we assume that RF access time determines the achievable operating frequency of the processor. As we scaled up all the resources, we assume that the RF access still decides the achievable operating frequency. The baseline configuration is shown in the *Conf. 1* column, with a 560 MHz operating frequency. The *Conf. 2* column represents an intermediate configuration, with the RF, ROB and IQ upsized to 48, 32 and 24 entries, respectively. Using a modified version of CACTI4 [19], the access time for RF is found to increase to 1.92 ns. As a result, the operating clock frequency for the processor cannot exceed 520 MHz for this configuration. The *Conf. 3* column represents a configuration in which resources are upsized aggressively and the achievable operating clock frequency is reduced further to 490 MHz. The final configuration (*Conf. 4 column*) presents an upper bound architecture for which the resource sizes may not be achievable in the design due to power/area and timing constraints.

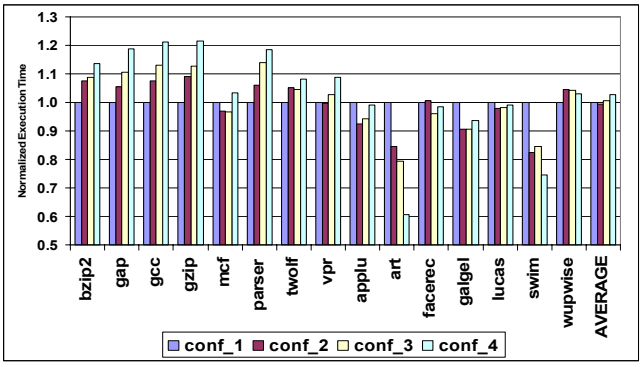

Figure 4. Normalized execution time for different configuration with reduced operating frequency compared to the baseline architecture.

Figure 4 shows the performance for the different configurations described above (normalized to the baseline configuration), while operating at their maximum achievable operating frequency. Figure 5 provides more insight on the frequency of ROB completely filling up during execution, for the different configurations.

For the INT benchmarks, it can be seen from Figure 5 that although increasing the ROB, RF and IQ sizes in configurations 2, 3 and 4 reduces the frequency of resource saturation (i.e., complete fill up) during execution, lowering the operating frequency impacts the performance negatively, as can be seen in Figure 4. In other words, for the trade-off between resource upsizing (and hence reducing their saturation rate) and lowering the clock frequency, the latter becomes more important and plays a major role in deciding the performance of the INT benchmarks. An exception is mcf for which upsizing the resources results in performance increase. Interestingly such increase exists only for intermediate and aggressive configuration (conf_2 and conf_3) but as we move to upper bound configuration (conf_4) we witness a performance impact.

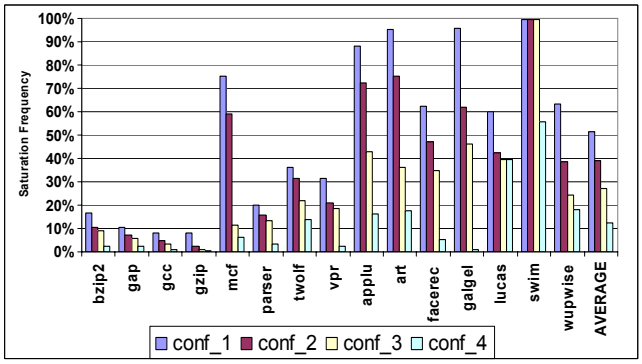

Figure 5. Case Study: Frequency of ROB saturation during execution, for different configurations.

To provide better insight we refer to how frequently ROB saturates across different configurations for mcf as reported in figure 5. As shown the ROB saturation rate drop significantly when moving from baseline configuration to aggressive configuration, but this decrease slows down significantly when moving to upperbound configuration. This in fact indicates a clear trade-off between the benefit of resource resizing on reducing resource saturation frequency and the impact it has on operating frequency.

Table 2. Configurations with pipelined resource access

Processor Configuration	Conf. 1 baseline	Conf. 2 intermediate	Conf. 3 aggressive	Conf. 4 upper bound
RF size	32	48	64	96
ROB size	24	32	48	64
IQ size	12	24	32	48
RF access time (ns)	1.76	1.92	2.03	2.19
ROB access time (ns)	1.43	1.58	1.83	1.91
Operating Freq (MHz)	560	560	560	560
Pipeline Depth	5	RF: 2 cycle; Bypass: 2 cycle	RF, ROB: 2 cycle; Bypass: 2 cycle	RF , ROB: 2 cycle; Bypass: 2 cycle; wakeup: 2 cycle

For the FP benchmarks shown in Figure 4, it can be seen that there is a performance improvement for some benchmarks, and performance degradation for others, as resources are upsized and operating frequency reduced. *applu* is similar to *mcf* with a performance improvement when going from *conf_1* to *conf_2* and *conf_3*, but performance degrades when we further increase the

size of resources at the cost of lowering the operating frequency. The maximum performance benefit is achieved for *art* which is consistent with the results in figure 5, with a significant saturation frequency reduction across different configurations. On average there is less than 0.5% performance improvement for intermediate configuration compare to our baseline. The aggressive and upper bound configuration results in performance degradation by up to 13% and 21%, and 0.7% and 2.8% on an average, compared to our baseline.

3.3 Impact of Pipelining Resource Access

In this section, we study the same scaled configurations presented in the previous section, but with the difference that this time we try to achieve the same operating clock frequency as the baseline architecture for all configurations. We refer to the RF access time of 1.76 ns for the baseline (or target) clock frequency as the worse pass delay (WPD). We pipelined resources with access times greater than the baseline WPD. This was the case for the RF in the intermediate configuration (as shown in Table 2). After pipelining, any access to the RF would require two processor cycles. To model this, it becomes necessary to modify several processor pipeline stages in which the RF is accessed. These include the *(i)* register rename logic, *(ii)* issue stage, and *(iii)* bypass logic. The bypass logic in particular requires an extra level. As previous studies [2] have shown, such an extra level leads to significant complexity in the pipeline. In this case, the data is available at the time of data bypass, then they are disappearing in the next cycle and finally they will be available the next cycle from register files. Using one level of bypass creates holes which as explained in [2] are undesirable and would lead to significant complexity to the issue logic. To avoid such complexity, an alternative is to just have two levels of bypass. In such a case only the last level of bypass is kept to avoid "holes" when there is an access to register file. We modify the pipeline according to this latter approach. For the aggressive configuration, in addition to pipelining the RF access, ROB access is also pipelined since its access delay is greater than the WPD, as shown in table 2. As such we need to modify the following additional pipeline stages (in addition to the changes described above for the intermediate configuration): *(i)* dispatch stage, and *(ii)* commit stage. For the upper bound configuration, in addition to pipelining the RF and ROB, we assume that the wakeup stage also required to be pipelined since its access delay goes above WPD. In other words, if the produced source operand *tags* are available in a given cycle,

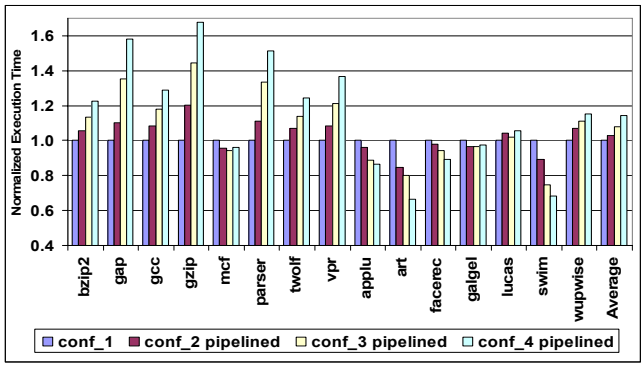

Figure 6. Normalized execution time for different configurations with additional pipelining while operating at the same frequency.

they will be broadcast in the next cycle. It should be noted that this is different from bypassing the *data* which has already been pipelined in the intermediate, aggressive and upper bound configurations.

Figure 6 shows the results for the pipelined configurations with the same operating frequency as the baseline configuration (target frequency) for all the scaled up configurations (i.e., *conf_2, conf_3* and *conf_4*).

As can be seen, a performance degradation is observed across most INT benchmarks. For all these configurations, introducing additional pipelining increases the branch miss prediction penalties. This is more visible in the INT benchmarks as their performance is more susceptible to branch penalties, compared to the FP benchmarks. For FP benchmarks there is a significant variation in performance impact. In *applu*, *art, facerec,* and *swim* there is a notable performance improvement. For these benchmarks, the benefit of increasing the resource sizes (ROB/IQ/RF) overcomes the negative impact of additional pipelining. This is not the case for *lucas* and *wupwise*. Interestingly these are the benchmarks with lowest resource saturation among all FP benchmarks, as reported in figure 5. As such, the negative impact of additional pipelining would be dominant in their overall performance. On average the performance degradation is 2.9%, 8.1% and 14.3% for the intermediate, aggressive, and upper bound configurations respectively, compared to the baseline configuration.

As the results in this and the previous subsection show, neither of these two techniques result in a noticeable performance improvement (in most cases there is actually a performance degradation). We conclude that the negative impact of pipelining and frequency scaling on the performance becomes dominant over the positive impact of resource upsizing. In the next section we describe a different approach that actually improves overall performance significantly during resource upsizing.

4. Proposed Approach

The results in figure 2 suggest that there is no need for aggressive resource upsizing during the normal period. Based on our observations in subsections 3.2 and 3.3, we realized the need for an adaptive resource scaling technique based on cache misses, that allows the processor to use smaller resources (having a lower access time) during the *normal period* and larger resources (having a higher access time) during the *cache miss period*.

Our proposed approach thus applies resource scaling and uses aggressively upsized resources only during the *cache miss period*. During the *normal period*, resources are kept at their typical size and as such they can operate at the target operating frequency (WPD of 1.76 ns for the RF from table 2, which determines the processor operating frequency). During the *cache miss period* we scale up the resource sizes, which results in an increase in their access time.

For the resources with access time greater than the typical WPD, we propose pipelining, as discussed in section 3.3. It should be noted that while the goal of our work is to increase performance, the power and complexity overhead should also be taken into consideration while implementing our approach. Pipelining the ROB and wakeup logic for instance significantly increases design complexity. As such we only consider a more realistic scenario,

which is scaling up resources to configuration 2 (intermediate), in which only the access to the RF needs to be pipelined.

Note that while it is also possible to increase resource sizes further, we avoid doing so due to the extra power overhead of a larger ROB, RF and particularly the IQ.

As part of our approach, we propose two techniques for scaling resources.

In the first technique, we start with resources at their typical size and only increase their size when we encounter an L2 cache miss. We refer to this technique as *L2 miss driven resource scaling (L2RS)*.

In the second technique, we perform scaling on a single L2 cache miss or when at least two DL1 cache misses are pending. We refer to this algorithm as *L2 and multi-DL1 miss driven resource scaling (L2ML1RS)*. In both techniques, the resource size is returned to its normal size once all of the cache misses have been serviced and the scaled up part has no more data.

It should be noted that both these techniques have a fairly simple implementation and do not add significant complexity to the embedded processor pipeline, since the scheduler in embedded processors already keeps track of miss load instructions in DL1 and L2 caches. Though, *L2ML1RS* is slightly more complex than *L2RS* since it requires counting the number of pending L1 cache misses. We now describe the circuit level modifications required to implement these techniques.

4.1 Circuit Modification

Table 2 indicates that increasing ROB size from 24 to 48 does not increase its access time beyond the baseline WPD. Consequently, the ROB can be designed with 48 entries without needing extra pipelining and still be accessed at the target clock frequency. We use the same reasoning to design the IQ with 24 entries (instead of 12 in the baseline configuration). For our adaptive resizing scheme, we divided the ROB into two partitions of 32 and 16 entries each. For power conservation we used gated Vdd technique to power gate the part with 16 entries always, except during cache miss period. Similarly, we divided the IQ into two equal partitions of 12 entries and power gated one partition always, except during a cache miss period. For the RF however, increasing its size results in an access time greater than the baseline WPD (table 2). The challenge here is to design the RF in such a way that its access requires only a single processor cycle when its size is 32 entries and a two cycle access time only when we increase its size.

Figure 7 shows our proposed solution which requires a minimal modification to a unified register file (unlike more complex banking schemes). As the results in figure 3 reported, among all RF components, the bitline delay increase is responsible for more than 90% of access time increase for a 64 entry RF compare to 32 entries one. This is due to the fact that bitline delay is decided by its equivalent capacitance which in turn is proportional to the number of RF entries (rows).

Accordingly, to be able to achieve an access delay close to the baseline WPD for a 64 entry RF when only 32 of its entries are being used, we need to reduce the equivalent capacitance on the bitline by eliminating the diffusion capacitance of the 32 unused entries.

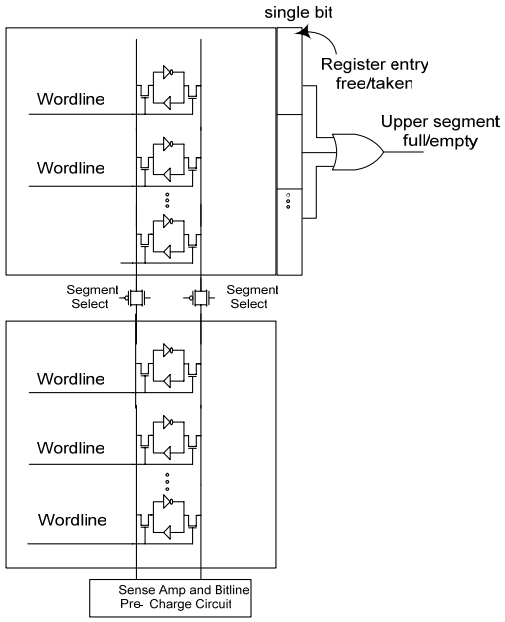

Figure 7. Proposed circuit modification for RF

For this purpose, the RF is divided into two segments of 32 entries each which are connected through pass transmission gates as shown in figure 7. This allows the upper segment bitline to become isolated from the lower segment bitline if the pass gate is off. It should be noted that all other components such as the bitline, sense-amp, etc. are shared for both structures. During the *normal period* the upper segment is power gated and the transmission gate is turned off to isolate the lower bitline segment from the upper bitline segment. Only the lower segment bitline is pre-charged during this period. Since the upper bitline segment is floating, the bitline capacitance during normal period is decided by the lower segment bitline. As such, the bitline delay in this case remain close to the bitline delay of a 32-row register file (the only difference is the delay added by the source capacitance of the pass gate, which is negligible). It should be noted that in our baseline RF configuration, the sense amp is used for pre-charging the bitline in addition to sensing the difference of voltage on the bitline and its complement during a read operation, as shown in figure 7. If the baseline architecture has a separate pre-charge circuitry, then such pre-charge circuitry has to be duplicated for both the top and bottom segments in our proposed architecture.

In addition, we need to be able to detect when the upper segment is empty (for downsizing at the end of cache miss period when the added segment is empty). To do that we have augmented the upper segment with one extra bit per entry. This bit is set when an entry (register) is taken and is being reset when the entry is committed. By ORing these bits we can detect when the segment is empty.

Since the remaining components of RF delay change very slightly, the RF access delay remains close to that for a 32-row RF when the upper segment is isolated. The delay of accessing the RF while the upper segment is isolated is 1.79 ns, which is only slightly larger than for a baseline 32-row register file (1.76 ns). When the upper segment is active and the RF has its full size, its access requires 1.93 ns, which can be completed in two processor cycles.

5. Experimental Results

In this section we present experimental results to show how our adaptive resource resizing approach impacts processor performance. First we describe our simulation framework. Table 3, describes the base processor architecture in detail, which operates at 560 MHz frequency and is similar to IBMPowerPC 750FX embedded microprocessor [11]. We use SPEC2K benchmarks executed with reference data sets, and compiled with the O4 flag using the Compaq compiler. The architecture was simulated using an extensively modified version of MASE (SimpleScalar 4.0) [20]. The benchmarks were fast–forwarded for 1 billion instructions, then fully simulated for 1 billion instructions. We used a modified version of CACTI4 [19] for estimating access time of the ROB and RF. For estimating energy consumption of our adaptive technique we integrate Wattch [21] into our simulator infrastructure. We used process parameters for a 65nm process at 560MHz with 1V supply voltage.

In figure 8(a) and (b) we report the results for both our proposed L2RS and L2ML1RS techniques. In figure 8(a) we report the performance improvement in terms of IPC and in figure 8 (b) we report the energy-delay product of adaptive technique compare to the non-adaptive architecture (*conf_1*). As can be seen from figure 8 (a), the performance improvement is significant across most benchmarks for L2RS, up to 24% for *art* and an average of 6.8%. The performance improvement for L2ML1RS is even more, up to 34% for *swim* and an average of 9.2%. The performance improvement for L2ML1RS is more in this case compared to L2RS because it presents more opportunities for upsizing the resources.

Table 3. Processor organization

L1 I-cache	32KB, 2 cycles	Instruction queue	12 entry
L1 D-cache	32KB, 2 cycles	Register file	32 entry
L2 cache	256KB, 8 way, 12 cycles	Load/store queue	12 entry
Fetch and dispatch	2 wide	Branch predictor	g-share,256-entry BTB
Issue	2 way out of order	Arithmetic unit	2 integer, 2 floating-point units
Memory	60 cycles	Complex unit	1 INT, 1 FP multiply/divide units
Reorder buffer	24 entry	Pipeline	5 cycles

The exception in the overall observations is *gzip* which degrades in performance by 1.36% for L2ML1RS. As can be seen in figure 8(a), the performance improvement for *gzip* when applying the L2RS technique is negligible while there is not much opportunity for resource scaling for this benchmarks (figure 2). As such applying L2ML1RS which comes with more resizing frequency resulted in performance degradation. Another trend seen in figure 8 is for the INT benchmarks, for which the upsizing frequency and as such the performance benefit is lower for the FP benchmarks.

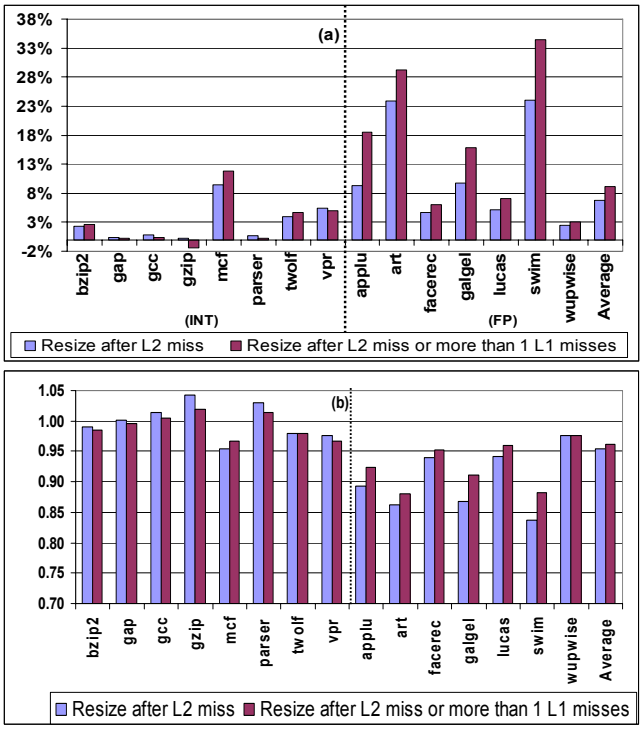

Figure 8. Experimental results: (a) performance improvement with L2RS and L2ML1RS compare to conf_1 (b) Normalized energy-delay product compare to conf_1.

Recalling from results of experiments presented in section 3, in the same benchmarks the performance for INT benchmarks degraded more compared to the FP benchmarks.

Finally, the energy-delay product of our adaptive architecture compared to the non-adaptive architecture (*conf_1*) is shown in figure 8(b). As can be seen for most benchmarks our technique results in overall reduction in energy-delay product. Exceptions are *gcc*, *gzip* and *parser* for which the energy-delay increased in both L2RS and L2ML1RS. Interestingly in the same benchmarks our technique did not result in performance improvements and as such the resizing would only result in increasing processor energy. The average energy-delay is reduced by 4.6% and 3.8% respectively for L2RS and L2ML1RS compared to the baseline configuration.

6. Conclusion and Future Work

In this work we presented the results of an architectural study which shows that the ROB, RF and IQ size are a performance bottleneck in embedded processor. We further studied the effect of increasing size of ROB/IQ and RF on processor performance. As our results demonstrate, increasing the size of these units (upsizing) would increase their access time and as such the processor cannot operate at its target clock frequency. We study two possible approaches to overcome this problem; either lowering the processor operating clock frequency or pipelining access to these units. Our result show that none of these techniques resulted in a noticeable performance improvement and in fact results in performance degradation for most of the studied cases. In response we propose two adaptive resource resizing techniques; L2RS and L2ML1RS. In L2RS the resources are increased during L2 cache miss service time. In L2ML1RS we increase the resources during L2 cache miss service time or when

at least two DL1 cache misses are pending. Our results show a significant performance improvement and overall energy-delay reduction of on average 9.2% (upto 34%) and 3.8% respectively across SPEC2K benchmarks for L2ML1RS. Applying L2RS resulted in 6.8% (24%) performance improvement and 4.6% energy-delay reduction. We also present the circuit modification to realize these techniques, which is shown to be minimal.

7. REFERENCES

[1] A. Terechko, M. Garg, H. Corporaal, "Evaluation of speed and area of clustered VLIW processors," VLSI Design, 2005. 18th International Conference on , vol., no., pp. 557-563, 3-7 Jan. 2005.

[2] J.L. Cruz, A. González, et al., "Multiple-banked register file architectures", International Symposium on Computer Architecture, pp. 316-325, Vancouver, Canada, June 2000.

[3] J.H. Tseng, K. Asanovic, et al., "Banked Multiported Register Files for High-Frequency Superscalar Microprocessors", International Symposium on Computer Architecture, San Diego, California, USA, 9–11 June 2003.

[4] Stijn Eyerman, Lieven Eeckhout, Koen De Bosschere, "Efficient Design Space Exploration of High Performance Embedded Out-of-Order Processors", DATE 2006.

[5] Joseph Sharkey, Dmitry Ponomarev, "An L2-Miss-Driven Early Register Deallocation for SMT Processors", ICS 2007.

[6] O. Ergin, et al.,, "Increasing Processor Performance through Early Register Release", Int'l Conference on Computer Design, 2004.

[7] S. Rixner,W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens. "Register organization for media processing." In Proc. of the 6th Intl. Symp. on High-Performance Computer Architecture, pages 375–386, 1999.

[8] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko G. Vranesic. "The Multicluster architecture: Reducing cycle time through partitioning." In MICRO-30, pages 149–159, 1997.

[9] A. Seznec, E. Toullec, and O. Rochecouste. "Register write specialization register read specialization: A path to complexity-effective wide-issue superscalar processors." In MICRO-35, Turkey, November 2002.

[10] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi. "Reducing the complexity of the register file in dynamic superscalar processors." In MICRO-34, December 2001.

[11] IBM Corporation. PowerPC 750 RISC Microprocessor Technical Summary. www.ibm.com.

[12] G. Kucuk, D. Ponomarev, and K. Ghose. "Low-complexity reorder buffer architecture." Proceedings of the 16th ACM International Conference on Supercomputing, 2002.

[13] Goto, M. and Sato, T., "Leakage Energy Reduction in Register Renaming", in Proc. 1st Int'l Workshop on Embedded Computing Systems (ECS) held in conjunction with 24th ICDCS, pp.890-895, March 2004.

[14] A. Buyuktosunoglu, D. Albonesi, S. Schuster, D. Brooks, P. Bose, and P. Cook, "A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors," Proc. Great Lakes Symp. VLSI Design, 2001.

[15] D. Folegnani and A. Gonzalez, "Energy-Effective Issue Logic," Proc. Int'l Symp. Computer Architecture, pp. 230-239, 2001.

[16] D. Ponomarev, G. Kucuk, K. Ghose, "Dynamic Resizing of Superscalar Datapath Components for Energy Efficiency," IEEE Transactions on Computers ,vol. 55, no. 2, pp. 199-213, February, 2006.

[17] Steven E. Raasch, Nathan L. Binkert and Steven K. Reinhardt, "A Scalable Instruction Queue Design Using Dependence Chains", Proceedings of 29th Annual of International Symposium on Computer Architecture, 2002 Page(s): 318-329.

[18] S. Palacharla, N. Jouppi, and J. E. Smith. "Complexity effective superscalar processors." In ISCA-24, pages 206–218, June 1997.

[19] "Cacti4," http://quid.hpl.hp.com:9081/cacti/.

[20] SimpleScalar4 tutorial, SimpleScalar LLC. http://www.simplescalar.com/tutorial.html

[21] D. Brooks, V. Tiwari, and M. Martonosi. "Wattch: A framework for architectural-level power analysis and optimizations." In 27th Annual International Symposium on Computer Architecture, June 2000.

[22] S. Geissler et al., "A low-power RISC microprocessor using dual PLLs in a 0.13/spl mu/m SOI technology with copper interconnect and low-k BEOL dielectric", in ISSCC 2002.