DYNAMIC ALLOCATION OF DATAPATH RESOURCES FOR LOW POWER*

Dmitry Ponomarev, Gurhan Kucuk, Kanad Ghose
Department of Computer Science
State University of New York, Binghamton, NY 13902–6000
e–mail:{dima, gurhan, ghose}@cs.binghamton.edu

Abstract

We show by profiling the execution of SPEC95 benchmarks that the usage of datapath resources in a modern superscalar processor is highly dynamic and correlated. The one–size–fits all philosophy used for permanently allocating datapath resources in a modern superscalar CPU is thus complexity–ineffective due to the overcommittment of resources in general. We propose a strategy to dynamically and simultaneously adjust the sizes of two such correlated resources – the dispatch buffer (also known as an issue queue) and the reorder buffer – to reduce power dissipation in the datapath without significant impact on the performance. We also show how the resizing technique can be augmented with dynamic adaptation of dispatch rate. Representative results show reduction in power dissipation of 69% for the dispatch buffer and of 52% for the reorder buffer with an average IPC loss below 8.5%.

1. Introduction

Today's superscalar processors are designed to achieve a high level of performance; they are also designed following a "one–size–fits–all" philosophy. The one–size–fits–all approach results in the permanent allocation of datapath resources to maximize performance across a wide range of applications. Earlier studies have indicated that the overall performance, as measured by the IPC, varies widely across applications [Wall 91]. The IPC also varies quite dramatically within a single application, being a function of the program characteristics (natural ILP) and that of the datapath and the memory system. As the natural ILP varies in a program, the usage of datapath resources also changes drastically – where the natural ILP is low, the datapath resources remain overcommitted.

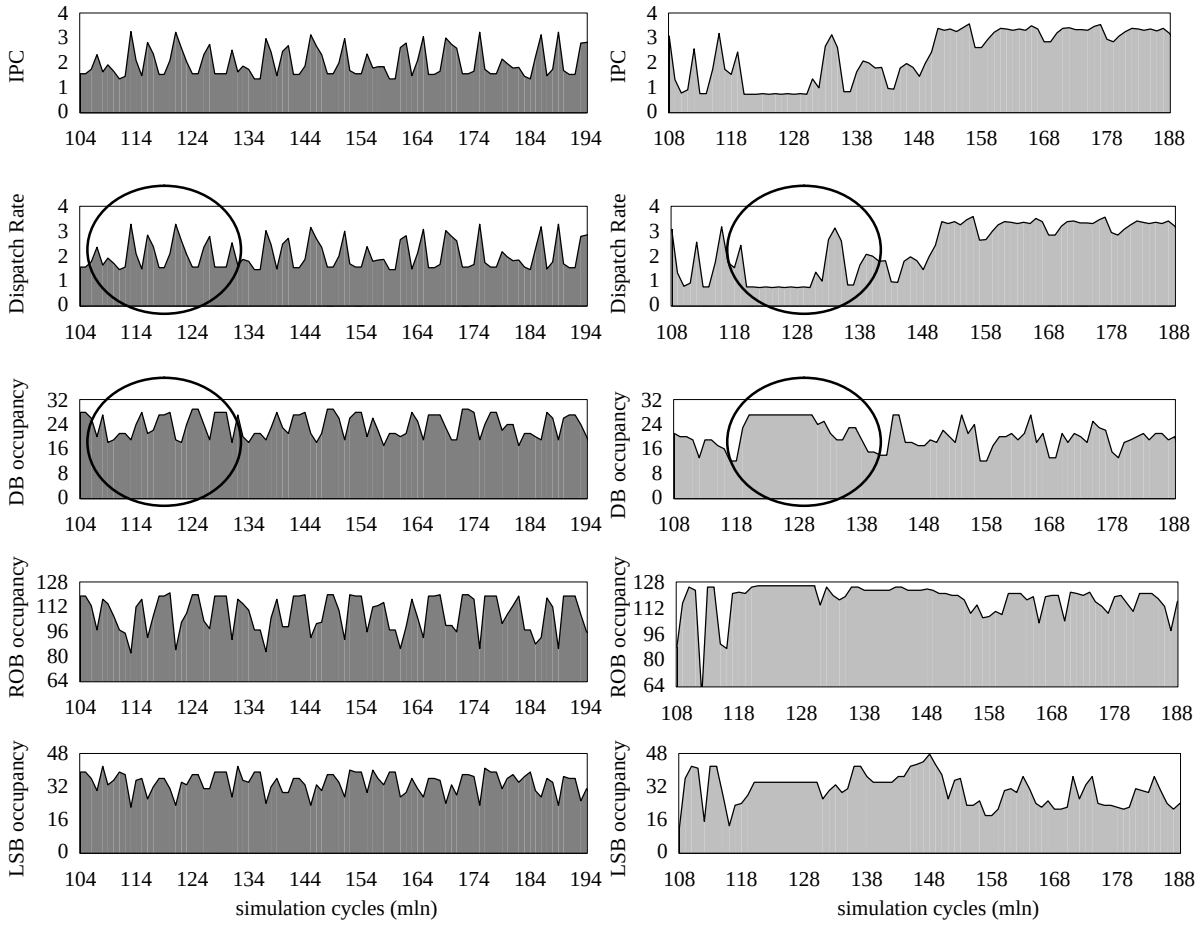
We profiled the execution of SPEC 95 benchmarks keeping track of the changes in the IPCs, dispatch rates and the average occupancy (number of valid entries) of various structures used in superscalar processors for supporting aggressive out–of–order execution. Namely, we considered the reorder buffer (ROB), the dispatch buffer (DB) and the load/store buffer (LSB). Figure 1 shows representative results for two of the floating point benchmarks for 200 million instructions after skipping the first 200 million. Samples of IPC, dispatch rate and the average occupancy of the ROB, the DB and the LSB have been taken every 1 million cycles. These samples reflected the activity within the most recent window of 1 million cycles.

As these representative results show, the average occupancy of the structures changes significantly throughout the course of program execution. Furthermore, the oscillations in the DB and the ROB occupancies are correlated – where the DB occupancy is higher, the ROB occupancy also increases. The results also suggest that it is hardly possible to efficiently control the allocation of the DB and the ROB entries solely based on the instruction dispatch rate. Indeed, changes in the dispatch rate may effect the DB and the ROB occupancies in two ways. In some situations, higher dispatch rate corresponds to increased occupancies of the ROB and the DB, because more instructions are in–flight at the same time, and the natural ILP within the code fragment is high. This is, for example, the case with *apsi* benchmark (part I of figure 1), where the spikes in the dispatch rate graph closely match the peaks of the lower three graphs. However, in other situations the opposite is true, the example of which is a snapshot of *hydro2d* benchmark shown in part II of Figure 1. Here, lower dispatch rate corresponds to higher occupancy of the DB and the ROB. This is a direct consequence of data dependencies among the instructions with long latencies (low natural ILP). In such scenarios, the momentary increase in the instruction dispatch rate results in a quick saturation of the DB (or the ROB, depending on the configuration), which leads to multiple instruction blockings at the time of dispatch, and inevitable degradation of the dispatch rate.

The premise of this paper is to conserve power and energy by dynamically and simultaneously resizing *multiple* datapath resources to closely track the demands of the application; unused resources are deactivated momentarily and reactivated when the resource demands go up. The net result is a drastic savings in the overall power/energy with minimal impact on performance. We also show that an additional level of control, namely the dynamic variation of the dispatch rate can be used to achieve further power/energy savings.

The rest of the paper is organized as follows. Related work is discussed in section 2 followed by the description of the dynamic datapath resource allocation strategy in section 3.

* supported in part by DARPA through contract number FC 306020020525 under the PAC–C program, by the IEEC at SUNY–Binghamton and the NSF through award no. MIP 9504767 and EIA 9911099



Part I. Profile of **apsi** benchmark

Part II. Profile of **hydro2d** benchmark

Figure 1. Dynamic behavior of two SPEC95 floating point benchmarks. Results are shown for 200 million instructions after skipping the first 200 million. Measurements of IPCs, dispatch rates and average DB, ROB and LSB occupancies were taken every 1 million cycles. Each of these measurements reflect benchmark's behavior within the most recent window of 1M cycles.

Section 4 describes our simulation methodology and processor configuration. Results are presented in section 5 followed by our conclusions in section 6.

2. Related Work

It is well–documented that the major power/energy sinks in a modern superscalar datapath are in the dynamic instruction scheduling components (consisting of the dispatch buffer – aka "issue queue", reorder buffer and the physical registers) and the on–chip caches. As indicated in [WM 99], as much as 55% of the total power dissipation occurs in the dynamic instruction scheduling logic, while 20% to 35% (and sometimes higher) of the total power is dissipated within the on–chip cache hierarchy. It is therefore not surprising to see a fair amount of recent research being directed towards the reduction of the energy dissipation within these components. Of this large body of work, we mention only the contributions that focus on dynamic resource allocation.

The dynamic allocation of a single datapath resource was studied in [BA+ 00, BAS+ 01, FG 01]. In [BAS+ 01, FG 01], the authors explored the design of an adaptive issue queue, where the queue entries were grouped into independent modules. The number of modules allocated was varied dynamically to track the ILP; power savings with minimal impact on performance was achieved by turning off unused modules. In [BAS+ 01], the activity of an issue queue entry was indicated by its ready bit (which indicates that the corresponding instruction is ready for issue to a function unit). The number of active issue queue entries, measured on a cycle–by–cycle basis, were used as a direct indication of the resource demands of the application. The control logic's energy overhead was also considered carefully in this design. An additional feature was the possibility of using a faster clock when the issue queue had fewer active modules; an asynchronous–synchronous interface was thus made an integral part of the queue to exploit this feature; it was left un-

clear how the remaining components of the datapath could be clocked at a higher rate to take advantage of a faster issue queue. In [FG 01], Folegnani and Gonzalez introduced a FIFO issue queue that permitted out–of–order issue but avoided the compaction of vacated entries within the valid region of the queue to save power. The queue was divided into regions and the number of instructions committed from the most–recently allocated issue queue region in FIFO order (called the "youngest region") was used to determine the number of regions within the circular buffer that was allocated for the actual extent of the issue queue. To avoid a performance hit, the number of regions allocated was incremented by one periodically; in–between, also at periodic intervals, a region was deactivated to save energy/power if the number of commits from the current youngest region was below a threshold. The energy overhead of the control logic for doing this resizing was not made clear. In [KGPK 01], we have introduced a dispatch buffer design that achieves significant energy savings using a variety of techniques, including the use of zero–byte encoding, comparators that dissipate energy on a match and a form of dynamic activation of the accessed regions of the dispatch queue using bit–line segmentation. In [BA+ 00], a reconfigurable cache organization is used to track program characteristics dynamically and save energy; a similar approach was also explored in [Alb 99, BA+ 00]. Bit–line segmentation and subbanking also achieves similar results [GK 99]. In [PKG 01], we have also explored the dynamic allocation of banks within a multi–banked register file to conserve power/energy with a very low IPC penalty. All of the above are examples of allocating a single datapath resource to achieve power/energy savings. In reality, however, resource usage within a datapath are highly correlated, as seen from Figure 1. For example, there is a direct correlation between the occupancies of the dispatch buffer and the reorder buffer (Figure 1).

Dynamic allocation of multiple datapath resources to conserve power/energy was first studied in the context of a multi–clustered datapath (with non–replicated register files) in [ZK 00], where dispatch rates and their relationship to the number of active clusters were well–documented. A similar but a more explicit study was recently reported in [BM 01], for the multi–clustered Compaq 21264 processor with replicated register files. The dispatch rate was varied between 4, 6 and 8 to allow an unused cluster of function units to be shut off completely. The dispatch rate changes were triggered by the crossing of thresholds associated with the floating point and overall IPC, requiring dispatch monitoring on a cycle–by–cycle basis. Significant power savings within the dynamic scheduling components were achieved with a minimum reduction in the IPC. The dynamic allocation of the reorder buffer – a major power sink – was left completely unexplored in this study. In this paper, we show how multiple resources, including the dispatch buffer and the reorder buffer can be controlled dynamically and simultaneously to achieve significant power savings with very simple control logic that avoids performance/usage monitoring on a cycle–by–cycle basis.

In [CGC 00], resource usages are controlled indirectly through pipeline gating and dispatch mode variations by letting the Operating System dictate the IPC requirements of the application. Along similar lines, we are currently exploring a compiler–directed approach for dynamically allocating datapath resources based on the IPC requirements, using the basic infrastructure described in this paper.

3. Dynamic Datapath Resource Allocation Strategy

Figure 2 depicts the superscalar datapath that employs the issue–bound operand fetch policy. Here, even if input registers for an instruction contain valid data, these registers are not read out at the time of dispatch. Instead, when all the input operands of an instruction waiting in the DB are valid and a function unit of the required type is available, all of the input operands are read out from the register file (or as they are yet to be written to the register file, using bypassing logic to forward data from latter pipeline stages) and the instruction is issued. The dispatch/issue logic can be implemented using a global scoreboard that keeps track of instructions and register/FU availability. Alternatively, an associative logic can be used to update the status of input registers for instructions waiting within the DB (as shown in Figure 2). Examples of processors using this datapath style are the MIPS 10000, 12000, the IBM Power 3, the HP PA 8000, 8500, and the DEC 21264 [Mi 9X, Bh 96].

We concentrate on the dynamic allocation and deallocation of two major datapath resources, namely the DB and the ROB. The dynamic allocation of other resources (register files, load/store buffer etc.) in conjunction with the DB and the ROB is being currently explored and the results are to be presented in a forthcoming paper. The organization of the DB allowing for incremental allocation is depicted in Figure 3. The ROB is partitioned in a similar fashion. The DB and the ROB are each implemented as a number of independent partitions. Each partition is a self–standing and independently usable unit, complete with its own precharger, sense amps, input/output drivers. In fact, the number of entries in a DB and ROB partition are 8 and 16 respectively in our studies. The maximum number of partitions available is 8 for the DB (for a total of 64 entries) and 8 for the ROB (for a total of 128 entries). A number of the DB (or ROB) partitions can be strung up together to implement a larger DB (or ROB). The connection running across the entries within a partition (such as bitlines, forwarding bus lines etc.) can be connected to a common through line (shown on the right in Figure 3) through the bypass switches to add (i.e., allocate) the partition to the current DB (or ROB) and extend the effective size of the DB (or ROB). Similarly, by turning off the bypass switches associated with a partition, it can be deallocated from the current DB (or ROB). Unallocated partitions are turned off to save power/energy. Entries can be allocated in the DB – across one or more active partitions in any order;

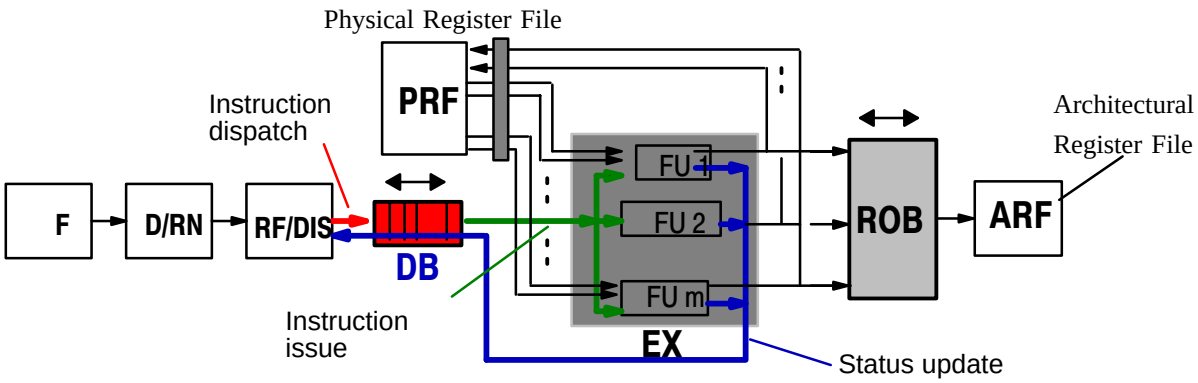


Figure 2. Superscalar datapath with issue–bound operand fetching

an associative searching mechanism is used to look up free entries. Entries within the active ROB are used in a circular FIFO, and their allocation/deallocation strategies are not as straightforward as that for the DB.

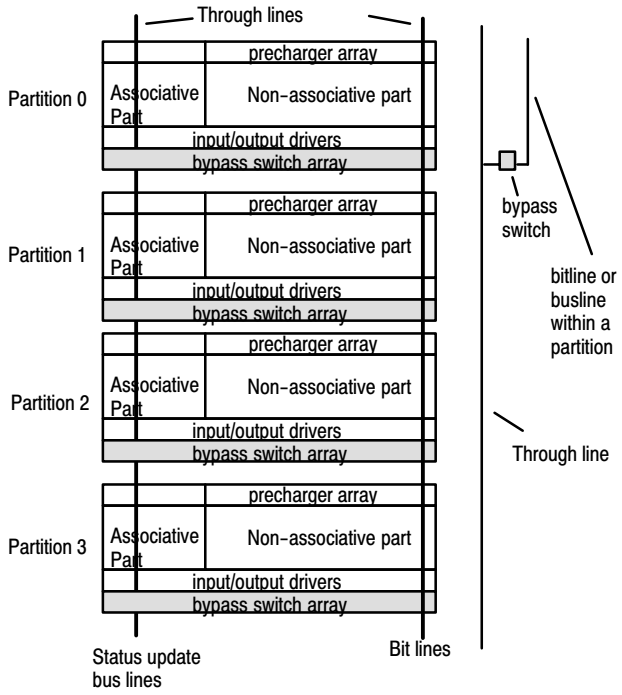


Figure 3. Partitioned DB

Below we discuss the strategies for the DB and the ROB resizing in the context of the aforementioned datapath.

3.1 Dispatch Buffer

Two separate phases are implemented for the DB resizing: One is used for resizing the structure down, and the other is used for resizing it up. We describe each of these phases below.

Resizing down

The decision of whether to decrease the DB size or leave it at its current level is made periodically – once every DB_UPDATE_PERIOD cycles. During this period, the DB occupancy (equal to the number of allocated entries within the active region of the DB) is sampled several times with the frequency of once every DB_SAMPLE_PERIOD cycles. The average of these samples is taken as the ACTIVE_DB_SIZE in the current update period. Both DB_UPDATE_PERIOD and DB_SAMPLE_PERIOD were chosen as powers of 2 to let the integer part and the fractional part of the computed DB occupancy be isolated easily in the occupancy counter. This avoids the use of a full–fledged division logic. Figure 4 depicts the relationship between DB_UPDATE_PERIOD and DB_SAMPLE_PERIOD.

At the end of every update period, the difference (DIFF=CURRENT_DB_SIZE–ACTIVE_DB_SIZE) is computed, where CURRENT_DB_SIZE is the size of the dispatch buffer at the end of the update period and ACTIVE_DB_SIZE is the average of the samples taken during this update period. If DIFF is less than the minimum amount by which DB size can be reduced (DB_DECREMENT, this is equal to the partition size), then the decision is to leave the CURRENT_DB_SIZE unchanged. If, however, DIFF is greater (or equal) than DB_DECREMENT, two scenarios are possible depending on whether an aggressive or a conservative (non–aggressive) downward resizing is implemented. If a conservative scheme is in use, then CURRENT_DB_SIZE is always decremented by one partition (8 entries in the current implementation). If an aggressive scheme is used, then CURRENT_DB_SIZE is decremented by the maximum allowable number of partitions as determined by DIFF.

ACTIVE_DB_SIZE provides a reasonable approximation of the average DB occupancy within the most recent DB_UPDATE_PERIOD. At the end of every sample period, we record the DB occupancy by using bit–vector counting logic for each active partition and adding up these counts. Estimation of power consumption in this process is beyond

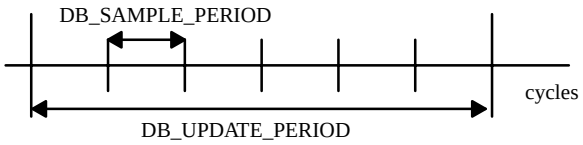


Figure 4. Sample period and update period used for resizing the DB down

the scope of this paper. The fact that this counting is performed only at the end of the sampling period (as opposed to maintaining a dynamic count on a cycle–by–cycle basis) suggests that the energy dissipated in estimating the DB occupancy is negligible. The difference between the current size of the DB and ACTIVE_DB_SIZE in the update period, as computed according to the resizing strategy described in this section, indicates the degree of DB overcommitment.

After the decision to reduce the DB size has been made, it is necessary to wait till all the instructions are issued from the partition that is being turned–off. Only after that the actual resizing can take place. During this transition period, the dispatch would stall if the DB entry allocated for the instruction belongs to the partition that will become inactive. Otherwise, dispatches may continue to the active partitions of the DB.

Resizing up

The second phase of the resizing strategy is implemented to scale the DB size back up once the demands of the application begin to require more resources. Here, we use the DB overflow counter (DB_OVERFLOW), which counts the number of cycles when dispatch is blocked because of the absence of a free entry in the DB to hold the new instruction. This counter is initialized to 0 every DB_UPDATE_PERIOD cycles and is incremented by one whenever instruction dispatch is blocked because of the unavailability of a free DB entry to hold a new instruction. Once the overflow counter exceeds the OVERFLOW_THRESHOLD, the CURRENT_DB_SIZE is incremented by adding another partition. After that, the counter is reinitialized. Note, that for performance reasons, the process of increasing the size of the DB is more aggressive than the process of shrinking the DB. Instead of waiting for DB_UPDATE_PERIOD cycles to increase the DB size, this is done as soon as the overflow counter reaches the OVERFLOW_THRESHOLD. To avoid the occurrence of several such incremental updates in one update period, especially if aggressive downsizing strategy is employed which can sometimes reduce the DB size unjustifiably, we reset the update period counter every time the DB size increase is triggered. This is to avoid the instability in the system associated with the fact that DB is sampled for different maximum sizes in the same update period.

By varying the different thresholds we can make either one of the processes more aggressive depending on the design goals as dictated by power (resizing down) / performance (resizing up) tradeoffs. Notice that a resource is scaled down only if the sampling of the resource occupancy discovers that resource is overcommitted – this is to avoid a potential performance loss associated with more aggressive designs.

3.2 Reorder Buffer

The resizing of the ROB requires additional considerations because of the FIFO nature of the ROB. In particular, the ROB is a circular FIFO structure with two pointers – *ROB_tail* and *ROB_head*. *ROB_head* indicates the next free entry in the ROB and is used during instruction dispatch to reserve the entries in the ROB in program order. *ROB_tail* is used during commit stage to update the architectural registers in program order. Such an organization of the ROB suggests that in some situations partitions can be activated and deactivated only when the queue extremities coincide with the partition boundaries. This involves the following changes to the scheme described for the DB:

Resizing down

Once the decision to resize the ROB down is made, *ROB_head* pointer should not be allowed to exceed the newly established ROB_SIZE. To ensure this, dispatch will block to prevent the advance of the *ROB_head*. In addition, the resizing cannot take place until *ROB_tail* is smaller than *ROB_head*. These two conditions ensure that all instructions in the to–be–deactivated partition(s) are committed and no new instruction is dispatched into these partitions before the partitions are actually deactivated.

Resizing up

If *ROB_tail* <*ROB_head*, then the ROB size can be increased immediately. Otherwise, one should wait until this condition is reached (the tail pointer becomes a 0, which would preserve the correct order of instruction commitments).

Figure 5 depicts the possible combinations of *ROB_head* and *ROB_tail* disposition at the time the decision to resize the structure is made.

If the decision to resize the ROB down is made in the situation of Figure 5a, dispatch should continue until ROB_head=NEW_ROB_SIZE. That should be the last instruction allowed to dispatch unconditionally. At this point, dispatch stalls if *ROB_tail*>*ROB_head*. Otherwise, the ROB is resized down by deactivating the partition between NEW_ROB_SIZE and CURRENT_ROB_SIZE. Note, that many cycles can pass between the decision to resize and actual resizing. If this disposition of pointers is encountered when the decision to resize up is made, the resizing can be performed immediately, allowing the *ROB_head* pointer to increment till it reaches the new boundary.

In the case of Figure 5b the conditions for performing actual downsizing are the same. However, a new partition can not be added to the ROB until *ROB_tail* is less than *ROB_head*.

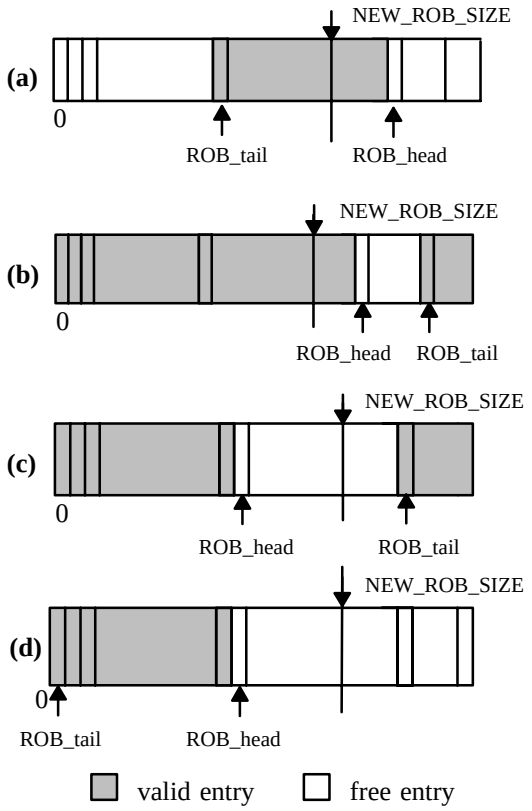


Figure 5. ROB resizing scenarios

To resize the ROB down in the situation depicted in Figure 5c, it is necessary to wait till *ROB_tail* becomes a zero (wraps around), so that all instructions in the to–be–deactivated region can commit. Dispatch could block if at the time the *ROB_head* reaches the NEW_ROB_SIZE boundary, *ROB_tail* pointer has not yet wrapped around. Similarly the size can not be increased until *ROB_tail* becomes a zero. However, no blocking at dispatch can occur in this case, because by the time *ROB_head* reaches the CURRENT_ROB_SIZE, *ROB_tail* would have already wrapped around (*ROB_head* is following *ROB_tail).*

Finally, no problems arise if the disposition of the ROB pointers is similar to Figure 5d. Resizing either way can be performed immediately.

Several features of the resizing procedure should be emphasized. First, the DB and the ROB sizes may be reduced at different times – this makes our control entirely distributed, where the ROB size only indirectly influences the decision of whether to resize the DB, and vice versa. Second, during the waiting period for resizing, no further resizing decision may be initiated till the previous one is completed. Third, we delay the update period (by initializing the appropriate counter) and ignore all of the samples taken so far, if the overflow threshold is reached and resizing logic increments the size of the structure. As a result, even if the ROB and the DB update periods are the same, they are not synchronized because the actual update period depends on the decisions of the resize–up logic of each structure.

3.3 Dispatch Rate Adaptation

To further reduce power dissipation within the DB and the ROB, we propose the use of variable dispatch rate based on the momentary IPC of the applications. Our study demonstrated that IPCs, as well as the effective dispatch rates, experience considerable variation throughout the course of execution. Thus, in the situations where there is no or little ILP within the code, temporarily limiting the maximum dispatch rate (MDR) could introduce considerable power savings (since fewer instructions are injected into the pipeline per cycle) without any substantial IPC loss.

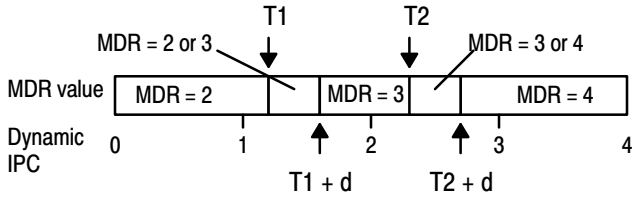


Figure 6. MDR selection based on dynamic IPC and applicable thresholds

To achieve this, we rely on measurements of the IPC in a certain period of time. In the forthcoming discussion, we call this period P and the IPC measured in this period is called dynamic IPC. At the end of every P, the average commit IPC during this period is estimated from the total number of commits made during this period P (kept in a commit counter) and the MDR for the next P is set based on thresholds as shown in Figure 6. P is chosen as a power of 2 to avoid complex division logic as described above. Three values of MDR were considered – 2,3 and 4. If the currently measured IPC turns out to be in the region between T1 and T1+d, then the MDR for the next period is set depending on the trend in the IPC change. If IPC has increased compared to the measurement obtained from the previous period, MDR is set to 3, otherwise it is set to 2. The same applies to the region between T2 and T2+d, where the selection is between the MDR of 3 and 4. For other regions, the MDR is unconditionally set as specified in Figure 6. In the course of decreasing the MDR, we record the dynamic IPC value at which this transition occurs. To limit further IPC drop due to the lower MDR, the subsequent changes in dynamic IPC are monitored and compared against this recorded value. If the IPC drops by more than X% (results for X=5 are shown in the result section) from this value, the MDR is unconditionally incremented by 1. This avoids substantial performance drop because of the limitation to the dispatch rate and effectively nullifies the act of decrementing the MDR should it happen to be a factor of further performance loss.

The values of T1 and T2 are not static, but instead, they are allowed to float within a certain range according to the rules described below. Our experiments with SPEC95 benchmarks showed that it is extremely difficult, if not impossible,

to set the static values of T1 and T2 that would provide a reasonable power–performance trade–off even for the majority of the benchmarks. This is fundamentally because the difference of the IPCs and dispatch rates varies considerably among the SPEC benchmarks due to the varying branch prediction accuracies and frequencies of branch instructions. Another reason to keep the thresholds adaptive is to avoid the following two disastrous scenarios:

a) Suppose an application with a high IPC runs with the dynamic IPC of 3, thus commanding the MDR of 4. Further suppose that the threshold values are statically set like this: T1=1.7, T2=2.7, d=0.1. Then, the absence of the ILP in a small piece of code results in a momentarily drop of dynamic IPC of this application from 3.0 to, say, 1.4. The MDR selection algorithm described so far will then set the MDR for the next period P to 2, because threshold T1 has been crossed. The value of transition IPC would be recorded as 1.4. Then, suppose, the ILP picks up and the IPC goes up to 1.6 and cannot exceed that because of the limitation on the dispatch rate, branch prediction accuracy and frequency of branches. Lets assume that 1.6 is the maximum IPC that can be sustained for this benchmark for the dispatch rate of 2, just like 3.0 was the maximum sustained IPC for the dispatch rate of 4. Of course, the application is then doomed to stay in the zone with the MDR of 2, resulting in a tremendous IPC drop compared to its potential. All of this happens because the threshold T1 was set too high. At first glance, the easiest solution would be to simply use the lower value of T1. But then, scenario b) may occur:

b) Let us suppose that the smaller value of T1 (1.4) is used to cope with the above sequence of events. An application can experience the IPC drop from 3.0 to the value just above T1+d (1.6, for example). The IPC of this application will then stay at 1.6 and the dispatch rate will stay at 3, potentially wasting a possibility to try a lower MDR of 2 for power savings. Of course, this is not as bad as the scenario a), but considerable power savings potential can be neglected in this case, which is undesirable.

Therefore, the following mechanisms are proposed to cope with the above situations. To prevent the application from being permanently stuck at MDR of 2, we decrement T1 by d if MDR stays at 2 for K1 (10 in our current experiments) periods P in a row. Similarly, T2 is decreased by d if MDR stays at 3 for K2 (10 currently) periods P in a row. By the same token, it is desirable to allow the application to test the waters of the lower MDR even if current threshold values do not allow this – consequently we increment T1 by d if MDR stays above 2 for K3 (currently 5) periods P in a row. Also, T2 is incremented by d if MDR stays at 4 for K4 (currently 5) periods P in a row. The initial values of T1 and T2 were set as 1.6 and 2.5 respectively for all experiments presented in this paper. T1 was allowed to move in the region between 0.5 and 1.9, and T2 boundaries were set as 2.1 and 2.5. The upper bound for T2 was deliberately set low for performance reasons.

Since the P periods are relatively long (32K cycles in our current experiments) and the value of d is fairly small (the value of 0.1 was used in all experiments described in the results section), such an optimization does not bring any instability into the system, but rather allows applications to test different MDR zones. In any case, the wrong decisions would be undone in a fast fashion – almost always in the next P period.

Additional tuning can be done by dynamically adapting the update periods and possibly the overflow thresholds. This is, however, beyond the scope of this paper.

4. Simulation methodology

The widely–used Simplescalar simulator [BA 97] was significantly modified to implement *true hardware level, cycle–by–cycle* simulation models for such datapath components as dispatch buffers, reorder buffers, load–store buffers, register files, forwarding interconnections and dedicated transfer links. The Simplescalar simulator lumps ROBs, Physical Register Files (PRFs) and the DB together into RUUs, making it impossible to directly assess switching activities within these components independently and to make an independent resizing decisions for each structure. It also assumes a single stage that performs instruction decoding, dispatching and register renaming. Such feats are difficult to accomplish entirely within a single, realistic pipeline stage. The front end of the Simplescalar pipeline was therefore modified to perform these steps over two pipeline stages.

For estimating the energy/power for the key datapath components, the transition counts and event sequences gleaned from the multithreaded simulator were used, along with the energy dissipations for each type of event, as measured from the actual VLSI layouts using SPICE. CMOS layouts for the DB, PRF, architectural register files and ROB in a 0.5 micron 4 metal layer CMOS process (HPCMOS–14TB) were used for the key datapath components to get an accurate idea of the energy dissipations for each type of transition. The results for the 0.5 micron layouts are quite representative, although greatly scaled down compared to what one would see with 0.18 micron implementations running at a faster clock rate. The exception to this are the wire dissipations *outside* these components. Dissipations on such wires at small feature sizes become relatively dominant.

The register files that implement the ROB and DB were carefully designed to optimize the dimensions and allow the use of a 300 MHz clock. A Vdd of 3.3 volts is assumed for all the measurements (The value of the clock rate was determined by the cache layouts for a two–stage pipelined cache in the same technology.) In particular, these register files feature differential sensing, limited bit line driving and pulsed word line driving to save power. Augmentations to the register file structures for the DB were also fully implemented; a pull-down comparator was used for associative data forwarding to entries within the DB and the device sizes were carefully optimized to strike a balance between the response speed and the energy dissipations. For each energy dissipating event,

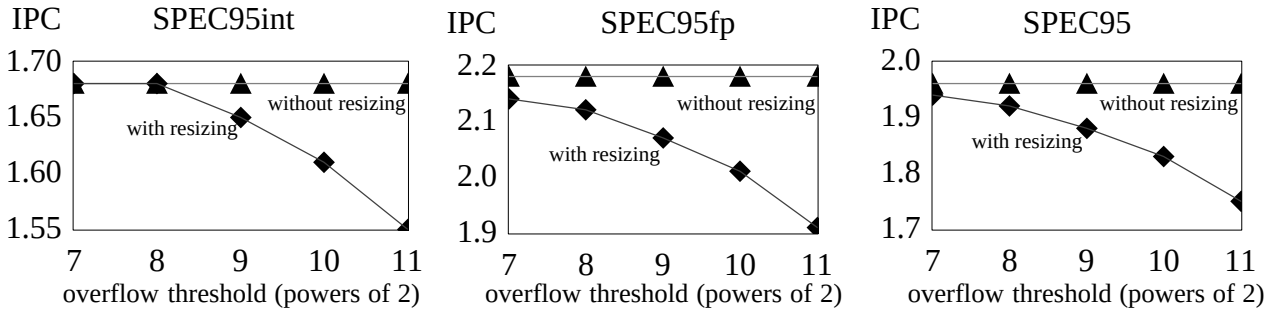


Figure 7a. IPC as a function of overflow thresholds for FDR scheme

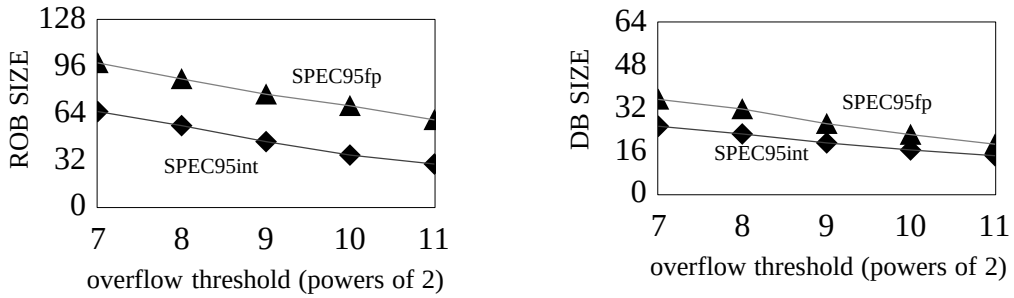


Figure 7b. Average ROB and IDB sizes as functions of overflow thresholds for FDR scheme

SPICE measurements were used to determine the dissipated energy. These measurements were used in conjunction with the transitions counted by the hardware–level, cycle–by–cycle simulator to estimate energy/power accurately.

Our methodology in computing the energy dissipations within these datapath components is to look at the traffic on each of the internal buses for the DB and the ROB and the traffic directed through register file ports and use these traffic measures to quantify the resulting energy requirements.

For the traffic directed to the DB from the FUs, we also record the number of DB entries that match the tag value floated on the result buses to estimate energy dissipations in the tag matching process . A precharged comparator, similar to what has been described in [PJS 96] is used; mismatches cause energy dissipation by discharging the match lines.

The configuration of the system studied was as follows. The L1 I–cache and L1 D–cache were both 64 KBytes in capacity with a line size of 32 Bytes, with the former being direct–mapped and the latter being 4–way set–associative. A 4–way set–associative, integrated L2 cache with a capacity of 512 KBytes and a line size of 64 Bytes was assumed. The maximum sizes of the dispatch buffer and the reorder buffer were kept at 64 entries and 128 entries respectively. The physical register file for integers and floats were 128 in number each. The function units are as follows: 4 integer units, one integer multiply/divide unit, 4 floating point multiply–add units, one floating point multiply/divide unit one load unit and one store unit. The following latencies of basic operations were assumed: integer addition – 1 cycle, floating point addition – 2 cycles, multiplication – 4 cycles, division – 12 cycles, load operation – at least 2 cycles excluding address computation. A 4–way dispatch, a 4–way commitment and a 6–way issue were assumed. For all of the SPEC 95 benchmarks, the results from the simulation of the first 200 million instructions were discarded and the results from the execution of the following 200 million instructions were used. Specified optimization levels and reference inputs were used for all the simulated benchmarks.

5. Trends and results

In the following discussion, we refer to the resizing strategy with fixed dispatch rate as FDR (Fixed Dispatch Rate) and we refer to the resizing strategy with adaptive dispatch rate as ADR (Adaptive Dispatch Rate).

Clearly, the values of overflow thresholds and update periods, and more importantly, their ratios, define the power–performance characteristics of the adaptive datapath discussed in this paper. Figure 7 shows IPCs, and the average sizes of the DB and the ROB for various values of overflow thresholds. Similar results are shown in Table 1.

The update periods of 2K cycles were chosen and the dispatch rate was kept constant at 4 for these experiments. As expected, lower overflow thresholds result in higher IPCs and larger average sizes of the structures, which directly translates into higher power dissipation.

	Overflow Threshold	128	512	2048
FDR	DB size drop %	52	64	74
	ROB size drop %	35	51	64
	IPC drop %	0.7	3.5	10.0
ADR	DB size drop %	59	70	78
	ROB size drop %	44	58	70
	IPC drop %	5.0	8.5	15.6

Table 1. Effects of various overflow threshold values on FDR and ADR (update period is fixed at 2K cycles). Results are shown for the averages across all SPEC95 benchmarks.

We experimented with the values of overflow thresholds and various update periods, and found that the best trade–off between power (average sizes of the structures) and performance (IPC value) was achieved when the ratio of update period to overflow threshold was 4. (As discussed above, we limited the values of thresholds and update periods to powers of 2 – this is to simplify the hardware implementation). In this case, the overflow threshold is reached if the rate at which blocks at dispatch occur is greater than one block in four cycles. This provides a reasonable balance between the reduction in power dissipation and performance loss (which we wanted to keep below 10%). Results in the Figure 8 are for the system with overflow thresholds of 512, update periods of 2K cycles, the period P (period of dispatch rate update) of 32K cycles and sampling period of 16 cycles for both the DB and the ROB. We deliberately chose to keep the update period of structure sizes smaller than the update period of dispatch rate to allow the sizes to change and stabilize after the dispatch rate is changed. All results in Figures 7 and 8 are shown for the non–aggressive resizing strategy, where the sizes can be decremented by no more than one block. We discuss the aggressive alternative later.

Graphs shown in Figure 8 compare the performance of three systems. As the base case, we take a system where no resizing is used and sizes of the DB and the ROB are always at their maximums. Then we consider the system where the DB and the ROB are resized according to the strategy described in this paper, but the maximum dispatch rate is constant at 4 dispatches/cycle (FDR). Finally, the augmented resizing strategy with adaptive dispatch rate is considered (ADR). For the FDR scheme, the average IPC drop is about 3.5% across all of the SPEC95 benchmarks. Reduction of the average DB and ROB sizes is 64% and 51% respectively. Reduction in power dissipation is 63.5% 49% for the DB and the ROB respectively.

For the ADR scheme, the drop is 8.5%. Average DB and ROB sizes are reduced further and total reduction compared to the base case is 70% and 58% respectively. Power dissipation is reduced by 69% for the DB and by 52% for the ROB. Thus, significant power savings can be realized using the dynamic resizing strategy proposed in this paper without significant degradation of IPC performance.

Another potential advantage of introducing the variable dispatch rate is the possibility of having dual–banked DB. Each of these two banks would have half as many read and write ports as the single DB in the base case. If the dispatch rate is 4 or 3, both banks are active, but if the dispatch rate is 2, one of these banks can be temporarily turned off. This may lead to further significant reduction in power dissipation. We are currently in the process of quantifying these additional advantages of variable dispatch rate. Along the similar lines, power savings can be achieved in the register file, if partitioned register files are used, similar to those proposed in our previous work [PKG 01].

Figure 9 breaks down the total execution time into three parts, each of these refers to the percentage of cycles when the corresponding MDR is used. Results are shown for the averages of all SPEC95 benchmarks.

For the system configuration of Figure 7, we also computed the percentage of time when the algorithm for resizing the structures down makes the decision to resize as opposed to making a decision to leave the sizes unchanged for different values of overflow thresholds. Results are summarized in Table 2 below. Notice that the percentages are higher for the DB, because its minimum partition size is smaller. As the overflow threshold decreases, the decision to resize down is made more often (approaching 100% as threshold becomes smaller). This is because the sizes are increased more often for the lower overflow thresholds, and thus there is more potential to resize it down.

overflow threshold	128	256	512	1024	2048
DB resizing down %	84	73	52	37	26
ROB resizing down %	55	45	37	27	17

Table 2. Effects of various overflow threshold values on resizing decisions. Results are shown for the FDR scheme for the averages across all SPEC95 benchmarks.

We also experimented with the update periods and preliminary results show that there is little difference in the structure sizes and slightly higher IPC drop for larger update periods. This is because the system with lower update periods captures the dynamic behavior and reacts to the changes in a more fine–grained manner. At the same time, we did not want to make the update period too small for two reasons. First, it will increase the overhead of instrumentation needed to support our algorithms and second, such a system could easily overreact to the momentarily events such as short spikes in pipeline activities.

We also studied the impact of employing aggressive–decrement mode. The aggressive mode saves less than 8% in the

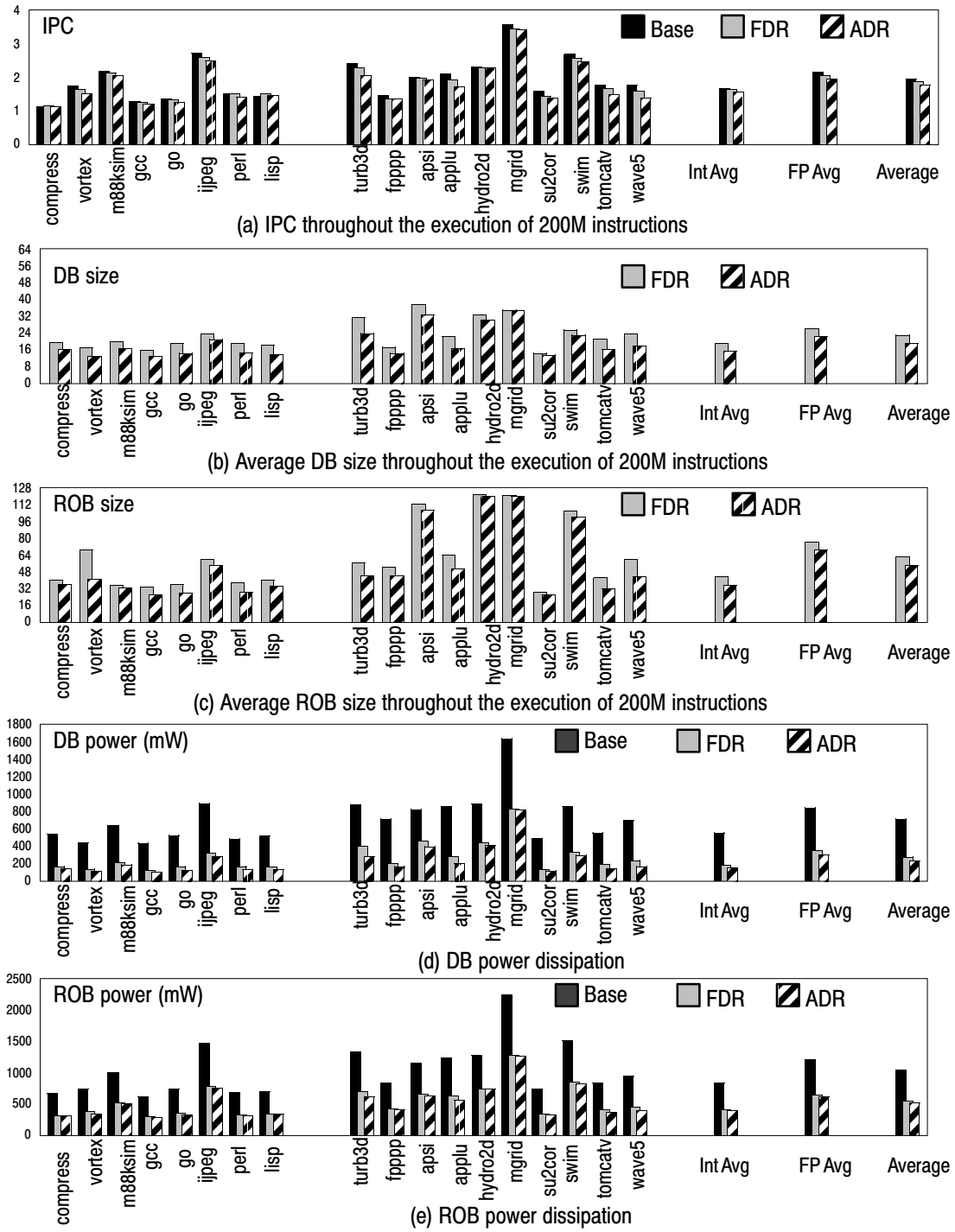


Figure 8. Effects of various resizing schemes on ROB and IDB sizes, IPCs and power dissipation
(sample period=16, overflow threshold=512, update period=2048)

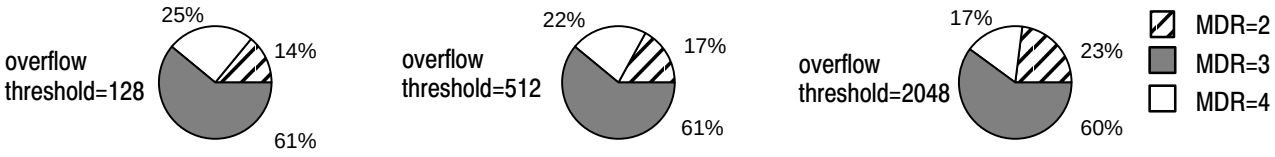


Figure 9. Distribution of maximum dispatch rates during the execution of SPEC95 benchmarks

average ROB size, and less than 2% in the average DB size. The difference is almost invisible for SPEC95int benchmarks. SPEC95fp benchmarks have some benefit from the aggressive mode with some penalty in the IPC. The IPC difference between these two modes is less than 2% in favor of non–aggressive mode.

Finally, we explored processor configuration with smaller DB and ROB sizes. We considered a dispatch buffer of 32 entries and a reorder buffer of 64 entries. Partition size was 8 entries for both structures. We observed no IPC drop (compared to the system with 64 and 128 entries, respectively) for integer benchmarks, and 9% IPC drop for floating point benchmarks on the average. The worst performing floating point benchmarks were *mgrid* (IPC drop of 21.1%), *swim* (19.4%) and *hydro2d* (18.5%). Even in this case, considerable power savings were realized using dynamic resource allocation strategies described in this paper. Specifically for the ADR scheme, power dissipation was reduced by 42% in the DB and by 38% in the ROB.

6. Conclusion & Work in Progress

We exploited the variations in usage of datapath structures to reduce power dissipation in superscalar processors through dynamic resizing of major datapath components – the dispatch buffer and the reorder buffer. Two separate phases of the resizing strategy are described in this paper. The first phase reduces the size if the sampling of the actual usage patterns indicates that the structure is overcommited. The second phase increases the size if an earlier decision to reduce the size led to an unacceptable performance degradation in the form of multiple instruction blockings at the time of instruction dispatch. The resizing strategy is then augmented by introducing the technique for dynamic control of the maximum dispatch rate based on the periodic monitoring of the actual system performance in the form of IPC. Our results indicate that the sizes of the DB and the ROB can be reduced by as much as 70% and 58% respectively with only less than 8.5% loss in performance. This translated to power savings of about 69% for the DB and 52% for the ROB.

As an extension of the work described in this paper, we are currently exploring the control and dynamic allocation of datapath resources beyond the DB and ROB (such as memory queues, register files, caches, function units). Additional work in progress is looking at the use of compiler–inserted directive in unused fields of instructions to change resource allocation dynamically – a natural transition of the hardware–directed resource allocation techniques of the current work into the compiler.

7. References

[Alb 99] Albonesi, D., “Selective cache ways: On–demand cache resource allocation”. In Proceedings of the International Symposium on Microarchitecture, November, 1999.

[BA 97] 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).

[BA+ 00] Balasubramonian, R., Albonesi, D., Buyuktosunoglu, A., and Dwarkadas, S, “Memory Hierarchy Reconfiguration for Energy and Performance in General–Purpose Processor Architectures”, *in Proceedings of MICRO–33*.

[Bh 96] Bhandarkar, D., “Alpha Implementations and Architecture – Complete Reference and Guide”, Digital Press, 1996.

[BM 01] Bahar, I., Manne, S., “Power and Energy Reduction Via Pipeline Balancing”, to appear in Int’l Symposium on Computer Architecture, 2001.

[BAS+ 01] Buyuktosunoglu, A, Albonesi, D., Schuster, S., Brooks, D., Bose, P., Cook, P., “A Circuit Level Implementation of an Adaptive Issue Queue for Power–Aware Microprocessors”, in Proc. of Great Lakes Symp. on VLSI Design, 2001.

[FG 01] Folegnani, D., Gonzalez, A., “Energy–Effective Issue Logic”, to appear in Int’l Symp. on Computer Architecture, 2001.

[GCG 00] Ghiasi, S., Casmira, J., and Grunwald, D., “ Using IPC variation in workloads with externally specified rates to reduce power consumption”. In Workshop on Complexity–Effective Design, June 2000.

[GK 99] Ghose, K., Kamble, M., “Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit–line Segmentation”, in Proceedings of ISLPED’99, pp.70–75.

[KGPK 01] Kucuk, G., Ghose, K., Ponomarev, D., Kogge, P., “Efficient Instruction Dispatch Buffer Design for Superscalar Processors”, to appear in ISLPED’01.

[Mi 9X] Microprocessor Report, various issues, 1996–1999.

[PJS 96] Palacharla, S., Jouppi, N. P. and Smith, J.E., “Quantifying the complexity of superscalar processors”, Technical report CS–TR–96–1308, Dept. of CS, Univ. of Wisconsin, 1996.

[PKG 01] Ponomarev, D., Kucuk, G., Ghose, K., “Partitioning and Allocating Register Files for Low Power”, submitted to ICCD’01.

[Wall 91] Wall, D.W., “Limits on Instruction Level Parallelism”. In Proceedings of ASPLOS, November, 1991.

[WM 99] Wilcox, K., Manne, S., “Alpha processors: A History of Power Issues and a Look to the Future”. In Cool–Chips Tutorial, November, 1999.

[ZK 00] Zyuban, V. and Kogge, P., “Optimization of High–Performance Superscalar Architectures for Energy Efficiency”, in Proc. of Int’l Symposium on Low–Power Electronics and Design, 2000, pp. 84–89.