Increased Scalability and Power Efficiency by Using Multiple Speed Pipelines[§]

Emil Talpes and Diana Marculescu
Department of Computer and Electrical Engineering,
Carnegie Mellon University, Pittsburgh, PA, 15217
{etalpes, dianam}@ece.cmu.edu

Abstract

One of the most important problems faced by microarchitecture designers is the poor scalability of some of the current solutions with increased clock frequencies and wider pipelines. As several studies show, internal processor structures scale differently with decreasing device sizes. While in some cases the access latency is determined by the speed of the logic circuitry, for others it is dominated by the interconnect delay. Furthermore, while some stages can be super-pipelined with relatively small performance loss, others must be kept atomic.

This paper proposes a possible solution to this problem, avoiding the traditional trade-off between parallelism and clock speed. First, allowing instructions to enter and leave the Issue Window in an asynchronously manner enables faster speeds in the front-end at the expense of small synchronization latencies. Second, using an Execution Cache for storing instructions that are already scheduled allows for bypassing the issue circuitry and thus clocking the execution core at higher frequencies. Combined, these two mechanisms result in a 50% to 60% performance increase for our test microarchitecture, without requiring a completely new scheduling mechanism. Furthermore, the proposed microarchitecture requires significantly less energy, with 30% reduction in a 0.13um or 20% in a 0.06um process technology over the original baseline.

1. Introduction

For the last few years, a major point of contention among microprocessor designers has been deciding the best way to increase performance in the presence of the undesirable effects of increased complexity. Some argue that complex implementations that attempt to maximize the number of instructions issued per cycle are the most desirable, since they usually rely on relatively short pipelines and are less affected by the increasing gap in memory speed. The other camp argues that a better approach is to keep the implementation complexity low and sacrifice some of the parallelism in order to maximize the clock frequency. Such processors rely on much longer pipelines, are more affected by poor branch predictability and are dependent on highly optimized memory hierarchies. However, they are better suited for highly sequential code and should theoretically scale better with newer process technologies and smaller feature sizes.

As clock speed increases and pipelines grow wider, the overhead for dynamically scheduling instructions becomes significant both in terms of latency and power consumption. In addition, as pipelines grow longer to sustain increasing clock frequencies, internal latencies may also increase. This trend has been uncovered by several previous studies. While Hrishikesh *et al.* [1] show that clock speed could still be improved by further reducing the logic depth per stage, they also acknowledge that simply superpipelining the main structures of an out-of-order processor will not improve overall performance. The issue logic is singled out as one of the structures that will ultimately impose limitations on the achievable clock speeds.

Clock speed improvement is not the only aspect limited by a monolithic, single clock, issue queue. Palacharla *et al.* [2] studied the effect of increasing pipeline width on the complexity of a superscalar, out-of-order processor. Specifically, their study assumes that future designs need to support an increasing number of in-flight instructions and larger issue widths. While latencies of most internal structures increase linearly with the issue width, the complexity of the wakeup logic grows quadratically with the width of the pipeline. Thus, it is concluded that designing single-cycle structures for the issue logic is likely to become problematic for future implementations.

Since other pipeline structures are likely to scale better with future process technologies and increasing parallelism, methods for circumventing this limitation will become more and more important. Ideally, all pipeline stages would need to be sped-up by the same ratio, either by using fast transistors or by superpipelining. However, ideal scaling cannot be achieved by using plain superpipelining for the issue logic, without losing the capability of back-to-back scheduling. To overcome this problem, we introduce a special microarchitecture that can take advantage of the modules that can be clocked faster, even in the presence of a large and slow Issue Window.

This paper proposes two mechanisms that allow both the front-end and the execution core to run at their own nominal speed. Working together, they allow future growth in performance while also providing significant energy savings.

- **Dual Clock Issue Window**. A Dual Clock Issue Window design allows the front-end of the pipeline to run at a different clock speed, asynchronously from the execution back-end. Assuming that the front-end can be clocked faster,

§ This work was supported in part by SRC Grants No. 2001-HJ-898 and 2004-HJ-1189.

this would allow a large (and relatively slow Issue Window) to uncover more Instruction Level Parallelism.

- **Pre-Scheduled Execution**. Trying to keep the execution as much as possible inside the faster back-end of the pipeline, we propose to use an **Execution Cache** placed after the Issue Window. Instructions that are fetched, decoded and have had their registers renamed are sent to execution while being stored in *issue-order* (instead of *program-order*) in the Execution Cache. Most of the time instructions are executed out of this cache and fed directly to the execution engine. Assuming the execution core can be clocked faster than the front-end (which in this case contains the relatively slow Issue Window), the overall performance can be improved. At the same time, this mechanism promises better power efficiency since it allows a significant part of the work to be reused when re-executing instruction sequences.

Such a cache has already been proposed [3] as an efficient mechanism for reducing the dynamic power in a superscalar, out-of-order processor, but its potential for improving the overall performance has not yet been investigated. In our setup, the Execution Cache is used to effectively create a new and faster execution path. To enable this operating mode, we use a new **Register File** and **Register Renaming** mechanism to alleviate the capacity constraints of previous solutions. While both the Dual Clock Issue Window and the Execution Cache have been described before, they have not been used together to create an efficient, high performance microarchitecture. Furthermore, the resulting *Flywheel* microarchitecture is more scalable than its superscalar, out-of-order counterpart for deep sub-micron technologies as it does not rely on a single clock design methodology.

The adaptation mechanism supported by our microarchitecture is similar to a mechanical engineering concept: the *flywheel*. After applying an external stimulus, the flywheel continues spinning because of its large inertia. In our design, the front-end of the pipeline is only used for creating traces that will be later re-executed on the back-end alone. When the front-end is decoupled, the back-end continues executing traces already stored in the EC as long as it hits there. This holds due to high locality of these traces (the equivalent of inertia in mechanical terms) and progresses until locality is lost and the front-end must be restarted[1].

This paper is organized as follows: in Section 2 we present the motivation behind our approach, while Section 3 describes the main details of our microarchitectural design, including the pipeline organization, the new issue queue implementation, the Execution Cache and the new Register File. The experimental setup is described in Section 4 and results on several Spec95 and Spec 2000 benchmarks are included in Section 5. We present a summary of the previous work that is relevant to our approach in Section 6 and conclude the paper in Section 7.

[1] As it can be seen, the analogy is not perfect by equating speeds in mechanical and hardware sense, in that the flywheel gradually slows down after the external stimulus is removed, whereas in our design the execution speed is actually increased after the slow, front-end circuitry is decoupled.

2. Motivation

As feature size decreases and transistors become faster and faster, the relative importance of the time spent driving signals on long metal lines grows. Thus, signal paths that are dominated by transistor speed are expected to scale significantly better in future process technologies than wire-dominated ones. In [2], Palacharla *et al.* look at how different structures scale with faster logic and increasing wire importance. The study found the wake-up and bypass networks to be the two structures that scale worst with decreasing technology sizes. While the latency of the bypass network increases quadratically with the number of functional units, the latency of the wake-up circuitry depends quadratically on the Issue Window size.

By extending the models presented in [2] and [4], we can compare the relative latencies of different modules in future process technologies. In Figure 1, we can see that a reasonably sized cache is about two times slower than the Issue Window in 0.25um or 0.18um process technologies, but it scales much better achieving about the same access time as the 128-entry Issue Window in 0.06um[2].

The ability to perform back-to-back scheduling is so important for obtaining high performance that most

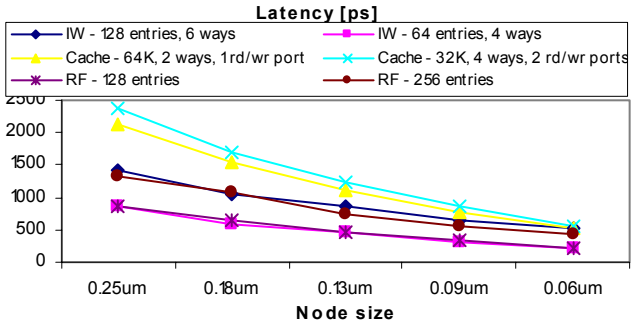

Fig. 1. Latency scaling for Issue Windows (IW), caches and register files (RF).

microarchitects assume it as a requirement. The scheduling logic performs two basic operations: **Wake-Up** and **Select**. During the Wake-Up phase, instructions in the Issue Window check their dependencies and, if these are resolved, they request access to a functional unit for execution. During the second phase, some of these instructions are selected and scheduled. To be able to schedule dependent instructions in the next clock cycle, the output of a previous Select operation must be available when Wake-Up starts. Thus, these two operations must be executed together during a single cycle.

The importance of performing these operations in a single cycle has been shown before [1]. Three operational loops are

[2] Even though some authors use 0.065um as the next process technology after 0.09um, we use the numbers provided in [4] and thus we assume a 0.06um process technology.

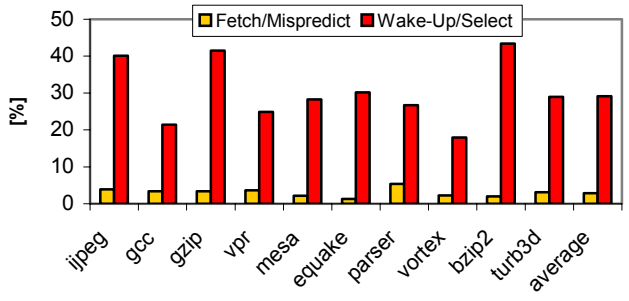

Fig. 2. IPC degradation when adding one extra stage to the front-end of the pipeline (Fetch/Mispredict loop) and to the Issue Window (Wake-up/Select loop).

studied (Load/Use, Branch/Mispredict, Wake-Up/Select) and Wake-Up/Select is singled out as the most critical. We show in Figure 2 the impact of further pipelining the issue logic for a four-way superscalar, out-of-order microprocessor based on a nine-stage pipeline, with seven cycles mispredict penalty (dark bar). We also show, as a comparison, the impact of adding one stage to the front-end (light bar).While the performance hit of the additional front-end stage is limited (less than 3% on average), it can increase to more than 40% when the Wake-Up/Select loop is pipelined (slightly less than 30% performance loss on average). As a large Issue Window is likely to become the bottleneck for future frequency growth in a single-clock processor, our design tries to build on the multiple clock methodology, limiting the performance degradation induced by the slow Wake-Up/Select logic.

3. Design exploration framework

In this paper, we propose an alternative solution to the traditional tradeoff between parallelism and clock speed. As the rest of the pipeline is likely to scale better than the Issue Window in future process technologies, we assume that it will be able to work at higher clock speeds. Thus, instead of trying to make the Issue Queue faster, our proposed approach limits its impact on the overall microarchitecture performance. The solution relies on using multiple clock domains, allowing different segments of the pipeline to work at different speeds.

The front-end of the pipeline is decoupled through the use of a Dual Clock Issue Window. In this setup, instructions can be brought in the Issue Window faster, exposing more parallelism. This mechanism can significantly improve performance in cases where many independent instructions can be found; for code with poor instruction level parallelism (ILP) the only way to improve performance is to execute each instruction faster. Even though the execution core might scale, we cannot directly increase the clock speed of the ALUs as the cycle time may be dictated by other slower structures (namely, the Issue Window). Should the ALUs be clocked faster, instructions would need to be scheduled at this increased speed, and this is not possible due to the slow issue logic.

In order to allow the execution core to work at a faster clock speed, the back-end is decoupled by the use of an Execution Cache (EC). Assuming that instructions are retrieved from the Execution Cache most of the time, they can be sent directly to the execution core allowing it to work at its own maximum frequency. The resulting design will thus have two operational modes: **trace creation mode** and **trace execution mode**. In the first mode, instructions are read from the I-Cache and fed through the normal front-end of the pipeline. The execution core is synchronous with the slow Issue Window, and thus it has to operate at the baseline frequency. However, the front-end will function faster, enabled by the Dual Clock Issue Window that allows dispatching and issuing instructions asynchronously one to each other.

After each branch misprediction, the EC is searched for a trace starting at the current Program Counter (PC) location. Should such a trace be found, the processor switches to trace execution mode. The front-end is completely shut down, including the scheduling logic. Instructions are read from the EC and fed directly to the functional units. In this setup, the resulting critical path can be clocked faster. For the front-end of the pipeline, we can use a dedicated PLL or ring oscillator to generate the maximum clock frequency supported by these stages. However, in order to avoid using two separate circuits for generating the back-end clock signal, we can generate them both starting from a unique (and fast) clock signal. By dividing this signal by different constants, we can generate both the slow clock signal required in trace creation mode, as well as the fast clock signal used in trace execution mode. While this method prohibits the use of many possible clock speeds combinations, it only requires a negligible overhead for switching between the two operating modes. When running in the trace-execution mode we can either let the faster clock go into the Issue logic and use clock gating, or we can additionally use power gating for additional power savings [8]. In our tests, we are using clock gating of the Issue Window when not in use (and thus, our results are conservative as power gating may provide additional power savings).

3.1. Pipeline organization

To this end, we start with a nine-stage pipeline that implements a four-way superscalar, out-of-order processor. Our baseline microarchitecture is able to fetch up to four aligned instructions and move them in lockstep towards a monolithic **Issue Queue** (similar to the one used by MIPS R10000 [6]). Here, they are reordered according to their data dependencies and ready instructions are sent in parallel to the out-of-order execution core. When an instruction is selected, its tag is broadcast to all entries in the Issue Window (for Wake-up) and to the Register Availability Table (RAT) [7].

The proposed microarchitecture is presented in Figure 3. In this setup, the pipeline is effectively broken into two separate clock domains. In the first domain, instructions are fetched from the I-cache, decoded and their registers are renamed. The second domain contains the Issue Window and the Reorder Buffer, the Register File (Read and Write Back stages) and the functional units. As independent instructions are retrieved

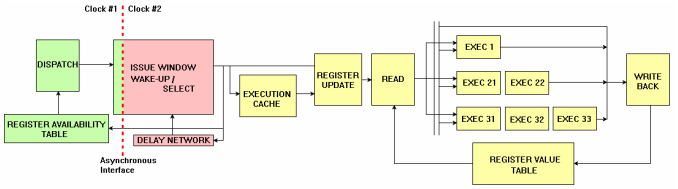

Fig. 3. Proposed modifications for the Issue Mechanism.

from the Issue Window and sent towards the execution core, they are, in parallel, added to the Execution Cache creating program traces. The EC can also be a provider of instructions. After a branch misprediction, the next PC address is compared against existing traces and, should a match occur, instructions will be retrieved directly from EC.

The last modification that we bring to the typical superscalar, out-of-order processor is the Rename mechanism. As instructions come from the EC in execution order, typical register renaming can not be performed and we have to rely on information stored during trace creation. Thus, in order to reuse our traces, we have to make sure that the mapping between architected and physical registers is preserved whenever a trace is restarted. To solve this problem, we split the renaming mechanism in two sub-stages: a normal, **Rename** stage that allocates physical entries for each architected registers and a second, **Update** stage that remaps these entries when instructions are re-executed.

3.2. Fine grain dynamic voltage scaling

Using an altered Instruction Window, we attempt to increase the overall performance of the microprocessor by increasing the clock speed used in the front-end. Thus, the design needs to be slightly modified in order to provide synchronization capabilities for data coming from one domain and requests coming from the other.

Most of the asynchronous communication takes place after the Dispatch stage, as instructions must be synchronized with the back-end clock signal before being considered for Issue. Other synchronization points exist as well: fetch redirect requests go from Write Back to the Fetch stage, branch predictor updates are sent from Retire to the Fetch stage and physical register releases are triggered in Retire, but must be operated in the Rename Table. In all these cases, we assume a FIFO-based communication with the same synchronization latency as proposed before [9][10].

The synchronization circuitry needed by the Issue Window is similar to the one proposed in [11]. Essentially, instructions are placed in unused entries synchronously with the producer clock, where they wait a certain synchronization time before being seen by the Wake-up/Select circuitry that operates synchronous with the consumer clock. After this delay is observed, Wake-up and Select can proceed in the same manner as in the synchronous counterpart.

An additional problem appears though when tags are forwarded after selection. As the tag matching circuitry from Wake-up is placed in the same clock domain, it is reached earlier than the Register Availability Table (RAT) (Figure 3).

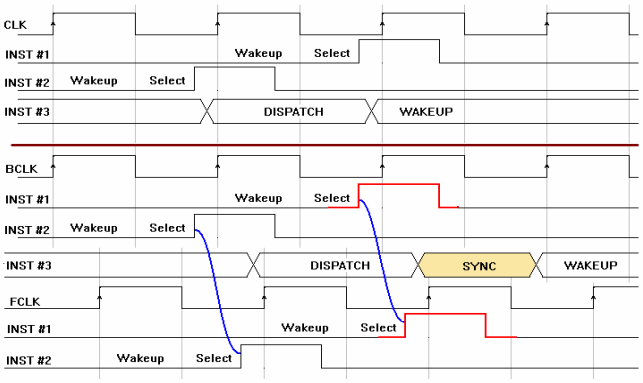

Fig. 4. Timing behavior for a synchronous Issue Window (top) vs. a Dual Clock Issue Window (bottom).

Thus, it becomes possible for an instruction to pass the Dispatch stage without seeing the broadcast tag, and also miss it before reaching the tag match circuitry from Wake-up.

In Figure 4, instruction *INST #3* depends on the results generated by *INST #1* and *INST #2*. In the fully synchronous case (top of Figure 4) the tags can arrive either early, before the instruction reaches the Issue Window (*INST #2*) being sampled in RAT, or later, and in that case they are seen during the first Wake-Up cycle (*INST #1*). However, in the asynchronous case (bottom of Figure 4), the synchronization delay (*SYNC* in Figure 4, lower signals) creates another possible scenario: *INST #1* can generate its tags late enough that they reach RAT after *INST #3* leaves Dispatch, but too early to be seen during the first Wake-Up cycle.

This problem has been acknowledged before [11], and two possible solutions have been proposed for it: delay the tag matching of the Wake-up until the broadcast data is seen in the other clock domain (Delay Network in Figure 3) or memorize this tag for a few cycles and duplicate the tag matching lines. While the implementation is simple in the first case, it loses the exact same capability that we intended to preserve in the first place – back to back scheduling. In the second case, the performance penalty is negligible, but the complexity of the Issue Window increases (Figure 5). In our design, the back-end clock includes the Wake-Up/Select logic and thus will always be slower than the one used by the front-end. This limits the extent of the problem, so in this case we only need to perform wake-up using the tags forwarded in the previous two cycles.

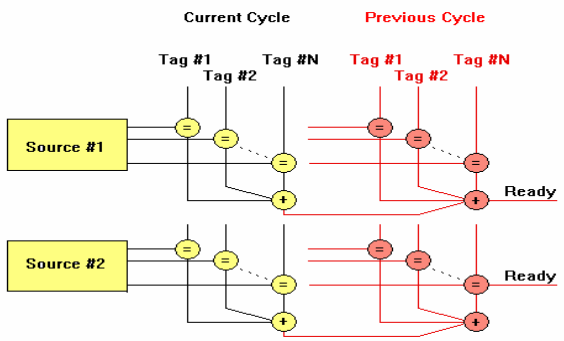

Fig. 5. Duplicated tag matching for the Dual Clock Issue Window.

3.3. Pre-scheduled execution

In order to take the Wake-Up/Select logic off the execution path, we propose that an Execution Cache be placed deep in the pipeline, after the Issue Stage (as illustrated in Figure 6). In this setting, the branch misprediction path can be significantly shortened by feeding the execution units directly from the EC whenever possible. Initially, when EC is empty, instructions are launched from the Issue Window, while a trace is built in parallel. This step is called the *trace segment build* phase. Upon a mispredict (or a trace completion condition), the EC is searched and, should a hit occur, instructions start to be sent on the alternative execution path (*the trace execution phase*). When operating on this alternative execution path, the processor behaves like a VLIW core, with instructions being fetched from the EC and sent directly to the execution engine. If a miss is encountered on a trace search, the pipeline front-end is restarted and a new trace is built.

Similar to conventional trace-cache implementations, the proposed design divides the program into traces of instructions that are stored in a different order than the one given by their original addresses. This allows for implicitly encoding information about the reordering work done in the Fetch and Issue stages through the actual order in which instructions are stored. However, the cache chosen in the proposed architecture is structurally different from the trace-cache typically used for increasing the fetch bandwidth.

When stored in issue order, instructions lose their original, logical order and can be retrieved only on a sequential basis. Only the start address of each trace needs to correspond to a physical address in the memory space. Instructions from two consecutive traces cannot be interleaved, so with each change of trace the processor must restart execution in-order. Furthermore, after each trace, a *trace look-up* step must be performed. While most of the time the penalty associated with this look-up can be hidden (the look-up being started in advance), there are certain conditions when this is not possible. Together with the need for in-order start, this leads to some performance penalty associated with each trace change.

To minimize the performance impact of this design, traces must be created as long as possible. While most trace cache designs proposed in the literature limit the traces to at most three basic blocks, it is desirable to include as many instructions as possible if no misprediction is encountered. The simultaneous presence of traces with different lengths (some of them very long) in the EC has precluded the use of a standard trace cache model [21][22] or the very efficient block

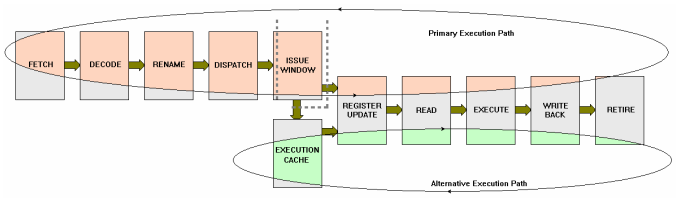

Fig. 6. Superscalar microarchitecture using an EC for reusing scheduled instruction streams.

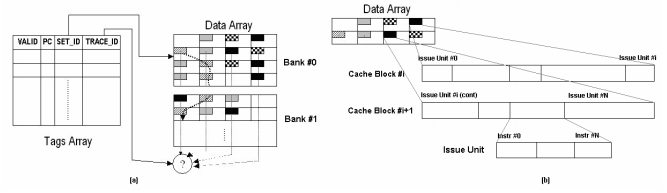

Fig. 7. Execution Cache structure (a) and block organization (b)

based trace cache structure [23]. Thus, *Flywheel* relies on a solution that resembles the Pentium 4 implementation [25] to accommodate traces of arbitrary length. The architecture of this solution is presented in Figure 7a.

The EC structure consists of a ***Tag Array*** (TA) and a corresponding ***Data Array*** (DA). The TA is an associative cache, addressed using the translated program counter. It is used for trace look-up and it should be as fast as possible to reduce the performance overhead associated with searching for a new trace. The *SET_ID* value obtained from the TA points to the DA set containing the first instructions of the trace. DA is a multi-way set associative cache composed of multiple memory banks. A comparison with *TRACE_ID* is performed for each block in the set, identifying the correct starting point of the trace. The next chunk of instructions is located in one of the blocks of the following set, and so on (see Figure 7a). A special end-of-trace marker identifies the end of the trace. By knowing beforehand which set will be accessed next, we avoid performing a new look-up for every access. Knowing the next set also allows the use of multiple memory banks to implement the DA. While one of the banks is used, the others can be turned off, resulting in further energy savings.

Inside each block, an arbitrary number of Issue Units are stored (Figure 7b). An Issue Unit consists of independent instructions that can be issued in parallel to the functional units. Since Issue Units are recorded during the trace building phase and then reused, the processor will make the same optimizing decisions each time it executes the code. Each block accommodates multiple Issue Units, causing a significant reduction in the total number of accesses. All instructions coming from the Issue Window are first assembled into traces using a Fill Buffer and then recorded in the EC. The Fill Buffer can accommodate two DA blocks; when enough instructions are available to fill a block, they are written to the EC. When reading from the EC, we issue one Issue Unit at a time, using a similar mechanism. If enough space is available in the Fill Buffer, a block is read from the DA and added to the buffer.

As each block contains more than one Issue Unit, it needs more than one clock cycle to be sent to the execution core. This organization allows for hiding a large part of the EC latency, an access being initiated before we are forced to stall the pipeline. While the longer cache lines increases the power required for each access, they also help reducing the total number of accesses. One complete block is read during each access, and instructions are stored in the circular Fill Buffer. Since the buffer can accommodate two DA blocks, a second access can be started immediately. When the new set of

instructions arrives (several cycles later), they are added to the Fill Buffer. Thus, even if an Issue Unit spans across two DA blocks it will still be issued in a single cycle most of the time. Corner cases can still occur though, when the Issue Units are very large. In such cases the second block might arrive late, the Issue Unit being split over two or more execution cycles. The downside of increasing the block size is that the efficiency of the EC decreases, many instruction slots remain empty. Our tests have shown that smaller blocks can be significantly more efficient in terms of instruction space storage, whereas very small blocks tend to have negative influence on the overall performance. For our evaluation, we have focused on eight-instruction blocks that are usually able to accommodate three or more Issue Units.

3.4. Associative register file

To enable pre-scheduled execution, we need a special Register File, along with a new Register Renaming mechanism. As instructions come out of the EC without preserving their original program order, registers cannot be renamed for such trace replays. At the same time, operand values cannot be simply stored in the EC and reused, as they are generally different during each trace run. Since the processor must rely on the renaming information collected during the *trace build* phase, a different registers pool and control unit has to be used. To handle this task, a special register file structure is used [3].

This structure employs a special pool of physical registers for renaming every logical register of the ISA. Unlike in a typical scheme where an architected register can be mapped onto any physical register, here it can be renamed using only the entries of the corresponding pool (Figure 8). Each write goes circularly to a different physical register, solving the problem of false data dependencies. As with any rotating register file, the number of in-flight instructions that have the same logical destination is bounded by the number of physical locations available in the circular buffer.

When going through the Rename stage, each instruction is allocated a physical register as destination. This entry must be different than the one holding the last known value for the corresponding architected register. Having different physical destinations, instructions can write their result as soon as it becomes available, setting the *V* (valid) bit. The *S* (speculated) bits are used for remembering which values have actually been

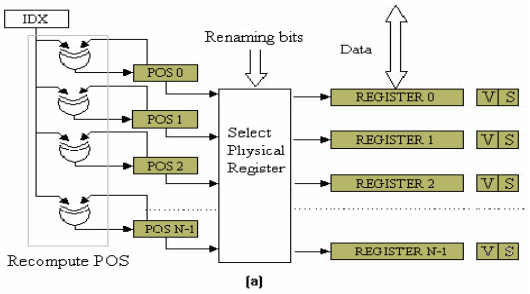

Fig. 8. Architected register structure

committed, and they are cleared only after the instruction has retired. The *N* indices (values *POS* 0 - *POS N*-1) are initialized with consecutive values (0,1,2 … *N*-1) and represent the logical order of the registers in the circular queue. *IDX* is a pointer in this queue representing the most recent register used for writing. All accesses are associative, comparing the renaming information against the *POS* tags. Inside a trace, the *POS* tags remain constant and represent the logical order of the circular queue (the order in which entries are allocated).

Each trace generation is started with an initial value of *IDX* = 0 for all architected registers, meaning that the current value of the register is stored in the location marked by *POS* = 0. If this condition is respected, all subsequent executions can be performed without further renaming the registers. The caveat is that this requires a *checkpoint* to be performed when a trace execution ends: all *POS* values must be recomputed so that the circular buffer starts with the latest written entry. This can be done by subtracting from *POS* the *IDX* value, but it would require a separate adder for each physical register. However, since the physical order of the registers is not relevant and it does not have to match the logical one, the same effect can be obtained by *XOR*-ing the *IDX* with each *POS* value. By doing so, all registers have different tags ranging between 0 and *N*-1 and the register holding the last value receives *POS* = 0.

While it solves the problem of false data dependencies, this associative register renaming scheme has several important limitations. First, an associative organization complicates the actual hardware implementation of such a mechanism. In this setup, each architected register would have to be implemented as a separate piece of logic, being accessed directly as a block and associatively inside the block.

A second drawback of this scheme is that it only offers limited rename capacity. Our tests show that a situation where the Rename stage has to stall is quite common with a limited number of physical entries inside each pool, and this problem gets worse as pipelines grow longer. The actual pool size required for eliminating most of these conditions depends on the number of in-flight instructions, and, as we target a design with a high number of such instructions, this can rapidly become an important problem.

Finally, another problem is that the associative Register File design can introduce stalls in the pipeline when a checkpoint needs to be executed. All accesses that we perform on this structure must see a coherent state, so all the control information (the *POS* fields) must be preserved until the last access is performed for the current trace. This happens in the Retire stage (for updating the *S* bits), so the checkpoint cannot be performed until the last instruction of the trace is retired. Requiring a different *POS* configuration, a new trace will not be allowed to access the register file before the checkpoint occurs at the end of the previous trace. Depending on the execution mode, this problem blocks the first instruction of the next trace either in the Register Rename stage or in the Execution Cache, introducing bubbles in the pipeline.

The effect can be reduced by allowing very long traces in the EC. In this case, the problem will mostly arise when a trace must be abandoned due to a branch misprediction. Such a condition can be detected as early as the Execute stage, so there are at least several cycles lost until the checkpoint can be performed. This problem can become even more important if the instruction is speculated and must wait additional cycles in the Reorder buffer. If it takes multiple clock cycles for a mispredicted branch to be retired after it was detected (in the Write Back stage), it is very likely that the next trace will be stalled waiting for this checkpoint to occur. A similar problem occurs in the Pentium 4 microarchitecture [28]. Keeping in flight a large number of instructions, the processor must speculate across multiple branches. Each of these branches can turn out to be mis-speculated, so the processor must have the possibility to recover after each of them. In order to avoid keeping copies of the Rename Map (named Front-End RAT) for each such branch instruction, Pentium 4 uses a second copy (Retirement RAT) that is updated in-order, at completion. Whenever a mispredict occurs and the speculative state must be rolled back, the Front-End RAT is replaced by the content of the Retirement RAT before the next instruction can pass through Register Rename. This introduces a similar penalty, between the moment when the condition is observed and the one when the offending instruction can be retired.

3.5. Direct access register file

While the mechanism described above can conceptually solve the problem of false data dependencies, its limitations make it unsuitable for a real hardware implementation. Thus, the Flywheel microarchitecture relies on a modified renaming algorithm that addresses these problems, as described next.

To avoid the associative Register File access, we must generate a physical register *ID* in both operating modes, trace creation and trace execution. For this, we must split the Register Rename operation in two separate phases, each of them occupying one pipeline stage. The first phase (Register Rename) functions essentially in the same way as the associative mechanism. It assumes that the Register File is still organized in logical clusters, and always allocates the next pool entry for a new result. The second phase (Register Update) performs a remapping between the cluster-based logical renaming space and the contiguous physical Register File. The required resources are presented in Figure 9.

The Rename Table holds essentially the same information as the *IDX* fields in Figure 8. For each architected register, this table holds the *ID* of the current entry in the logical pool.

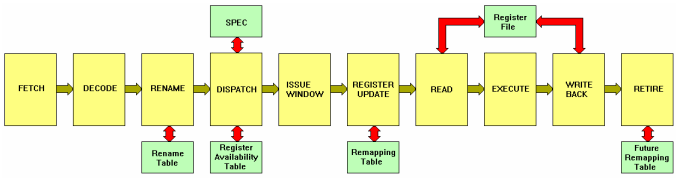

Fig. 9. Resources used for the two-phase renaming mechanism

When a source operand must be renamed, this value is read and assigned directly to the instruction. For a destination register, the value must be first incremented (specifying the next logical entry in the pool) and then stored back to the Rename Table. After passing through the first Rename stage, each architected register is assigned a logical identifier (*LID*). During the Dispatch and Issue stages, registers are only specified inside the logical (associative) renaming space. However, they can be uniquely identified using the architected register name and the *LID*. This information is used for accessing the Register Availability Table (RAT) and for keeping track of data dependencies in the Wakeup logic. The SPEC table is accessed using the same method and it holds the same information as the *S* bits in Figure 8.

The second phase of the renaming process is performed in the Register Update stage. Here, the register is remapped from the logical renaming space to the physical Register File, generating the correct register addresses. Instructions can come from either the front-end of the pipeline or from the EC; as long as they have *LID*s associated with their architected registers the process can be performed in similar fashion. Each logical pool is defined in the Remapping Table (*RT*) by its size and starting address, and the physical offset (*PO*) is obtained by combining them with the *LID*.

Further down the pipeline, the Register File can be accessed directly, using the information generated in the Register Update stage. The *PO* is also needed at completion time, for updating the Future Remapping Table (*FRT*). *FRT* is identical to the regular *RT*, holding one entry for each architected register. Similar to the Retirement *RAT* in Pentium 4 [28], the purpose of this table is to keep track of the latest non-speculative values, and it is updated with the *PO* of each destination register. Similar to the Pentium 4 microarchitecture [28] where the Retirement *RAT* is copied to the Front-End *RAT*, the *FRT* must be copied to the *RT* during the checkpoints that occur at each trace-change.

After a trace completion (e.g., a branch mispredict), the Rename Table is reset and the *LID*s start being generated from 0. When passing through the Register Update stage, these *LID*s are XORed together with the information in the *RT*, causing the source operands to point to the latest committed values for their respective architected registers.

Fitting inside this dual-stage register renaming scheme, a dynamic approach has been proposed for solving the problem of the limited renaming capacity [12]. At specified intervals, the history of the renaming constraints is evaluated and register redistribution can be performed. Specifically, architected registers that have been detected to be bottlenecks are supplemented with additional physical entries, taken from other pools that are only infrequently accessed. This scheme ensures that the processor can adapt itself to different conventions for register allocation. When registers are redistributed, a size and base address are set for each architected register. This information is stored in the Rename

Table, and used for generating the *LIDs* or the physical register address. The drawback of such a mechanism is that whenever redistributions are performed, previous renaming information becomes obsolete and all entries in the EC must be invalidated. Together with the actual register redistribution, this can require a significant number of clock cycles, so this operation should not be performed very often.

In our experiments, we assumed that the counters are tested every 500,000 cycles, and, if needed, redistribution will require an additional 100 cycles. Our results show that only a small fraction of the total architected registers need more than four physical entries (typically 10 to 15%). Furthermore, the best configuration is mostly dictated by the conventions used for register allocation, so the number of redistribution is fairly small and steady state can be rapidly reached.

This new register renaming design still maintains several limitations. First, it requires an additional pipeline stage, which adds a cycle to the mispredict penalty and will cost about 2-3% in performance. Second, it still imposes restrictions on starting a new trace. As in Pentium 4, the *FRT* must be copied to the *RT* before new instructions can pass through Register Update. This checkpoint can only be performed after the offending instruction is ready for commitment, and thus pipeline stalls can occur in certain cases. However, this can only happen if it takes more than five cycles from the moment when the misprediction is detected until the instruction can be retired.

An additional problem arises when using this design with the Execution Cache (Figure 10).

When instructions are issued from the EC, a trace can end

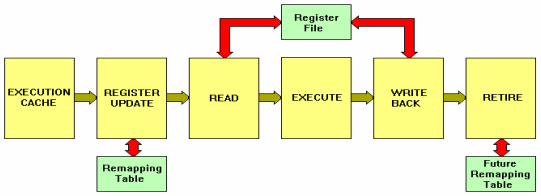

Fig. 10. Equivalent pipeline in trace execution mode

without generating a mispredict. Such a trace completion will be detected when reading instructions from the EC, and a new trace can be looked up immediately. Ideally, instructions from the new trace will be sent to execution as soon as possible, but in this design they cannot go through Register Update until the previous trace has been completely retired. As the Register Update stage is very close to the EC, most of the time the new trace will have to wait for a few cycles before it can proceed towards the execution core. On the alternative execution path, the last instruction of the trace will require at least 5 cycles to go through the pipeline. Very often, it takes significantly more than 5 cycles due to data dependencies or higher execution latency. However, the look-up for a next trace can be started early, so new instructions could be retrieved very fast.

To solve this problem, we can add a Speculative Remapping Table (*SRT*) to the Register Update Stage. As instructions pass through this stage and physical register addresses are computed, this *SRT* is updated with the same information that eventually goes to the *FRT*. After a trace-end instruction updates SRT, this table holds the exact same information that will be contained in *FRT* after the instruction is retired. Apart from the *PO* for each physical register, this table must also hold the *Trace Id* of the last instruction that modified it. As instructions arrive out-of-order, this field is needed for making sure that an older instruction does not modify the record of a newer one. For each instruction that passes through Register Update, the *SRT* entry corresponding to its destination is read together with the normal entry in the *RT*. In parallel with computing the physical register address, the *Trace Id* is compared against the *SRT* and, if newer, the *SRT* entry is updated with the new *PO*. The update will likely require an extra cycle, so the content will be up-to-date one cycle after the last instruction of the trace passes through Register Update.

By swapping this table with the regular *RT*, the execution of a new trace can be started immediately. Thus, the penalty for changing the trace is reduced to a single cycle. However, this mechanism can only be used when the end-of-trace instruction is detected before the Register Update stage. When a branch mispredict is detected in the Execute stage, the content of the *SRT* might not be coherent so it cannot be used for restarting a trace right away. In this case, the *FRT* must be used so the new trace must wait until the previous one retires completely.

4. Experimental setup

For a 128-entry Issue Window with an issue width of six instructions per cycle, a fairly balanced baseline design (in 0.18um process technology) can be obtained by using 64K L1 caches with a two-cycle access time. For the D-cache we assume a four-way associative implementation and for the I-cache a simpler, two-way associative implementation. A 192-entry Register File offers enough rename entries for this design, without introducing further clock speed restrictions. Assuming a single-cycle access for the Issue Window and Register File, and a two-cycle pipelined access for L1 caches, the clock speeds achieved by these modules are presented in Table 1[3]. While the D-cache is dual ported in order to support two load/store instructions in parallel, we only need a single read/write port for the I-cache. Thus, the front-end of the pipeline can potentially be clocked higher than the back-end. As suggested by many recent processor implementations, the functional units are unlikely to become a limitation, being scalable to frequencies significantly higher [27][28].

For the *Flywheel* microarchitecture, we sized the modules in such a way that they do not become a bottleneck themselves for future frequency increases. Thus, a 128K EC access will take three cycles, while the larger, 512-entry Register File will require two cycles for read/write. These results suggest that in future process technologies we will reach a point where the front-end of the pipeline will support twice the frequency of

[3] The access times have been computed as in [4].

TABLE 1
Clock frequency estimated for the main modules of the pipeline, in future process technologies (MHz).

Baseline	0.18um	0.13um	0.09um	0.06um
Issue Window (single cycle)	950	1,150	1,500	1,950
I-Cache (two cycles)	1,300	1,800	2,600	3,800
D-Cache (two cycles)	1,000	1,400	2,000	3,000
Register File (single cycle)	1,150	1,650	2,250	3,250
Flywheel				
Execution Cache (three cycle)	1,000	1,400	2,050	3,000
Register File (two cycles)	1,050	1,500	2,000	2,950

the Issue Window, while the Execution Core will also support a higher clock speed, but by only 50%. The clock frequencies presented in Table 1 are obtained assuming the implementation details described in [2]. Using different circuit-level implementations, the absolute frequencies can be quite different, but the scaling trends will most likely be preserved.

To measure the impact of our microarchitecture on both performance and power consumption, we have implemented a cycle-accurate simulation model of the original fully synchronous pipeline and its dual clock counterpart. Our simulator is based on Simple Scalar [13], but reflects the target pipeline more accurately. As opposed to SimpleScalar, it uses normal pipeline registers, an Instruction Windows that closely resembles the mechanism described herein and a Retire Buffer. The register renaming mechanism chosen is similar to the one used by the MIPS R10000 processor [6]. We have used the Wattch modeling framework [14] to include power models in our simulator. However, instead of assuming that unused modules are consuming 10% of their normal in-use power consumption, we developed models for the static energy leakage based on the framework described in [15] using STMicro technology.

TABLE 2
Microarchitecture parameters

Parameter	Value
Pipeline	9 stages baseline, 4 way out-of-order
Instruction Window	128 entries, issue width of 6
Register File	192 entries, single cycle for the baseline 512 entries, two-cycle access for *Flywheel*
Load / Store Queue	64 entries
I-Cache	64k, 2 way set-associative, 2 cycles hit time, LRU replacement alg.
D-Cache	64k, 4 way set-associative, 2 cycles hit time, LRU replacement alg.
L2 Cache	Unified, 512k, 4 way set-associative, LRU replacement alg.
L2 access time	10 cycles
Execution Cache	128k, 2 way set-associative, 3 cycles hit time
Memory access time	100 cycles (scaled accordingly when clock speed is increased)
Functional Units	4 Integer ALUs, 2 Integer MUL/DIV 2 Memory ports 2 FP Adders, 1 FP MUL/DIV
Branch Prediction	G-share, 12 bits history, 2048 entries
Technology (for power estimation [12])	0.13um - V_{dd} = 1.4V, V_t = 0.22V 0.09um - V_{dd} = 1.2V, V_t = 0.20V 0.13um - V_{dd} = 1.1V, V_t = 0.18V
Normalized leakage current per device	0.13um – 80 nA (normalized per device) 0.09um – 280 nA (normalized per device) 0.06um – 280 nA (normalized per device)

In addition to modeling the switching capacitance of memories and buses inside the processor, we have also modeled the switched capacitance of the global and local clock grids. We have assumed a clock distribution hierarchy resembling the one used by the Alpha 21264 processor, modeling one global clock grid and local clock grids corresponding to each of the synchronous domains. The area and metal density for each clock grid are the ones published for the Alpha 21264 processor.

The parameters for the microarchitecture under consideration are presented in Table 2. In our experiments, we have used integer and floating-point benchmarks from both SPEC95 and SPEC2000 suites. For all experiments, we have fast-forwarded over the first 500 million instructions and then continued simulation for another 100 million instructions. As our proposed microarchitecture uses different clock frequencies for each section of the pipeline, all our evaluations use the total execution time as a measure of performance.

5. Experimental results

When limited to the same clock frequency, our microarchitecture is still able to keep pace with the superscalar, out-of-order baseline. The effect of the limited renaming capacity offered by our register allocation mechanism is very significant in some cases. Together with the Dual-Clock Issue Window, the adaptive register file makes the pipeline almost three stages longer and accounts for more than 10% performance drop in several benchmarks (*gzip*, *vpr*, *parser*). However, due to the reduced mispredict penalty offered by the alternative execution path, the *Flywheel* microarchitecture is able to overcome these limitations and offer an average 5% increase in performance over the baseline. In Figure 11, all the results are normalized with respect to the fully synchronous superscalar, out-of-order baseline.

Furthermore, the proposed microarchitecture is able to utilize the alternative execution path for an average 88% of the time. Assuming this path can be sped up using a faster clock signal, this offers an important opportunity to increase the performance. While in some cases the front-end of the pipeline is still used for 40% of the time, on most benchmarks the processor spends more than 90% of the execution time fetching instructions directly from the EC.

As expected, performance increases significantly when increasing the clock speed of the execution core. According to

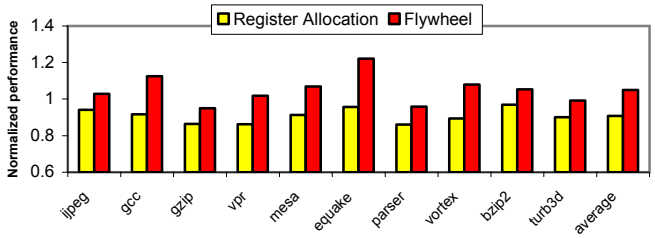

Fig. 11. Performance of the baseline processor using the Dual-Clock Issue Window and the new Register Allocation mechanism and of the Flywheel microarchitecture at the baseline clock speed. All results are normalized with respect to the baseline performance (Table 2).

the assumptions made in Section 4, we limited our exploration to a 50% increase in the execution core clock speed and an up to 100% increase in the front-end clock speed (Table 1). We assumed that all internal modules scale according to their section of the pipeline, while the external memory does not scale and thus the number of clock cycles required for a memory access increases correspondingly.

In Figure 12, clock speeds are presented as pairs, with the first value representing the increase for the front-end (FE) and the second value representing the increase for the execution core (BE). In all tests, the back-end of the pipeline (BE) is sped-up with 50% in trace-execution mode, while being synchronous with the Issue Window in trace-creation mode. The front-end (FE) clock is varied from the Issue Window speed (FE0%) to twice this speed (FE100%). While in some cases the front-end clock speed does not influence much the overall performance (*vpr*, *parser*), in other cases it helps significantly. The large effect seen in *vortex*, where the speed-up increases from 29% (FE0%, BE50%) to 59% (FE100%, BE50%), is mainly generated by the reduction in mispredict penalty achieved by the faster clock speed in the front-end. In this case, the alternative execution path is only used for less than 60% of the time. In all the other cases, the faster front-end mainly helps bringing more instructions to the Issue Window, exposing more parallelism for the traces that are to be stored and replayed from the EC.

An interesting observation is that the performance of our microarchitecture actually scales super-linearly with the clock speed. In the (FE50%, BE50%) case where we apply the same clock speed increase to both the front-end and the execution core, the overall performance increases by an average of 54%. This fact is caused by the increased parallelism that we expose by accumulating instructions faster in the Issue Window. In Figure 12, all values are normalized with respect to the fully synchronous baseline microarchitecture.

In terms of total energy, our microarchitecture is more effective than the baseline because we use less hardware for executing each instruction when replaying traces from the EC. Whenever the processor uses the alternative execution path, all the dynamic power that would normally be consumed in the front-end is saved. On average, the *Flywheel* microarchitecture saves almost 30% in energy across our entire set of benchmarks, with larger savings for benchmarks like *gcc* and

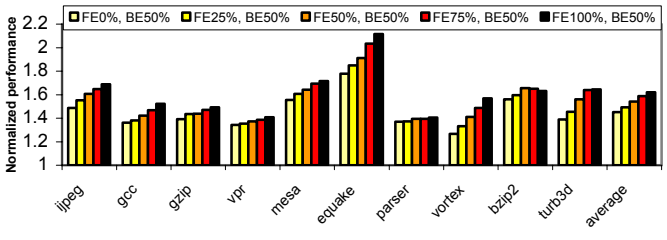

Fig. 12. Relative performance of the Flywheel microarchitecture when increasing the clock speed in both the front-end and in the execution core by up to 100% for the front end (FE) and 50% for the back-end (BE). Speed-up factors have been computed based on latency analysis using Cacti (Table 1).

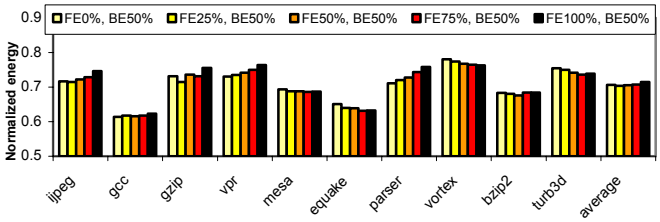

Fig. 13. Relative energy consumed by the Flywheel microarchitecture when increasing the clock speed in both the front-end and the execution core. All results are normalized with respect to the baseline microarchitecture.

equake and smaller savings for *vortex* where the front-end utilization is larger (Figure 13). As the front-end clock speed increases, the higher dynamic power can be noticed in some benchmarks (e.g., *vpr, parser*) while in other cases the higher execution parallelism is more important and the total energy decreases (*equake*, *turb3d*).

While the total energy remains relatively constant with increasing the front-end clock frequency, the power actually increases. The average power required for the (FE0%, BE50%) case is only 2% larger than the baseline power consumption, but for the (FE100%, BE50%) this average grows to 15% more than the baseline. However, the *Flywheel* microarchitecture offers a large increase in performance for a significantly smaller increase in power, making this processor more suitable for applications where the power consumption is a priority. In the (FE50%, BE50%) case, our microarchitecture offers 54% increase in performance for only 8% increase in power consumption (Figure 14).

As process technology evolves and transistors are getting smaller, dynamic power becomes a smaller fraction of the total power used by the processor. Thus, the power efficiency of the *Flywheel* microarchitecture will be significantly diminished. In Figure 15, we study the evolution of the achievable energy savings in 130nm, 90nm and 60nm process technologies for the (FE100%, BE50%) case. While at 130nm this configuration saves almost 30% of the total energy, at 60nm we project that it will still be more efficient than its baseline counterpart, by saving around 20% of total energy.

6. Related work

Using multiple clock frequencies in the same chip design, Globally Asynchronous, Locally Synchronous circuits have been proposed by previous studies as ways for exploiting the trend towards making functional blocks more autonomous [16]. Superscalar, out-of-order processors built under this

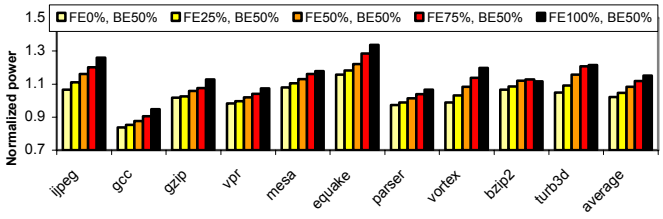

Fig. 14. Relative power consumed by the Flywheel microarchitecture when increasing the clock speed in both the front-end and the execution core. All results are normalized with respect to the baseline microarchitecture.

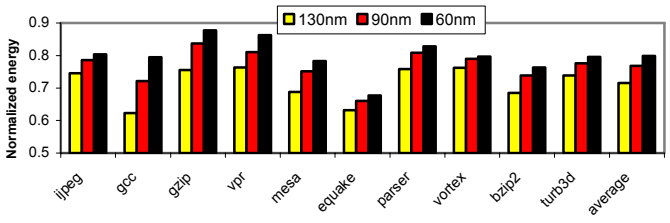

Fig. 15. Relative energy for the Flywheel microarchitecture at 130nm, 90 nm and 60 nm. All results are normalized with respect to the baseline in their respective process technology.

paradigm have been evaluated for both performance and power consumption [17][9][18]. While performance can be significantly worse than in the fully synchronous case, these papers identify the ability of the GALS processor to use different clock frequencies and supply voltages for each of the synchronous islands.

The circuitry needed for decoupling the front-end of the pipeline from the execution core is similar to the one proposed in [11]. There, a more general case is considered. For our purpose, the Dual-Clock Issue Window design can be greatly simplified since it will always interface a faster front-end with a slower Issue circuitry. While the front-end of the pipeline needs to be decoupled from the Issue Window using a mixed-clock interface, increasing the clock frequency in the execution core can be done without introducing another such interface. The proposed mechanism relies on the ability to completely bypass the slower circuitry of the Wake-Up/Select by using an Execution Cache to store instructions in issue order.

A somewhat similar mechanism has been previously explored by Nair in [19]. The DIF engine is a processor with a dual core. The first core is a simple, single-issue pipeline, capable of scheduling traces of instructions and storing them in a special cache. Should these traces be re-executed, they will be fed directly to a more parallel, VLIW-like execution core, improving the performance when compared against the initial single-issue pipeline.

In the MPS microarchitecture [20], an external instruction scheduler is added to a VLIW core. Normally, the execution takes place in the simple, parallel engine and the MPS (Miss Path Scheduler) is used only infrequently to bring instructions from the memory and schedule them (creating VLIW words and storing them in the Instruction Cache for further reuse). Since the instruction and data caches are assumed infinite and the branch prediction is based on profiling, the performance is dictated here only by the VLIW core capabilities.

Storing instructions in the logical program order rather than actual memory order was previously proposed [21][22][23]. Usually, the trace-cache employed in all these studies is used as a mechanism for improving the fetch efficiency and allowing for multiple branch predictions during each clock cycle. An exception is the TurboScalar microarchitecture [24], where a long and thin pipeline is used for creating and scheduling traces that feed a very short and thick pipeline.

A Trace Cache-based mechanism aimed at both improving performance and reducing power consumption has been proposed by Intel [25][26]. By storing traces of decoded instructions, their design can bypass the Fetch and Decode stages and send instructions from the Trace Cache directly to the renaming mechanism. Due to the use of CISC ISA, the parallel x86 decoders have been traditionally one of the most complex parts of the processor, limiting the maximum achievable clock frequency and amounting for an important part of the total power budget. Going one step further, it has been proposed [3][12] to move a similar structure deeper in the pipeline, after the Issue logic. Such a design would allow reusing all the work performed in the front-end of the pipeline, offering significant energy reductions at the expense of a small decrease in performance. However, as shown in this paper, such a design can be beneficial for both energy and performance if used in a properly designed architecture.

As shown by Palacharla *et al.* [2], a large Issue Queue is likely to become a problem for both clock speed increase and wider pipelines. Several solutions to deal with it have been proposed. A straightforward approach is to pipeline the issue logic by splitting the Wake-Up and Select phases into two separate pipeline stages and using Speculative Wake-Up to compensate for the loss in IPC [7]. As instructions are waken up using tags that have not been selected yet, the Select stage is removed from the critical loop leaving the Wake-Up circuitry alone to dictate the clock speed. While IPC is only slightly affected if the processor has enough execution resources, this solution does not address the issue of quadratic increase in the Wake-Up latency. As this latency starts dominating the Issue Window speed, this solution loses its efficiency. In addition, the energy-performance operating point for this design is inferior to the one proposed in this paper as it targets only performance, without regard to power efficiency.

Other solutions focus on reducing the complexity of the Issue Queue through limiting the number of instructions that must be woken-up simultaneously. Thus, Palacharla *et al.* [2] proposed to implement the scheduling mechanism as a collection of dependency queues in which only the topmost instruction in every such queue is woken-up. Hrishikesh *et al.* [1] proposed to organize the Issue Queue into multiple stages and forward results to only one stage during each clock cycle.

Instead of reducing the complexity (and implicitly its effectiveness) of the Issue Window, our microarchitecture aims at limiting its influence in the overall performance. Using the parallelism exposed by a large Issue Window, but without having to settle for the limited clock speed imposed by it, we are effectively removing it as a bottleneck for future scalability.

7. Conclusions

As transistors become smaller and faster, a large Issue Queue is likely to become a bottleneck for both clock speed increases and more parallel designs. First, its latency is quadratically dependent on the number of in-flight instructions. Second, the Wake-Up delay is dominated by

interconnections, so it will not scale well with future process technologies. Considering that the rest of the pipeline is likely to scale better with future process technologies and increasing parallelism, methods that circumvent this limitation will become more and more important.

In this paper, we proposed a method for circumventing the traditional tradeoff between parallelism and clock speed. Our microarchitecture can benefit from the potentially higher clock speeds achievable in the front-end of the pipeline and in the execution core. While still using a large and slow scheduling mechanism, the proposed design effectively limits its importance in the overall performance.

Our analytic estimates (based on Cacti) show that significant clock speed increase can be expected during next process technologies, for both the front-end and back-end execution core. Based on these estimations, our results show that the *Flywheel* microarchitecture will scale very well, with performance increase surpassing the increase in clock frequency. For a 50% clock speed increase, our microarchitecture shows a 56% performance increase at the expense of only 9% increase in power. In terms of total energy, the *Flywheel* microarchitecture is more efficient than its equivalent superscalar, out of order counterpart, providing up to 30% energy reduction across various benchmarks.

8. References

[1] M. S. Hrishikesh, D. Burger, N. P. Jouppi, S. W. Keckler, K. I. Farkas, and P. Shivakumar, *"The optimal logic depth per pipeline stage is 6 to 8 fo4 inverter delays,"* in Proc. of the 29th International Symposium on Computer architecture, May 2002.

[2] S. Palacharla, N. P. Jouppi, and J. E. Smith, *"Complexity-effective superscalar processors,"* in Proc. of the 24th International Symposium on Computer Architecture, June 1997.

[3] E. Talpes and D. Marculescu, "*Power Reduction Through Work Reuse*," in Proc. of International Symposium on Low Power Design, August 2001.

[4] S. Wilton, and N. Jouppi, "*CACTI: An enhanced cache access and cycle time model*," in IEEE Journal of Solid-State Circuits, vol. 31, nr. 5, pages 677-688, May 1996.

[5] S. McFarling, "*Combining branch predictors,*" Technical Report DEC WRL Technical Note TN-36, DEC Western Research Laboratory, 1993.

[6] K. C. Yeager, "*The MIPS R10000 superscalar microprocessor,*" in Proc. of the International Symposium on Microarchitecture, April, 1996.

[7] J. Stark, M. D. Brown, and Y. N. Patt, "*On pipelining dynamic instruction scheduling logic,"* in Proc. of the 33th International Symposium on Microarchitecture, December 2000.

[8] F. Theeuwen, E. Seelen – "*Power Reduction through Clock Gating by Symbolic Manipulation*" - Workshop on Logic and Architecture Synthesis, 1996.

[9] D. E. Lackey, P. S. Zuchowski, T. R. Bednar, D. W. Stout, S. W. Gould, and J. M. Cohn, "*Managing Power and Performance for System-on-Chip Designs using Voltage Islands,*" in Proc. of the International Conference on Computer-Aided Design, Nov 2002.

[10] R. W. Brodersen, M. A. Horowitz, D. Markovic, B. Nikolic, and V. Stojanovic, "*Methods for True Power Minimization*," in Proc. of ICCAD*,* November 2002.

[11] V. S. Rapaka, E. Talpes, and D. Marculescu, "*Mixed-Clock Issue Queue Design for Energy Aware High-Performance Cores,*" in Proc. of the Asia and South Pacific Design Automation Conference, January 2004.

[12] *E. Talpes and D. Marculescu, "Design Space Exploration for Execution Cache-based Microarchitectures,"* in Proc. of International Symposium on Low Power Design, August 2004.

[13] D. Burger, T. Austin, and S. Bennet, "*Evaluating Future Microprocessors: the SimpleScalar Tool Set*," Technical Report 1308, University of Wisconsin, Madison, July 1996.

[14] D. Brooks, V. Tiwari, and M. Martonosi, "*Wattch: A Framework for Architectural-Level Power Analysis and Optimization*s," in Proc. of the International Symposium on Computer Architecture, June 2000.

[15] J. A. Butts, and G. S. Sohi, "*A Static Power Model for Architects*," in Proc. of the International Symposium on Microarchitecture, pages 191–201, December 2000.

[16] T. Meincke, A. Hemani, S. Kumar, P. Ellervee, J. Oberg, D. Lindqvist, H. Tenhunen, and A. Postula, "*Evaluating benefits of Globally Asynchronous Locally Synchronous VLSI architecture,*" in Proc. of the 16th Norchip, November 1998.

[17] A. Iyer and D. Marculescu, "*Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors,*" in Proc. of the International Symposium on Computer Architecture, May 2002.

[18] *G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. L. Scott, "Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling,"* in Proc. of the Symposium on High Performance Computer Architecture, February 2002.

[19] R. Nair and M. Hopkins, "*Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups*," in Proc. of International Symposium on Computer Architecture, June 1997.

[20] S. Banerjia, S. Sathaye, N. Menezes and T. Conte, "*MPS: Miss-path Scheduling for Multiple-issue Processors*," IEEE Transactions on Computers, Vol. 47, No. 12, December 1998.

[21] E. Rotenberg, S. Bennett, and J.E.Smith, "*A trace Cache Microarchitecture and Evaluation*," in IEEE Transactions on Computers, Vol. 48, No. 2, February 1999.

[22] S. J. Patel, D. H. Friendly, and Yale N. Patt, "*Evaluation of Design Options for the Trace Cache Fetch Mechanism*," in IEEE Transactions on Computers, February 1999.

[23] B. Black, B. Rychlik, and J. P. Shen, "*The Block-based Trace Cache*," in Proc. of the International Symposium on Computer Architecture, May 1999.

[24] B. Black and J. P. Shen, "*Turboscalar: A High Frequency, High IPC Microarchitecture*," in Proc. of the International Symposium on Computer Architecture, June 2000.

[25] B. Solomon, A. Mendelson, D. Orenstein, Y. Almog, and R. Ronen, "*Micro-Operation Cache: A Power Aware Frontend for Variable Instruction Length ISA*," in Proc. of International Symposium on Low Power Electronics and Design, August 2001.

[26] R. Rosner, A. Mendelson, and R. Ronen, "*Filtering Techniques to Improve Trace-Cache Efficiency*," in Proc. of the International Conference on Parallel Architectures and Compilation Techniques, September 2001.

[27] E. Sprangle, and D. Carmean, "*Increasing Processor Performance by Implementing Deeper Pipelines*," in Proc. of the International Symposium on Computer Architecture, May 2002.

[28] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Rousel, "*The Microarchitecture of the Pentium 4 Processor*," in INTEL Technology Journal, 2001.