Execution Cache-Based Microarchitecture for Power-Efficient Superscalar Processors

Emil Talpes and Diana Marculescu, *Member, IEEE*

***Abstract*—This paper investigates a possible solution to the problem of power consumption in superscalar, out-of-order processors by proposing a new microarchitecture, specifically designed to reduce increasing power requirements of high-end processors. More precisely, we show that by modifying the well-established superscalar processor architecture, significant savings can be achieved in terms of power consumption. Our approach aims at limiting the growing amount of power used in a typical processor for dynamic optimizations (including out-of-order scheduling and register renaming). Our proposed approach achieves significant power savings by reusing as much as possible from the work done by the front-end of a typical superscalar, out-of-order pipeline, via the use of a special cache nested deeply into the processor structure. By reusing instructions that are already *decoded, reordered,* and have their registers already *renamed*, the front end of the pipeline can be turned off for large periods of time with significant savings in the overall power consumption. Experimental results show up to 35% (30% on average) savings in average energy per committed instruction, and 35% (20% on average) savings in energy-delay product, with about 9% average performance loss, over a large spectrum of SPEC95 and SPEC2000 benchmarks.**

***Index Terms*—Computer architecture, microprocessors.**

I. Introduction

TODAY'S superscalar processor microarchitectures place an increasing emphasis on exploiting instruction-level parallelism. This often translates into having multiple execution units that can accommodate a large variety of instruction mixes. In order to support all these execution capabilities and to achieve a significant speedup, the conventional instruction path needs to include wide issue buffers to allow a larger number of in-flight instructions and wide instruction paths to feed all available execution units.

Traditionally, performance concerns have taken priority over energy costs or power consumption. Power efficiency has been addressed mainly at the technology level, through lower supply voltages, smaller transistors, silicon-on-insulator (SOI) technology, better packaging, etc. Nevertheless, power dissipation has become one of the primary design constraints for modern processors, and thus, microarchitecture designers must now take power requirements into consideration as well. Most of the processors that are currently intended for mobile applications support techniques like frequency and voltage scaling [1], [2] in order to reduce their power requirements in those applications where high performance is not required.

Another research direction focuses on microarchitectural techniques that aim at limiting the power consumed in areas of the design that are not needed during specific computations. Gating the input transitions has been one of these techniques, proposed mainly as a potential way to reduce the power consumption for large designs [3]. Focusing on a specific component of the processor, techniques like filter cache [4] or *L*-cache [5] were proposed to increase the power efficiency of the cache subsystem.

More recently, an interesting architectural innovation implemented by Intel in the Pentium 4 microprocessor is the use of a special cache that shortens the branch misprediction path [6], [7]. By placing this cache in the pipeline after the x86 decoding stages, and by storing decoded instructions (uops) in the trace-cache, the whole decode stage can be shut down for significant periods of time while the rest of the execution engine continues working. When a hit in the trace-cache occurs, instructions do not need to be decoded again and can be fed into the pipeline directly from the trace-cache.

Moving one step forward, one can envision such an execution cache (EC) as being placed even deeper in the pipeline to allow for even further improvements through shortening the branch misprediction path. If the cache is placed after the issue stage, instructions that are fetched, decoded, and have already had registers renamed can be stored in *issue-order* (and not in *program-order*) in the EC. Assuming that the efficiency of this EC is very good, most of the time instructions are executed out of this cache and fed directly to the execution engine. However, when a trace is not available, the front-end of the pipeline can be restarted such that the execution is resumed like in a normal superscalar processor.

Like all caches proposed or used in current microprocessors, our EC is based on the code locality property. Usually, programs are comprised of instructions executed many times in a short interval of time. Furthermore, once an instruction is repeatedly executed, there is a large probability for the subsequent program sequence to be re-executed in the same fashion. Since all current superscalar processors are built to dynamically optimize the execution of program sequences by reordering instructions so they can be executed faster, it makes sense to assume that most of the optimizations will be done in the same fashion, each time the sequence is re-executed.

To this end, we propose a novel micro-architectural organization that allows for better power efficiency through reusing the work done by the front-end of the pipeline. Furthermore, techniques like guarded evaluation [8] or clock gating [3] will enable

Manuscript received September 4, 2002; revised May 13, 2003, January 24, 2004, May 18, 2004. This work was supported in part by the National Science Foundation (NSF) under Career Award CCR-008479, and in part by the SRC under Grant 2001-HJ-898.

The authors are with Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: etalpes@ece.cmu.edu; dianam@ece.cmu.edu).

Digital Object Identifier 10.1109/TVLSI.2004.840406

TABLE I
BASELINE MICROARCHITECTURE PARAMETERS

Parameter	Value
Pipeline Width	Eight-way, out-of-order
Instruction Window	128-entry
Load Store Queue	64-entry
Register file	MIPS-like, 128 physical registers, full renaming
I-Cache	32K, 2 way, block size 32 bytes, LRU replacement policy
D-Cache	32K, 4 way, block size 32 bytes, LRU replacement policy
L2 Cache	Unified, 256K, 4 way, block size 64 bytes, LRU replacement policy
L2 access time	10 cycles
Memory access time	100 cycles
Branch predictor	Gshare, 10 bits history
Functional Units	8 Integer ALUs, 4 Integer MUL/DIV 4 Memory address units 4 FP Adders, 2 FP MUL/DIV
Technology	0.13um (high speed STMicro technology) V_{dd} = 1.8V, V_t = 0.2V
Normalized leakage current per device [9]	80 nA

significant reductions in power consumption for pipeline stages that are not used during different phases of the program execution. If the front-end of the pipeline is not used for long periods of time, we can shut it off and thus obtain further savings in the leakage power.

This paper is organized as follows. Section II gives a brief motivation for our work, while Section III presents some of the previous work that is most relevant for our approach. Section IV presents the main aspects of our microarchitectural design, including the selected EC structure, the organization of the new register file, and the register renaming technique that will allow us to reuse renaming information collected while building the traces. The experimental setup is described in Section V and the results of these tests are included in Sections VI–VII. In Section VIII, we present the power-performance tradeoffs available when using this type of microarchitecture. We conclude in Section IX with some final remarks and possible directions for future research.

II. Motivation

Probably the most important disadvantage that out-of-order processors have over other cores not relying on dynamic scheduling hardware is their relative power inefficiency. To identify the major sources of energy overhead, we have considered a superscalar, out-of-order pipeline (Table I).

Analyzing the power consumption for each major component of the superscalar architecture, we obtain the breakdown in Fig. 1. To this end, we have assumed that each component not used can be clock-gated, on a per-cycle basis, with an overhead given by the leakage current. The leakage value presented in Fig. 1 is obtained using the models proposed by Butts and Sohi [9]. The normalized leakage current per device was estimated as in [10].

A large part of the total power budget is required for the modules accounting for the front-end of the pipeline, branch predictor, *I*-cache, and instruction window. To achieve better power

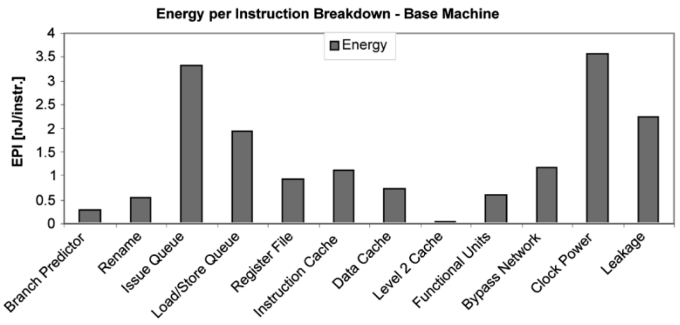


Fig. 1. Energy per committed instruction for *ijpeg* (0.13 μm process technology, $V_{\rm dd} = 1.8$ V, $f_{\rm clk} = 1.0$ Ghz, large input set).

efficiency we focus on these front-end modules, trying to reuse as much as possible from the already performed work. Obviously, the functions performed by the fetch and decode stages are identical each time a specific trace from the program is executed. Using a novel register file structure, we can also reuse the work done by the rename and issue stages.

For truly achieving work reuse, we propose a new microarchitecture, with a modified type of trace-cache placed between the issue and execute stages, storing traces of instructions in issue order. This new microarchitecture will allow turning off the front-end of the processor for large periods of time, saving the power required by all the modules belonging to these pipeline stages.

III. Related Work

When it comes to performance, high-end processor designers have always been the last to accept a possible compromise. As intended for applications where raw performance is the primary target, the last bit of potential efficiency is usually squeezed from each architectural design. In this respect, all the power-reduction work has usually been concentrating on defining mechanisms at lower levels of abstraction. Traditional circuit-level approaches, as voltage scaling, transistor resizing or libraries redesign [12], are now employed by most modern superscalar processors.

Guarded evaluation [8] was proposed as a static technique to reduce the power required by a combinational circuit when some of the operands do not change through successive time steps. Another method, clock gating [3], was proposed to save the power wasted by units that are temporarily not used. Both of these techniques require some extra logic (or a static algorithm) to identify when sub-blocks of the larger design that are not used in order to prevent input transitions from occurring.

These techniques are currently widely accepted and tools like Wattch [11] that model the power consumption of a superscalar processor consider them as implemented by default. However, even though high performance commercial processors implement clock gating to some extent, most of them do not use it very aggressively. This is mostly due to the inherent difficulties in predetermining, on a cycle-by-cycle basis, whether a module will be needed or not, or in finding longer intervals when a module can be completely shut down.

A mechanism aimed at using a very long instruction word (VLIW)-like core for improving the performance of a processor was previously explored [13]. Unlike our approach, the DIFF engine is a dual core processor. The first core is a simple, single-issue pipeline that is capable of scheduling traces of instructions and store them in a special cache. Should these traces be re-executed, they will be fed directly to a secondary, VLIW-like execution core, greatly improving the performance when compared against the initial single-issue pipeline.

In the miss path scheduler (MPS) microarchitecture [14], an external instruction scheduler is added to a VLIW core. Normally, the execution takes place in the simple, parallel engine and the MPS 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. MPS does not use two execution cores but, unlike both our approach and DIFF, assumes an infinite *I*-cache. Since the external instruction scheduler is very seldom used, the overhead incurred for scheduling the VLIW words is not an issue.

Storing instructions in the logical program order rather than actual memory order has been previously proposed [15]–[17]. 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 [18], where a long and thin pipeline is used for creating traces that feed a very short and thick pipeline, thus harvesting a much higher instructions per cycle (IPC).

All these studies have focused on increasing the performance of the processor core, but do not address the power consumption. Modified DIFF or MPS implementations that use limited caches can conceptually be quite close to our design. However, DIFF uses two different pipelines, resembling TurboScalar more than our implementation [18]. Furthermore, all these examples assume the existence of a separate scheduler for a VLIW engine. In these studies, the complexity of creating traces and storing them in an efficient structure is taken off of the critical execution path and placed after the Retire stage of the pipeline. Thus, they avoid most of the performance drop caused by the extra work required for filling the trace-cache and can perform a lot of optimizations before storing the instructions (reordering, renaming, and hashing). However, they do not address nor quantify the substantial power overhead incurred by all the required logic.

More recently, a trace cache-based mechanism aimed at both improving performance and reducing power consumption was proposed by Intel [6], [7], [19]. Storing traces of decoded instructions, this design can bypass the fetch and decode stages and send instructions from the trace cache directly to the Rename. Due to the 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 accounting for an important part of the total power budget. Since the new trace cache based microarchitecture only relies on these decoders for a small fraction of time, it can use a much simpler (albeit, a lower performance) decoding scheme, reducing the overall power requirement.

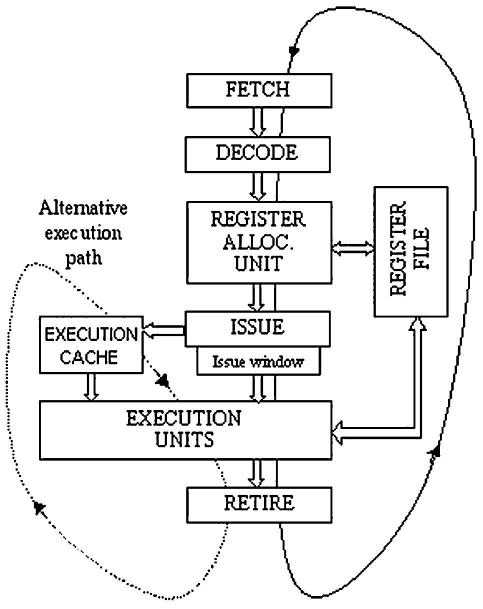


Fig. 2. Superscalar microarchitecture using an EC for reusing scheduled instruction streams.

Unlike the Intel-proposed trace cache, the work described herein intends to reuse as much as possible from the work performed in decode, rename and issue mechanisms instead of just bypassing the decoders. While this approach is not intended to increase performance, it can yield some important savings in the power required by the entire pipeline front-end.

The general approach has been previously described in [20]. In this paper, we propose a different caching technique to improve the utilization of the EC. Thus, we can reduce the required cache size, obtaining comparable results under more realistic assumptions. Also, we include a more in-depth study of this microarchitecture and we evaluate the advantages and limitations of this approach.

IV. Microarchitectural Design

In this paper, we propose a more power efficient processor microarchitecture. In order to achieve this when an instruction is re-executed, the goal is to reuse as much as possible from the computations performed during previous executions.

To reuse the work and reduce the branch misprediction path, we propose to place an EC deep in the pipeline, after the issue stage. To avoid the performance penalty incurred by an extra pipeline stage between the issue and the execution stages, instructions are issued in parallel to both the execution stage and the EC. The conceptual microarchitecture is illustrated in Fig. 2.

Normally, instructions are fetched from the *I*-cache through the fetch stage and then decoded. In the next stage, physical registers are assigned for each logical register, avoiding potential false dependencies. The resulting instructions are placed in the issue window for dependency checking. A number of independent instructions are issued to the execution stage and, in parallel, added to a *fill buffer* in order to create program traces. When enough instructions are placed in this fill buffer, the entire program sequence is stored in the EC in the issue order, for later potential reuse.

In this setting, the branch misprediction path can be significantly shortened by feeding the execution units directly from EC whenever possible. Initially, when EC is empty, instructions are launched from the issue window, while a trace is built in parallel. We call this step the *trace segment build* phase. Upon a mispredict (or a trace completion condition), a search is performed to identify a possible next trace starting at that point, and should a hit occur, the instructions continue to be executed from the EC, on the alternative execution path (dotted line in Fig. 2). When operating on this alternative execution path, the processor behaves essentially 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 must be launched again and a new trace is built.

A. EC Architecture

Similar to the conventional trace-cache implementations, our design divides the program into traces of instructions that are stored in a different order than the one given by their original addresses. This design allows to implicitly include 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 they can be retrieved only on a sequential basis. However, in order to allow for traces to be reused, 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 at each change of trace the processor must restart trace execution in-order.

Specifically, at each trace end, a *trace look-up* step must be performed. While most of the time the performance 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 an in-order start of each trace, this leads to some performance penalty associated with each trace change.

To minimize the overall performance impact of this design, traces must be as long as possible. While most trace-cache designs proposed in the literature limit the traces to at most three basic blocks, in our case it is desirable to include as many instructions as possible. However, as they get longer, the number of traces that can be accommodated in a reasonably sized cache decreases. This leads to a hit rate decrease at trace look-up, so a higher front-end utilization is achieved. While this does not impact overall performance significantly, it increases the power consumption since both the front-end and the EC (in trace-build mode) are simultaneous utilized. To address this problem, we allow the maximum size of the traces to be dynamically modified. The trace size is *proportional* to the *number of branch mispredicts*, such that longer traces are obtained whenever program locality is very good.

The simultaneous presence of traces with different lengths in the EC (some of them very long) has prevented the use of a standard trace cache model [16], [17] or the very efficient block based trace-cache structure [15]. Thus, we have decided to go for a solution that resembles the Intel implementation of the Pentium 4 microarchitecture [6] because it can accommodate traces of arbitrary length. The block architecture of this solution is presented in Fig. 3.

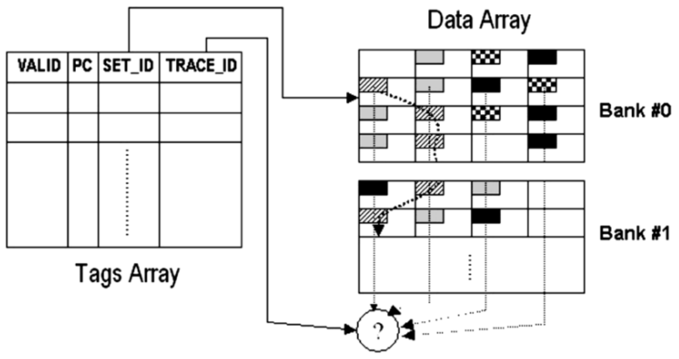


Fig. 3. EC structure.

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. SET_ID (in the TA) points to the DA set containing the start of the desired trace. DA is a multi-way set associative cache (four-way in our case) composed of multiple memory banks. A comparison with TRACE_ID is performed for each block in the set to identify the correct starting block. The next chunk of instructions is located in one of the blocks of the following set, and so on (see Fig. 3). 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 each subsequent read. Furthermore, knowing the next set 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. Thus, the entire array is used only when accessing the first instructions in a trace. On all subsequent accesses, the energy consumed by the line decoder and by the unused banks can be saved. Depending on the application, the ratio of the accesses made to only one bank against the total number of accesses can vary between 1:2 (e.g., *gcc*) and 3:4 (e.g., for floating point benchmarks).

Inside each block, an arbitrary number of issue Units is stored (Fig. 4). 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 (when the front-end of the pipeline is used) and then reused in all subsequent executions of the trace, the processor will take the same optimizing decisions each time it executes the code.

While in the previous implementation [20] only one issue Unit was stored in each block, here we allow as many as possible. This allows for a significant reduction in the number of accesses and thus, in the power required by the EC. All instructions coming from the issue window are first assembled into traces using the fill buffer and then recorded to the EC. The fill buffer can accommodate two DA blocks, and when enough instructions are available to fill a block, they are written to the EC.

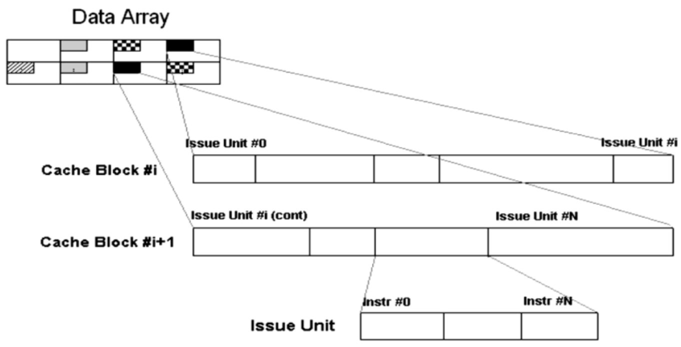


Fig. 4. EC block organization.

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.

In our proposed design, each block contains more than one issue unit, and thus, needs more than one clock cycle to be sent to the execution core. This organization allows for hiding some of the EC latency because an access can be started before actually being forced to stall the pipeline. At the same time, using longer cache lines increases the power required per access, but it also helps reducing the total number of accesses. Across our set of benchmarks, the longer line proved to be slightly more efficient in terms of power consumption.

The downside of increasing the block size is that the efficiency of the EC decreases, instruction slots remaining empty. We have focused on blocks that can accommodate eight or sixteen instructions, and our tests have shown that the eight instructions version can be significantly more efficient in terms of instruction space storage.

A least recently used (LRU) policy is used for freeing up blocks in each set from the DA when a new trace-building phase is initiated. To terminate the creation of a trace, the trace-building algorithm takes into account several criteria like: trace length, occurring mispredicts, jumps, and the ability of finding another existing trace starting at the current point.

B. Trace Creation and Ending Policies

When creating a new trace, instructions are added until an end condition is met. Such end conditions can occur when the trace grows beyond a maximum length, when we encounter hard to predict instructions (e.g., function returns or indirect branches) or when a branch mispredict occurs and execution must resume from a different address. When fetching instructions from the EC, execution is abandoned on trace-end (detected when attempting to fetch more instructions) or on a mispredict (detected by the write back stage). To replace a trace, a lookup is first performed on the TA; if this search misses, the front-end of the pipeline is restarted. Should a hit occur, instructions are issued directly from the EC, after observing the trace look-up penalty.

We point out that a trace is created following a number of branch predictions. Since we must store very long traces in a reasonably sized cache, we cannot afford to store multiple traces starting at the same physical address. Even more, storing multiple such traces would require a predictor for selecting between them, and such a predictor would be prohibitively complex. As no reasonably accurate predictors have been proposed for predicting more than three branches in a single cycle, conventional trace cache architectures limit the trace size to three basic blocks [15], [16].

Since our traces can be hundreds of instructions long, they can contain much more than three basic blocks. We tried to select among multiple program traces starting at the same physical address by using a limited predictor (predicting only two branches in advance), but results did not improve significantly. For this reason, we have decided against implementing a next-trace predictor, which would prove very inaccurate for more than three branches, while requiring an important energy overhead.

In our case, the trace is declared invalid and another one created if previous predictions prove to be wrong. The policy implemented here is to declare a trace invalid when M mispredictions are encountered in a row, on the same trace. Traces created on unusual code paths tend to generate a large number of mispredictions during re-execution, and they are filtered out by this mechanism. While these traces are filtered out faster and performance improves when M decreases, the usage of the front-end of the pipeline also increases, leading to an increase in power consumption. We have used values of two and three for M (that is, replace the trace after two or three consecutive mispredicts).

As mentioned above, the EC must be quite large so its power requirements can offset a large part of the savings. To address this issue, we try to avoid accessing it very often by using a large fill buffer. Also, given the known pattern of most accesses, we split the EC into banks that are gated when not in use. An *I*-cache with the same size would require significantly more power, as it would be accessed every cycle and its access pattern would not allow sub-banking. As it will be seen in Section VI, the results for a larger *I*-cache show some significant performance benefit (relative to the EC version) for some benchmarks, but at the cost of more than three fold increase in the *I*-cache power.

C. The Register File

Placing the above-described EC deep in the pipeline, after the issue stage, allows for reusing the work done by all the front-end stages. This also implies that register renaming is not performed on the instructions issued directly from the EC. Being different at each trace run, the values held by the register file cannot be stored in the EC and reused. However, register renaming is only performed in trace creation mode. This operating mode assumes that the virtual-to-architected register mapping is the same at the beginning of each trace. Some architectural changes need to be brought to the register pool and control unit to ensure that this can be implemented. We use a special register pool structure for this task, conceptually similar to the one proposed by Nair and Hopkins [13]. The logic structure is presented in Fig. 5.

Our structure employs a special pool of physical registers for renaming every logical register in the ISA. In our proposed microarchitecture, each architected register is organized as a circular buffer of physical registers, as opposed to a stack organization that was proposed previously [21].

Unlike a typical register file, where an architected register can be renamed to any physical entry, in our implementation an

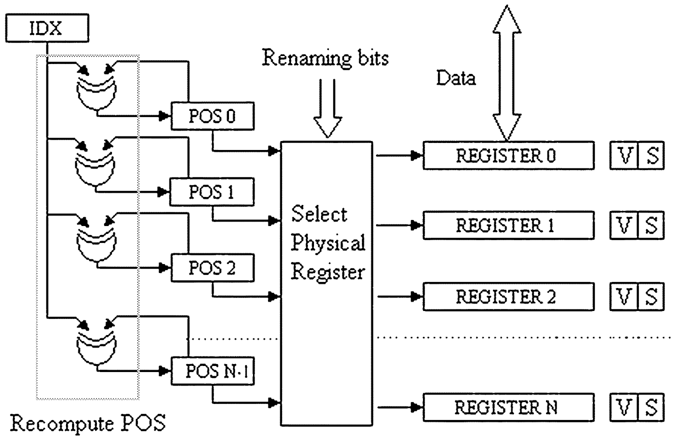


Fig. 5. Architected register structure.

architected register can be renamed by using just the physical entries of the corresponding pool. However, each subsequent write goes to a different physical register of the pool using a deterministic algorithm. This approach solves the problem of potential false data dependencies and can be used for implementing register renaming. Because of its very predictable behavior, this mechanism is similar to what has been proposed for implementing modulo-scheduling [17].

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 registers available in the circular buffer. This limitation reduces the renaming capacity when compared against other register renaming schemes. However, following the usual register allocation convention used by most compilers, we note that some registers are written very often (e.g., those holding local variables), while others are mostly read (e.g., stack pointer, return address, etc.). Following these conventions, we can use a different number of renaming registers for implementing each architected register, thus alleviating the renaming capacity problem while also preventing the register file structure from becoming too large.

When going through rename each instruction is allocated a physical register as destination, other than the one holding the last known value for the corresponding architected register. Having different physical destinations, instructions can write the result as soon as it is available, setting the V (valid) bit. This bit is used to detect where in the register file we have valid data. If all the source registers for an instruction have this bit set, the instruction can be issued to the execution unit. The bit is cleared when the physical register is allocated as a destination in the rename stage and it is set when the value is written in the write back stage. The actual value in the physical register (register $0 : N-1$) is only accessed when we have to perform the read/write operation.

The S (speculated) bits are used for specifying committed values and they are cleared only after the instruction is retired. In order to be able to roll back the speculative state when a mispredict (or interrupt) occurs, we need at least one committed value in the circular queue. Whenever a rollback condition occurs, the index is reverted to this "last committed" value for the architected register. A physical register cannot be assigned as a destination for a new instruction (in the register renaming stage) if the associated S bit is set. If this happens, there are not enough physical registers to perform renaming at this moment and the Rename stage stalls.

The N indices (values $\mathrm{POS0}$–$\mathrm{POS}N-1$) are initialized with consecutive values $(0, 1, 2, \ldots, N-1)$ and represent the logical order of the registers in the circular queue. IDX is a pointer in this queue representing the most recently allocated register. All accesses (to the value field or to the status bits V and S) are associative, comparing the renaming information against the POS tags. While executing a trace, the POS tags remain constant and represent the logical order of the circular queue (the order the registers are allocated). When the trace starts, the physical register with tag 0 $(\mathrm{POS} = 0)$ holds the actual value for the architected register.

D. Register Renaming

When the instruction reaches the renaming stage, physical registers must be assigned as its source and destination registers. The IDX value is the index of the physical register holding the last value written to that architected register. It is incremented (modulo N) for each "write" operation so successive writes to the same architected register will actually use different physical registers. For each "read," IDX is read and assigned to the instruction as a physical register selector.

The S (speculated) bit is checked for the corresponding physical register and, if it is found set, the pipeline is stalled. Otherwise, S is set and V (Valid) is reset to mark the value as not yet available. V will be set when the result is computed and written back to the register, while S will be deleted later, when the instruction is retired.

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

For an easier understanding of this mechanism, let us assume the following example. Assume a code is executed in a loop, with each iteration consisting of the following sequence:

```
Addr0:
Mov r1, #5        ;r1 ← 5
Mov r2, r0        ;r2 ← r0
Add r3, r1, r0    ;r3 ← r1 + r0
Sub r2, r3, r1    ;r2 ← r3 − r1
Xor r1, r1, r2    ;r1 ← r1 ∧ r2
Jmp Addr0         ; go back to the beginning
```

Let us assume that the trace will be created unrolling only two successive iterations of this loop (this can hold or the trace can be longer, but it doesn't make a difference for the register renaming algorithm). For this example, we also assume the use of four physical registers for each architected register. After unrolling, the trace becomes the following:

```
0. Mov r1, #5         ;r1 ← 5
1. Mov r2, r0         ;r2 ← r0
2. Add r3, r1, r0     ;r3 ← r1 + r0
3. Sub r2, r3, r1     ;r2 ← r3 − r1
4. Xor r1, r1, r2     ;r1 ← r1 ∧ r2
5. Jmp Addr0          ; go back to the beginning
                      ; (6 after the loop is unrolled in the EC)
6. Mov r1, #5         ;r1 ← 5
7. Mov r2, r0         ;r2 ← r0
8. Add r3, r1, r0     ;r3 ← r1 + r0
9. Sub r2, r3, r1     ;r2 ← r3 − r1
10. Xor r1, r1, r2    ;r1 ← r1 ∧ r2
11. Jmp Addr0         ; go back to the beginning
                      ; (restart the same trace)
```

When this trace is executed for the first time, the front-end of the pipeline, issue and register renaming is performed. At the beginning of the trace the IDX field is zero for all four registers used here. The registers are renamed as following:

```
0. Mov r1, #5         ;r1.1 ← 5 (IDX is incremented for r1)
1. Mov r2, r0         ;r2.1 ← r0.0 (IDX incremented for r2,
                      ; (IDX is found 0 for r0)
2. Add r3, r1, r0     ;r3.1 ← r1.1 + r0.0
3. Sub r2, r3, r1     ;r2.2 ← r3.1 − r1.1
4. Xor r1, r1, r2     ;r1.2 ← r1.1 ∧ r2.2
5. Jmp Addr0          ; continue with instruction 6
6. Mov r1, #5         ;r1.3 ← 5
7. Mov r2, r0         ;r2.3 ← r0.0
8. Add r3, r1, r0     ;r3.2 ← r1.3 + r0.0
9. Sub r2, r3, r1     ;r2.0 ← r3.2 − r1.3
10. Xor r1, r1, r2    ;r1.0 ← r1.3 ∧ r2.0
11. Jmp Addr0         ; go back to the beginning
```

In representing the above trace, we have used the notation *ri.j* for physical register j in the pool associated with the architected register i.

When the trace ends, each physical queue needs to be reorganized. For register r0, IDX remains set to 0 so all POS fields remain unchanged. For registers $r1$ and $r2$, IDX will be again zero (in our example we have four writes per trace for these registers), so the relative ordering in the circular queue remains the same. For $r3$, IDX is 2 at the end of the trace, so the XOR will actually modify all POS values. Physical entry #2 will become #0 and the rest will receive a new position in the circular queue. When retrieving the same trace from the EC, we execute directly:

```
0. Mov r1.1, #5
1. Mov r2.1, r0.0
2. Add r3.1, r1.1, r0.0
3. Sub r2.2, r3.1, r1.1
4. Xor r1.2, r1.1, r2.2
5. Jmp Addr0
6. Mov r1.3, #5
7. Mov r2.3, r0.0
8. Add r3.2, r1.3, r0.0
9. Sub r2.0, r3.2, r1.3
10. Xor r1.0, r1.3, r2.0
11. Jmp Addr0
```

As it can be seen, all renaming information is already present. Execution starts with all valid values in the physical entry #0 and the first destination, they will not destroy the old value. Of course, when re-executing the trace, physical register #0 in each pool may be different than last time. The associative scheme ensures that all traces will see the same register configuration each time they are executed. Another valid option here is to physically copy the values toward the origin of each queue when a trace ends. However, by using the associative approach, one can avoid copying the values and the potential latency associated with such an operation.

E. Supporting Precise Interrupts

A very important aspect of an instruction set architecture (ISA) is the support of precise interrupts. Should an exception occur, most legacy ISAs guarantee an in-order state before executing the service trap. To do so, the instructions must be tagged during dispatch in such a way that the retire stage knows how to place them in-order before testing potential exceptions. When executing instructions from the EC, we must make sure we have enough information to support reordering.

As described above, all accesses to the register file structure are associative on each architected register; a new trace is not allowed to access the registers before the checkpoint occurs at the end of the previous trace. This can be a potential drawback, since we essentially introduce a few bubbles in the pipeline between traces. On some trace changes, these bubbles appear anyway, due to branch mispredictions. We take into consideration these effects and include them in the overall performance numbers presented in Section VI.

However, such an operation mode guarantees that the issue window can only contain instructions belonging to a single trace. As in a normal out-of-order processor, instructions are tagged for in-order retirement before being sent to execution. Similar to the issue window, the retire stage can store only one trace, so the reordering tags are still valid. This fact allows in-order instruction retiring, each time using the same retire buffer allocation that was recorded during trace creation.

V. Experimental Setup

To validate our approach, we have modified the SimpleScalar microarchitectural simulator to support our register file and EC models, a longer pipeline and an operating mode based on inter-stage buffers. Also, to support instructions coming either in-order, from the front-end or out-of-order, from the EC, we had to modify the simulator to execute all the instructions

TABLE II
CONTENT OF THE DATA ARRAY

Per Instruction (x8 per entry)	Decoded instruction	48 bits
	Renaming information	6 bits (2 for each register)
	Trace tag	10 bits
	Retire position	6 bits
	Sequence id	1 bit
Per entry	Number of Issue Units	3 bits
	Trace Id	32 bits
	Type (first, last, middle)	2 bits
Total		76 bytes

TABLE III
CONTENT OF THE TAG ARRAY

Starting address	64 bits
Starting set in the Data Array	8 bits
Trace id	32 bits
Misprediction count	2 bits
Total	14 bytes

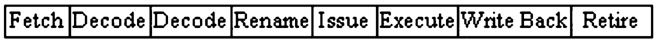


Fig. 6. Base pipeline microarchitecture.

out-of-order, in the execution stage instead of the dispatch stage. For the baseline microarchitecture, the register renaming mechanism chosen is similar to the one used by the MIPS R10000 processor [22].

The power models were based or similar to the ones used in Wattch [11]. For the modules that do not exist in the original SimpleScalar framework, we have used the same methodology. The EC modules (the tag array and the data array) have been modeled as regular caches optimized for speed (similar to the L1 cache models used in Wattch). The register file was broken in two modules: a regular register file that holds data and a rename file that holds the renaming information. The physical register file (together with the POS values) is simulated as an associative array, and one V (valid) bit is associated with each entry. The rename file includes the IDX and the S values, and is simulated as a directly mapped array.

For evaluating the leakage power, we have used the methodology proposed in [9]. The model is based on the estimation of the total number of devices for the entire processor, as well as their type.

As the combinational circuitry required by the checkpointing mechanism is very simple, we do not include its power consumption. The circuitry involved for checkpointing is very simple, the only operation needed being a 3-bit XOR between the values in POS and the values in IDX for all entries in the register file (67 ISA registers with four physical entries per register). Furthermore, this circuitry is not used during each cycle, but only when a trace is started, reducing its effect in the overall power consumption.

All results presented in the next sections assume an EC that can hold eight instructions per block in the data array. For each entry, the recorded information includes the *decoded instruction*, the *renaming information* for each of the three registers, a *trace tag*, the *retire position* and the *sequence identifier* (Table II). In this study, we have assumed an Alpha-like ISA, with 32-bit code and a factor of expansion after decode of 1.5 (thus, each stored *decoded instruction* is 48 bits long). The *trace tag* is created in the decode stage and is used for retrieving the initial instruction ordering inside the trace when this ordering is needed (e.g., when an interrupt or mispredict occurs inside the trace). The *retire position* is allocated during Rename and it is essentially an index in the retire buffer, allowing for supporting precise interrupts. On each reuse, instructions will retire in the same manner, using the information stored in this field. The *Sequence Id* is the same across instructions belonging to the same issue unit, but it is toggled when an issue unit ends to signal the last instruction that can be issued in parallel.

Overall, each entry is 76 bytes long. We assume a single clock cycle for passing this information from the EC to the fill buffer, due to a very wide bus. However, since most of the time we can start an access several cycles in advance, a narrower bus can be used, sending a block over two or three cycles. Furthermore, if an entire issue unit is present in the fill buffer, it can be sent to the execution before the entire line is retrieved from the data array.

The content of the Tag Array is presented in Table III. Each entry must hold the *Starting Address* that must be matched against the PC. A second address is needed for the Data Array, and it indicates the first set where the trace is stored. The *trace Id* is a value that identifies the DA entries belonging to the same trace and the *Mispredict count* stores the number of times that a misprediction occurred on the current trace.

We have considered an 8-stages pipeline, which is close to the pipeline depth used in some of today's processors. Its conceptual structure is presented in Fig. 6.

The main parameters of the microarchitecture we considered are presented in Table IV. We have accounted for the difference in global clock power due to an increased number of pipeline registers that have to be clocked. To validate our results, we have used a mix of benchmarks from both SPEC95 and SPEC2000. All tests were performed by fast forwarding over the first 500 M instructions and then doing detailed simulation for the next 50 M instructions.

VI. Performance Results

Generally, the performance of our processor is lower than the one achieved by the corresponding superscalar design. While performance is actually improved for benchmarks with a poor code locality, it can drop significantly for benchmarks with a very low miss rate in the first level *I*-cache. For the pipeline shown in Table IV, the EC version performed up to 6.5% faster than the basic one for benchmarks like *mesa* or *gcc* and up to 19% slower for *ijpeg* or *parser* (see Fig. 7).

Overall, our microarchitecture is 8.5% slower on average when using a large EC (100K), or 9.8% slower when using a smaller one (50K). All values presented in Fig. 7 are normalized with respect to the base microarchitecture. While most benchmarks run slower on this new microarchitecture, for

TABLE IV
MICROARCHITECTURE PARAMETERS

Parameter	Value
Pipeline Width	Eight-way, out-of-order
Instruction Window	128-entry
Load Store Queue	64-entry
Register file	Baseline: MIPS-like, 128 registers, full renaming EC-based: 32-entry, 4 physical registers/entry
I-Cache	32K, 2 way, block size 32 bytes, LRU replacement
D-Cache	32K, 4 way, block size 32 bytes, LRU replacement
L2 Cache	Unified, 256K, 4 way, block size 64 bytes, LRU replacement
L2 access time	10 cycles
Main memory access time	100 cycles
Branch predictor	Gshare, 10 bits history
EC Data Array	50K (100K), 4 way, 8 instr. / block, 4 banks
EC Tag Array	4K, 4 way, LRU replacement
Maximum Trace Length	512 instructions
EC Access Time	2 cycles for the first entry (1 cycle for TA access) 1 cycle for subsequent accesses to the same trace
Functional Units	8 Integer ALUs, 4 Integer MUL/DIV 4 Memory address units 4 FP Adders, 2 FP MUL/DIV
Technology	0.13um (high speed STMicro technology) $V_{dd} = 1.8\text{V}$, $V_t = 0.2\text{V}$
Normalized leakage current per device	80 nA

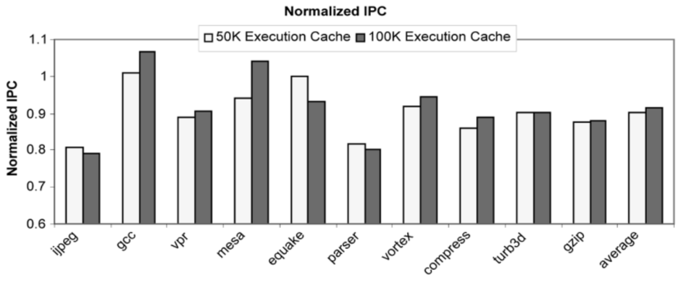


Fig. 7. IPC variation for the EC microarchitecture architected register structure.

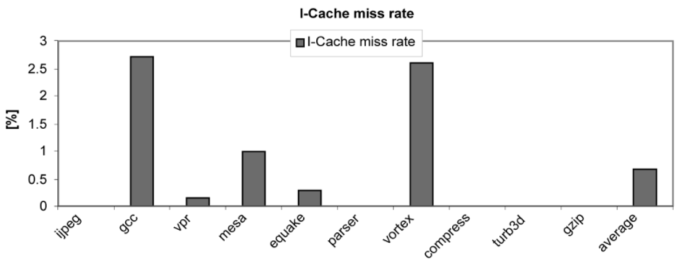


Fig. 8. Benchmark behavior: *I*-cache miss rate (baseline architecture as in Table IV, but without EC).

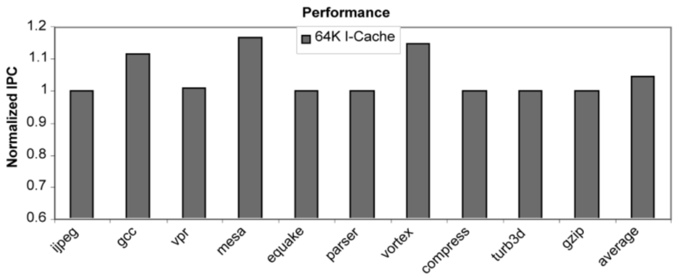


Fig. 9. Performance variation for larger *I*-cache sizes (baseline architecture, normalized with respect to the version using a 32K *I*-cache, as in Table IV.

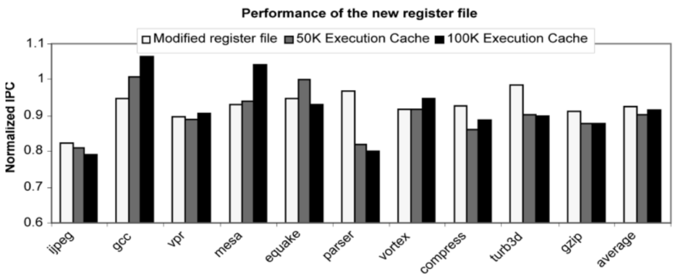


Fig. 10. Performance of a modified baseline using the queue-based register file (normalized against the baseline using the MIPS-like register file).

some of them the performance can actually increase. This may be considered somewhat surprising, as our approach is meant to alleviate power consumption problems and not necessary increase performance.

The proposed structure performs better on *mesa* and *gcc*, while it performs worse than the baseline microarchitecture on other benchmarks like *parser*, *gzip* or *compress*. Comparing these results against the *I*-cache miss rate (Fig. 8), we notice that our microarchitecture performs better when the *I*-cache miss rate is important in the baseline. Due to the much larger EC that is now storing the instructions, the overall miss rate is reduced and this factor tends to be significant for some of the benchmarks. An exception is *vortex*, where the new EC does not help much in reducing the *I*-cache miss rate. In this case, the number of traces that our microarchitecture creates is rather large, and the entries of the EC have to be refilled with new instructions very often.

Since our microarchitecture achieves maximum speed-up on benchmarks where the *I*-cache miss rate is high, it makes sense to test the performance and power requirements of a superscalar processor that uses a larger *I*-cache. In order to minimize the fetch miss rate, we doubled the number of sets in the *I*-cache, bringing its size to 64K. The normalized performance increase is presented in Fig. 9.

Significant performance gains are observed for benchmarks where our EC-based microarchitecture also performs better or very close to the baseline microarchitecture (*gcc*, *mesa, vortex*). These gains are obtained at the expense of an increase in power consumption—the 64K *I*-cache requires 60% more power than the original 32K cache (Section VII).

As even larger performance gains can be obtained for the baseline processor if the *I*-cache size is increased, it becomes clear that the proposed microarchitecture does not offer any significant performance benefits. In order to identify the actual cause for the performance degradation that we see on most of the benchmarks, we also simulated a modified baseline, using the queue-based register file (Fig. 10).

Except for *gcc, mesa,* and, to a lesser extent *equake*, where the original higher *I*-cache miss rate helps the EC-based microarchitecture, the modified baseline still performs similarly or even

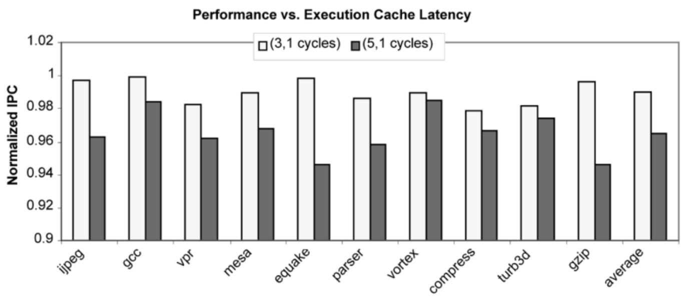


Fig. 11. Effect of the EC latency on the overall performance (normalized with respect to the (2, 1 cycles) EC-based microarchitecture). The baseline configuration (2, 1) assumes that the first access to a trace requires two cycles, and the subsequent ones can be performed in a single cycle. (3, 1) and (5, 1) are configurations where the first access takes three and five cycles, respectively, with subsequent accesses taking one cycle.

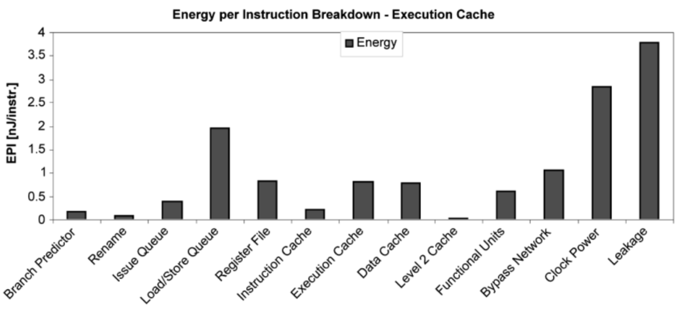


Fig. 12. Energy per instruction breakdown for *ijpeg* (0.13 μm process technology, $V_{\rm dd} = 1.8$ V, $f_{\rm clk} = 1.0$ Ghz, large input set).

significantly better than the EC-based microarchitecture (*parser, compress, turb3d, gzip*). This suggests that the smaller renaming capacity and the associative register access are not the only factors limiting the performance. However, most of the results are very close (less than 2% difference on average), which suggest that these are the main bottlenecks. The rest of the performance penalty is caused by the fixed scheduling not being able to cope with variable latency instructions, and by the slightly increased mispredict penalty observed for any miss in the EC.

Another interesting aspect is the effect the EC latency has on the overall performance. As pointed out in Section IV, most of these accesses can be started early, effectively hiding their latency. Accesses that cannot be predicted early are those that do not follow the normal execution path—at branch mispredictions or when loads issued speculatively collide with previous unexecuted stores. In all these cases, we have to sequentially access both the TA and the DA of the EC. We have run the same benchmarks, assuming a 100K EC and increasing the latency of these accesses from two cycles (as in the previous results) to up to five cycles. While some benchmarks experience a hit in performance of about 5% when using a five-cycle latency, for others the effect is negligible. On average, increasing this latency from two to five cycles results in about 3% average performance degradation (Fig. 11).

VII. Power Consumption Results

As presented in Section VI, the use of a large EC helps lowering the *I*-cache miss rate for two out of the ten benchmarks. On these benchmarks, performance of the proposed microarchitecture is similar to the one offered by a superscalar, out-of-order processor with a much larger *I*-cache. However, the larger *I*-cache translates in a significantly increase in the overall power requirements.

As not all benchmarks can benefit from the larger *I*-cache size, we decided against comparing our microarchitecture with a baseline that uses more than 32K of L1 *I*-cache. This way, the power consumption comparison is fairer for benchmarks that do not need the larger size cache. Should a different baseline microarchitecture be used, the power benefits will be significantly larger than the results presented in this section.

For reporting power consumption, we use the average energy per committed instruction (EPI). For a better understanding of our results, we start by presenting a breakdown of the power requirements for the major structures in our design. We have used here the same benchmark (*ijpeg*) as for the breakdown presented in Fig. 1 in order to estimate the overall savings that are achieved by our microarchitecture.

The results in Fig. 12 are obtained for *ijpeg*, using a 50K EC and correspond to a situation where roughly 90% of the time the processor is reading instructions from the EC. The EC energy cost includes both TA and DA energy values. While the power consumed in the front-end of the pipeline decreased (Instruction Window, *I*-cache), the other modules of the processor require roughly the same power (ALUs, load store queue, etc.). An interesting aspect is the big increase in leakage power, of almost 37%. This is caused by the introduction of a large cache (the EC) inside the pipeline, but also by the fact that many units are now unused, increasing their contribution to the total static power. However, most of the leakage due to array structures can be efficiently kept under control via the use of gated $V_{\rm dd}$ memory cells [25].

An interesting case is the EC: by sub-banking the DA we have managed to maintain reasonable power consumption. As the size grows, an increasing number of banks are needed in order to keep the power cost under control. However, the number of banks does not need to be very large. While our previous implementation [20] required very deep sub-banking to reduce power consumption, in this case a decent power cost can be achieved even with a reasonable number of banks.

For testing the overall power consumption of our microarchitecture, we considered two EC sizes (50K and 100K), both divided into four banks. With the smaller EC, average EPI is reduced by as much as 35% (e.g., *equake* or *compress*). Overall, EPI is reduced by 29%. In Fig. 13, all these values are normalized with respect to the energy requirements of the original pipeline organization.

When using a larger EC, the power consumption required by the cache increases. However, performance and the time spent executing instructions on the alternative execution path increase significantly for some benchmarks (*vortex*, *vpr*, *gcc*, *turb3d*). Thus, the increased power consumption of the EC is offset by the performance gains and we have slightly better results with an overall reduction of up to 31%.

Interesting cases are *gcc* and *vortex*. If EC size is doubled (to 100K), the hit rate increases and the alternative path usage becomes much better. If the 50K size EC is used for *gcc*, the

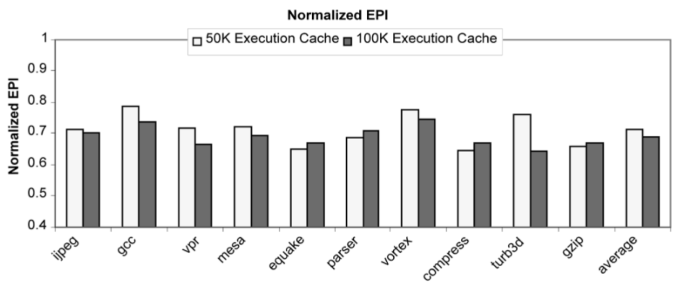


Fig. 13. EPI for a 50K and 100K EC organized in four banks (normalized with respect to the baseline presented in Table IV, without EC).

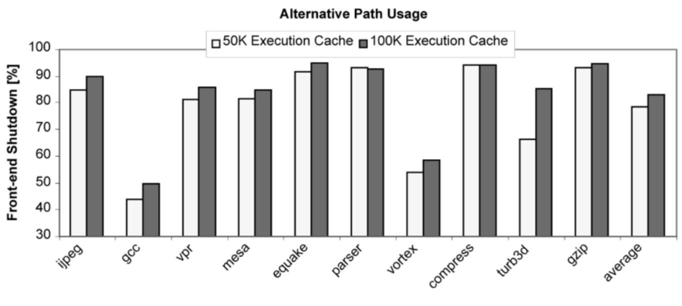


Fig. 14. Alternative execution path usage.

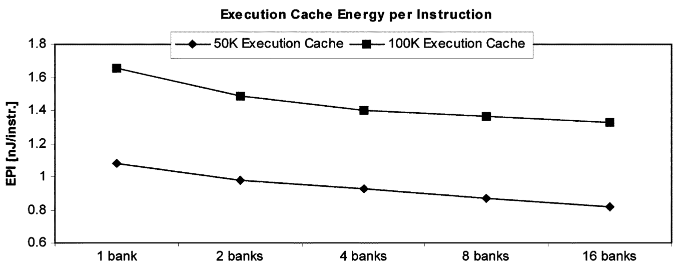


Fig. 15. EC EPI (including the associated leakage energy).

processor issues instructions from this cache only 43% of the time (compared to up to 94% of the time for *compress*).

In such situations, EC becomes less efficient since it may not be capable to accommodate a significant part of the application code. Traces are created more often than they are used, burning power both in the front-end and in the EC. For most benchmarks, instructions are issued from the cache more than 70% of the time, whereas in case of *gcc* the EC usage is only 43% if a 50K EC is used (Fig. 14). We can see the same situation (but to a lesser extent) for *vortex*, which is using the alternative execution path for only about 54% of the time.

Increasing the EC size to 100K increases the hit rate and thus, its efficiency on some of the benchmarks. Both EC configurations simulated in this study behave very well for FP benchmarks, because branches are very predictable and the number of possible traces is rather small. However, for integer benchmarks the overall utilization of the alternative path is much lower. The relatively long traces built in this case are only used a few times and then discarded. An example is *gcc*, where the alternative execution path usage is less than 45% of the total execution time for the 50K EC.

Another interesting aspect is the power consumption of the EC (see Fig. 15). The cache is divided in multiple banks, which can be separately turned off. The number of banks significantly

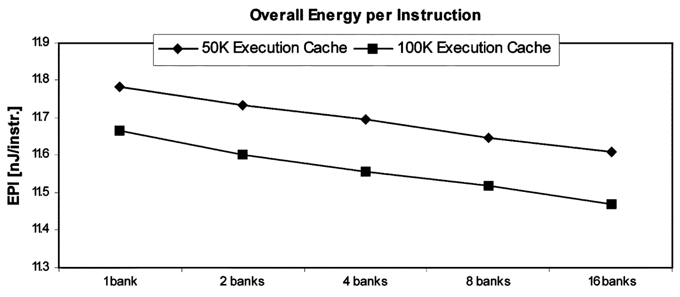


Fig. 16. Overall processor EPI (including the associated leakage energy).

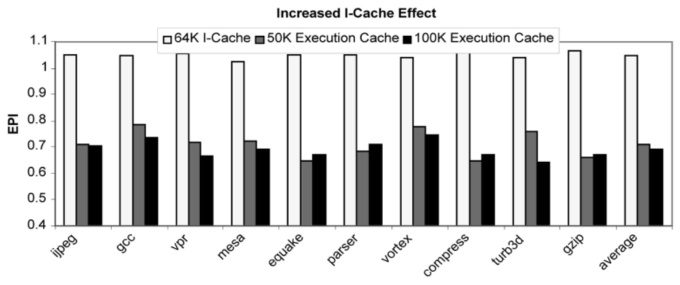


Fig. 17. Overall power consumption of the baseline microarchitecture using a larger *I*-cache (normalized with respect to the baseline from Table IV).

affects the power requirements of the EC. In order to study this effect, we have run *ijpeg* with different cache organizations.

The power consumption of the EC decreases significantly as we consider an implementation with several independent banks. However, increasing the number of banks above eight yields diminishing returns. Furthermore, the power required by the EC is only a small part of the overall power requirements of the microarchitecture, the differences being much smaller if we consider the EPI for the entire microarchitecture (Fig. 16).

In our tests, we have considered a four-bank EC organization. The overall gain that can be achieved by further splitting the cache is minimal and, in our opinion, a higher number of banks is not realistic for an actual implementation.

As seen in Section VI, a larger *I*-cache would significantly improve the performance of the original microarchitecture for several benchmarks. However, when using a larger *I*-cache, the power requirements increase as well. Fig. 17 presents the experimental results obtained with a larger L1 *I*-cache. For comparison, we included the results obtained using the EC. All results are normalized with respect to the baseline microarchitecture without EC and a 32K *I*-cache.

When increasing the *I*-cache to 64K, overall EPI increases by 5%. At the same time, when using the EC the energy per committed instruction decreases by up to 35%.

VIII. Power-Performance Trade-Off Analysis

From a performance point of view, the proposed microarchitecture has both advantages, as well as limitations. An obvious aspect is that the alternative execution path is shorter than the normal pipeline. This reduces the mispredict penalty when the next trace is found in the EC, and becomes a definite advantage when executing programs with bad branch predictability. As the current trend is to use deeper pipelines, the benefit is likely to increase. However, although placing the EC deep in the pipeline limits the branch mispredict penalty, there are some caveats associated to it. Each time instructions are issued from the EC, the normal branch predictor from the fetch stage is not used and the branch is speculated based on the pre-constructed trace. During trace execution, the branch prediction is equivalent to that of a 1-bit predictor, which will predict branches the same way as during the trace-build phase. By tuning the trace removal algorithm, the overall penalty can be decreased, choosing between optimizing for performance (a bad trace is rapidly erased) or power consumption (a bad trace is executed a few times before being erased).

Another potential drawback is the inability to schedule instructions around a variable latency operation. When a miss occurs in the data cache, a classic out-of-order microprocessor is able to execute other instructions. When scheduling is fixed at trace creation time, this ability is lost. However, most of these misses occur when the processor leaves a hot spot in the program and the cache needs to be refilled. Usually, these are the moments when the program traces need to be changed. We have tried avoiding the creation of traces in such cases, but the improvement in performance was marginal, at the expense of significant extra power consumed by the front-end of the pipeline. Some techniques for filtering out less useful traces generated during the execution of a program are presented in [19]. While we have not implemented them in our tests, they should be applicable on our microarchitecture as well.

The register file architecture introduces another potential drawback. As all accesses to this structure are associative, a new trace is not allowed to access the registers before the checkpoint occurs. Since very long traces are allowed, these trace changes appear mostly when a mispredict is detected. If a mispredicted branch takes multiple clock cycles for being retired after it was detected, it is possible for the next trace to be stalled waiting for this checkpoint to occur.

Overall, our microarchitecture doesn't offer any significant improvement in performance over the baseline superscalar, out-of-order processor. Our target is to obtain a structure with better power efficiency. We showed that this is possible, even though it comes at the cost of a small drop in performance. In terms of the Energy x Delay metric, our microarchitecture is performing significantly better, offering reductions of more than 20% over the baseline superscalar, out-of-order processor. There are a number of parameters that can be varied in order to tune the microarchitecture for better performance or better power efficiency, like the EC size and organization, the maximum trace length or the trace removal algorithm.

IX. Conclusion

In this paper, we have proposed a new microarchitecture aimed at increasing the power efficiency through reusing as much as possible from the work performed in the front-end of the pipeline. Our structure is slightly slower than a corresponding superscalar, out-of-order microarchitecture (8% to 9% on average) while using significantly less power. We show that this approach can achieve an average of 28% to 31% reduction in power consumption, depending on the EC size.

There are, however, drawbacks to this approach. Our structure splits the program into traces, breaking some of the potential parallelism that could have been harvested by the normal superscalar implementation. Furthermore, depending on the overhead considered for turning off the unused stages in the pipeline, the power savings could decrease.

A potential way to solve the problem of turning off the modules would be to use simultaneous multithreading (SMT). Being proposed as a way to increase the utilization of idle resources, the SMT architecture could use more efficiently the available hardware. Thus, one of the two threads can be taken from EC, while the other could be executed through the entire pipeline, avoiding the need to turn off front-end stages.

References

[1] A. Klaiber. (2000, Jan.) The Technology Behind Crusoe Processors. Transmeta Corp. [Online]. Available: http://www.transmeta.com

[2] L. T. Clark, "Circuit design of Xscale microprocessors," in *Proc. 2001 Symp. VLSI Circuits, Short Course Phys. Des. Low-Power, High-Performance Microprocessor Circuits*, Kyoto, Japan, Jun. 2001, [CD-ROM].

[3] F. Theeuwen and E. Seelen, "Power reduction through clock gating by symbolic manipulation," in *Proc. WLAS*, Grenoble, France, 1996, [CD-ROM].

[4] J. Kin, G. Munish, and W. H. Mangione-Smith, "The filter cache: An energy efficient memory structure," in *Proc. Int. Symp. Microarchitecture*, Dec. 1997, pp. 142–147.

[5] N. Bellas and I. Hajj, "Architectural and compiler techniques for energy reduction in high performance processors," in *Proc. ISLPED*, Aug. 1998, pp. 72–75.

[6] R. F. Krick *et al.*, "Trace Based Instruction Caching," Intel Corp., U.S. Patent 6,170,038, Jan. 2, 2001.

[7] B. Solomon, A. Mendelson, D. Orenstein, and R. Ronen, "Micro-operation cache: A power aware frontend for variable instruction length ISA," in *Proc. ISPED*, Aug. 2001, pp. 4–9.

[8] V. Tiwari, S. Malik, and P. Ashar, "Guarded evaluation: Pushing power management to logic synthesis/design," in *Proc. ISLPED*, Apr. 1995, pp. 221–226.

[9] J. A. Butts and G. S. Sohi, "A static power model for architects," in *Proc. Int. Symp. Microarchitecture*, Dec. 2000, pp. 191–201.

[10] E. Acar, A. Devgan, R. Rao, Y. Liu, H. Su, S. Nassif, and J. Burns, "Leakage and leakage sensitivity computation for combinational circuits," in *Proc. ISLPED*, Aug. 2003, pp. 96–99.

[11] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. ISCA*, Jun. 2000, pp. 83–94.

[12] V. Tiwari, D. Sigh, and S. Rajgopal, "Reducing power in high-performance microprocessors," in *Proc. Design Automation Conf.*, Jun. 1998, pp. 84–89.

[13] R. Nair and M. Hopkins, "Exploiting instruction level parallelism in processors by caching scheduled groups," in *Proc. ISCA*, Jun. 1997, pp. 13–25.

[14] S. Banerjia, S. Sathaye, N. Menezes, and T. Conte, "MPS: Miss-path scheduling for multiple-issue processors," *IEEE Trans. Computers*, vol. 47, pp. 1382–1397, Dec. 1998.

[15] B. Black, B. Rychlik, and J. P. Shen, "The block-based trace cache," in *Proc. ISCA*, May 1999, pp. 196–207.

[16] E. Rotenberg, S. Bennett, and J. E. Smith, "A trace cache microarchitecture and evaluation," *IEEE Trans. Computers*, vol. 48, pp. 111–120, Feb. 1999.

[17] S. J. Patel, D. H. Friendly, and Y. N. Patt, "Evaluation of design options for the trace cache fetch mechanism," *IEEE Trans. Computers*, vol. 48, pp. 435–446, 1999.

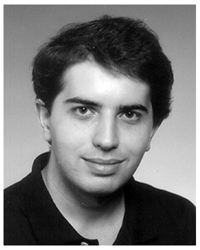
[18] B. Black and J. P. Shen, "Turboscalar: A high frequency, high IPC microarchitecture," in *Proc. Workshop Complexity-Effective Des.*, Vancouver, BC, Canada, Jun. 2000, [CD-ROM].

[19] R. Rosner, A. Mendelson, and R. Ronen, "Filtering techniques to improve trace-cache efficiency," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, Sep. 2001, pp. 37–48.

[20] E. Talpes and D. Marculescu, "Power reduction through work reuse," in *Proc. Int. Symp. Low Power Design*, Aug. 2001, pp. 340–345.

[21] "Scalable register Renaming Via the Quack register file," Tech. Rep., CMuART-2000-01.

[22] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," in *Proc. Int. Symp. Microarchitecture*, Apr. 1996, pp. 28–40.
[23] M. D. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated Vdd: A circuit technique to reduce leakage in cache memories," in *Proc. ISLPED*, Jul. 2000, pp. 90–95.
[24] F. Li and L. He, "Estimation of maximum power-up current," in *Proc. ASP-DAC*, Bangalore, India, Jan. 2002, pp. 51–58.
[25] A. Agarwal, H. Li, and K. Roy, "DRG-cache: A data retention gated-ground cache for low power," in *Proc. DAC*, New Orleans, LA, 2002, pp. 473–478.
[26] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proc. ISCA*, Denver, CO, 1997, pp. 206–218.



Emil Talpes received the M.S. degree in computer science from University Politehnica of Bucharest, Romania, in 2000. He is currently working toward the Ph.D. degree in the Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh. PA.

His research interests include computer architecture and energy-aware computing.

Diana Marculescu (S'94–M'98) received the M.S. degree in computer science from University Politehnica of Bucharest, Romania, in 1991, and the Ph.D. degree in computer engineering from the University of Southern California, Los Angeles, in 1998.

She is currently an Assistant Professor of Electrical and Computer Engineering at Carnegie Mellon University, Pittsburgh, PA. Her research interests include energy-aware computing, CAD tools for low-power systems, and emerging technologies (such as electronic textiles or ambient intelligent systems).

Dr. Marculescu is the recipient of a National Science Foundation Faculty Career Award (2000-2004), an ACM-SIGDA Technical Leadership Award (2003), and of the Carnegie Institute of Technology George Tallman Ladd Research Award (2004). She is an IEEE Circuits and Systems Society Distinguished Lecturer (2004-2005) and a member of Executive Board of the ACM Special Interest Group on Design Automation (SIGDA).