Power reduction through work reuse*

Emil Talpes
Carnegie Mellon University, ECE Department
5000 Forbes Ave
Pittsburgh, PA, 15213
412.268-3333
etalpes@andrew.cmu.edu

Diana Marculescu
Carnegie Mellon University, ECE Department
5000 Forbes Ave
Pittsburgh, PA, 15213
412.268-1167
dianam@ece.cmu.edu

ABSTRACT

Power consumption has become one of the big challenges in designing high performance processors. The rapid increase in complexity and speed that comes with each new CPU generation causes greater problems with power consumption and heat dissipation. Traditionally, these concerns are addressed through semiconductor technology improvements such as voltage reduction and technology scaling. This work proposes an alternative solution to this problem, by dealing with the power consumption in the very early stage of the microarchitecture design. More precisely, we show that by modifying the well-established out-of-order, superscalar processor architecture, significant gains can be achieved in terms of power requirements without performance penalty. Our proposed approach relies on reusing as much as possible from the work done by the front-end of a typical pipelined, superscalar out-of-order via the use of a cache nested deeply into the processor structure. Experimental results show up to 52% (20% on average) savings in average energy per committed instruction for two different pipeline structures.

1. INTRODUCTION

Today's superscalar processor microarchitectures place an increasing emphasis on exploiting instruction-level parallelism. This often translates into having multiple execution units, wider instruction issue buffers to support them and wider instruction paths for fetch, decode and rename stages. This trend leads to increasing power requirements to support all these resources. Until now, performance concerns have always had priority so the power dissipation issues were addressed mainly at the technology level, that is lower supply voltages, smaller transistors, SOI technology, better packaging, etc.

Nevertheless, power dissipation has become one of the design constraints for modern processors, and thus the microarchitecture designer must now take power requirements into consideration as well. Gating the input transitions has been one of the research directions proposed as a potential way to reduce the power consumption for large designs [10]. Focusing on a specific component of the processor, techniques like Filter Cache [8] or L-cache [9] were proposed to increase the power efficiency of the cache subsystem. Thus some parts of the cache are shut down when they are not needed, obtaining a better power efficiency.

More recently, an interesting architectural innovation unveiled by Intel with the release of the Pentium 4 microprocessor is the use of a trace-cache-like structure to shorten the critical execution path. By placing this cache in the pipeline, after the x86 decoding stages, and by storing the 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, creating a shorter critical execution path. 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, we can envision such a structure as being placed even deeper in the pipeline to allow for even further improvements through shortening the critical execution path [1]. If the trace-cache is placed after the Issue Stage, the instructions that are fetched, decoded and have already had registers renamed performed should be stored in *issue-order* (and not in program-order) in the trace-cache. The execution engine can thus be fed either from the Issue Stage (during the trace-build phase) or directly from the trace-cache (if a hit in the trace-cache occurs).

This new type of microarchitecture is the objective of this paper. 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 [5] or clock gating [10] will enable significant reductions in power consumption for pipeline stages not used during different phases of the program execution.

2. PREVIOUS WORK

When it comes to performance, superscalar processor design has 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 was usually concentrated on refining the CMOS technology. Traditional circuit-level approaches, as voltage scaling, transistor resizing or library redesign [6], are now employed by most of the modern superscalar processors.

Guarded evaluation was proposed as a static technique in [5] to reduce the power required by a design when some operands are left unmodified through successive time steps. More general, clock gating was proposed [10] to save the power wasted by units that are temporarily not used. Both of these techniques require some extra piece of logic (or a static algorithm) to identify when sub-blocks of the larger design are not used in order to prevent the input transitions. These techniques are currently widely accepted and tools like Wattch [12] that model the power consumption of a superscalar processor considers them as implemented by default. However, most of the commercial high performance processors are not able to use them on a large scale. This fact is mostly due to the inherent difficulties in predetermining on a cycle-by-cycle basis whether a module is needed or not in an out-of-order design or in finding longer intervals when a module can be completely shut down.

In this paper we propose to modify the usual pipelined, out-of-order microarchitecture to allow for longer (and predictable) intervals during which some of the resources are not used. In order to achieve that, we identify modules that perform the same computation each time an instruction is executed and try to reuse as much as possible from the

*This research has been supported in part by NSF Career Award CCR-008479.

previous 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 reusing all this work, 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*.

Storing instructions in the logical program order rather than actual issue order was previously proposed by several studies [2][4]. 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 [1], where a long and thin pipeline is used for creating traces that feed a very short and thick pipeline, thus harvesting a much higher IPC. All these studies are focused on increasing the performance of the processor but do not address the power consumption issue. In order to increase the performance, 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, these approaches avoid most of the drop in performance 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, these aforementioned microarchitectures do not address the substantial power overhead incurred by all the required logic.

3. ORGANIZATION OF THE PAPER

The paper is organized as follows: in Section 4 we present the main aspects of our microarchitectural design, including the selected trace-cache structure, the organization of the new register file and the register renaming technique that will allow us to reuse the renaming done when building the traces. In Section 5 we present the power-performance trade-offs we have to analyze when using this type of microarchitecture. The experimental setup is described in Section 6 and the results of these tests are included in Section 7. We conclude in Section 8 with some final remarks and possible directions for future research.

4. ARCHITECTURAL DESIGN

Typically, the microarchitecture-level design starts with the selection of a typical out-of-order, superscalar architecture. In order to reduce the critical execution path (and thus the number of modules that are used for executing an instruction) we propose to place a trace-cache deep in the pipeline, after the Issue Stage. In order to avoid the performance penalty incurred by an extra pipeline stage between the Issue and the Execution stages, we bypass it, issuing instructions in parallel to both the trace cache and the execution stages. The conceptual microarchitecture is illustrated in Figure 1.

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 used by the instruction, 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 stored in the trace-cache for potential reuse.

In this setting, the critical execution path can be significantly shortened by feeding the execution units directly from the trace-cache whenever possible. Initially, when the trace cache is empty, the instructions are launched from the Issue Window, while a trace is build in parallel in the trace-cache – 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 trace-cache, on the alternative execution path (dotted line in Figure 1). At the end of executing a trace (either at the end of the trace or when a mispredict occurs), the look-up must be performed again, in order to find a potential next-trace. If a miss is encountered, the pipeline front-end is launched again and a new trace is built.

4.1 Trace-cache Architecture

Similar to the conventional trace-cache implementations [4], our design divides the program into traces of instructions that are stored on a different basis than their original address. However, the cache we have chosen in the proposed architecture is structurally different from the trace-cache typically used for increasing the fetch bandwidth. From the beginning, the decision to reuse as much as possible from the work done in the first stages of the pipeline led to the necessity of storing the instructions in *issue order*, not in *program order*. This fact enforces a number of decisions that considerably limits the potential design space.

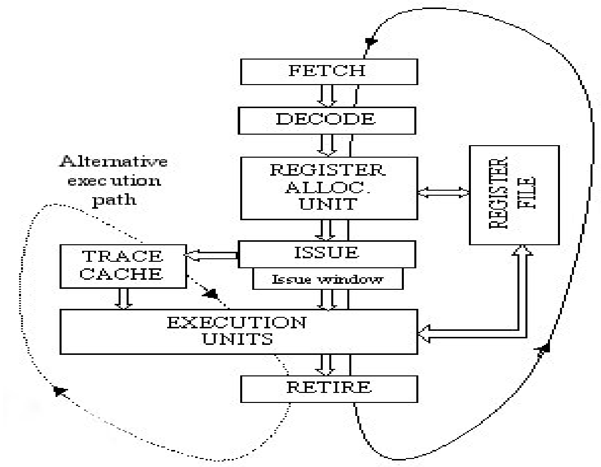

Figure 1. Superscalar microarchitecture that makes use of a trace-cache to shorten the critical execution path

If 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 the traces to be reused, the start address of each trace needs to correspond to a physical address in the memory space. So, with each change of trace, the processor must come back to an in-order execution status, leading to some breaks in the potential parallelism. As described later in the paper, we have not included complex or sophisticated branch prediction mechanisms that are typically used in trace-cache based microarchitectures. Such complex branch prediction hardware is very likely to offset any power savings that we achieve via dynamic work reuse.

At each trace end, a trace look-up step must be performed, leading to some more performance penalty. Because of the overhead associated with each trace change, the traces have to be as long as possible. However, as they get longer, the number of traces that can be accommodated in the cache decreases. This leads to a decrease of the hit rate at trace look-up, so we end up with a higher utilization of the front-end pipeline. In order to address this problem, we allowed the maximum size of the traces to be dynamically modified. The size is proportional to the number of mispredicts, in order to have longer traces whenever the program locality is very good.

The simultaneous presence of traces with different lengths (some of them very long) in the trace-cache prevented us from using the standard model [4] or the very efficient block based trace-cache structure [2]. So, we decided to go for a solution that resembles the Intel implementation in the new Pentium 4 microarchitecture [13][15]. This solution is presented in Figure 2.

The trace-cache structure consists of a tag array and a corresponding data array. The tag array is a highly associative cache, addressed using the program counter. It is used for trace look-up and it should be as fast as possible. The SET_ID value obtained from the tag array points to the set in the data array that contains the beginning of the trace we are looking for. The data array is an N-way set associative cache (in Figure 2, N=4). A comparison with the TRACE_ID is performed for each of the N blocks in the set to identify the starting block of the right trace. Each block generally contains more than one instruction – essentially the number of instructions that could be issued in parallel in the trace segment build phase. The next chunk of instructions is located in one of the blocks of the following set, and so on (see Figure 2). Knowing beforehand which is the next set, we avoid the trace look-up penalty at the subsequent reads. A special end-of-trace marker identifies the end of the trace.

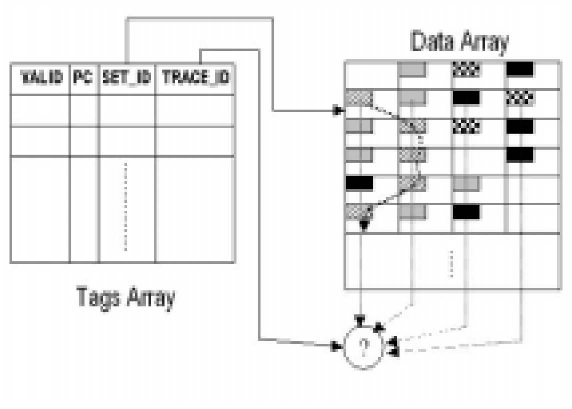

Figure 2. The trace-cache architecture

A LRU approach is used for freeing up blocks in each set from the data array 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.

To avoid the latency of searching in the highly associative tag array, a separate, small lookup table (FTLT – Fast Trace Lookup Table) is used to cache the most recently used trace tags. The trace is abandoned on trace-end (detected when attempting to issue more instructions to the execution engine) or on a branch mispredict (detected by the Retire Stage). When the trace must be replaced, a lookup is performed in the FTLT for a new trace. If a miss in the FTLT occurs, the full-search is performed in the tag array, and the front-end is restarted if a miss occurs in the trace-cache. On a hit in the trace-cache, instructions will be issued on the alternative execution path directly from the trace-cache, but incurring the trace look-up penalty.

We should point out that a trace is created following a number of branch predictions. If these predictions prove to be wrong, the trace must be declared invalid and another one created. The policy we implemented here is to declare a trace as invalid when we encounter two mispredicts in a row while executing the trace.

In this implementation, we use a fairly large Data Array (100k) for implementing the trace-cache. However, the access pattern for this structure is highly predictable (most of the cycles we just increment the row address we used for the previous access). This behavior allows us to use sub-banking [14] for implementing this structure and turn off the banks that are not being used in each cycle. All the banks have to be validated at the same time only when we start a new trace (after accessing the Tags Array).

4.2 Register File

Placing the above-described trace-cache deep in the pipeline, after the Issue stage, allows us to reuse the work done by all the units belonging to the front-end stages. This implies also that we do not perform register renaming on the instructions issued directly from the trace-cache. However, 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 made to the register pool and control unit to ensure that this can be implemented. In order to handle this task, we designed a special register pool structure. The logic structure we propose implementing each register is presented in Figure 3.

As the one proposed in [3], our structure employs a number of physical registers for renaming every logical register of the microarchitecture. In our proposed microarchitecture, each of the architected registers is organized as a circular buffer of physical registers, as opposed to a stack organization proposed in [3].

Using this type of structure, each subsequent write to an architected register goes to a different physical register. This approach solves the false data dependencies and is used for implementing register renaming. The number of in-flight instructions that can have the same logical destination is N (the number of physical registers in the circular buffer).

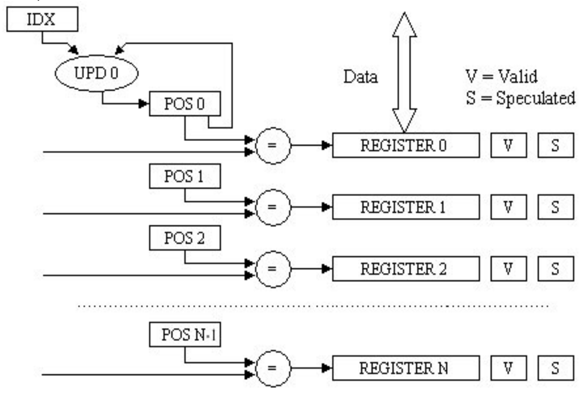

Figure 3. Architected register structure

Each read or write to this structure is associative, the physical register used being validated by a comparison between its position (POS) bits and the bits presented by the instruction. Having different physical destinations, instructions can write the result as soon as it is available, setting the V (valid) bit to signal an available value. However, the S (speculated) bit will be reset only after the instruction is retired. 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, we don't have enough physical registers to perform renaming at this moment and the rename stage will stall.

The N position 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.

4.3 Register Renaming

When the instruction reaches the renaming stage, some physical registers must be assigned to its source and destination registers. For the destination register, the IDX value is incremented and assigned to the instruction. 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 deleted to mark the value as not yet available. V will be set when the result is written back to the register and S will be deleted later, when the instruction will be retired. For a source register, the IDX value is read and assigned to the instruction.

Each trace generation is made with IDX starting from 0 (the correct value for the register is stored in the location marked by POS = 0). If this condition is respected, all the subsequent executions of a trace can be done without further renaming the registers. The caveat is that this requires some extra work to be done when a trace execution ends. In fact, all the POS values need to be recomputed for the circular buffer to start each time with the latest value for that architected register. This can be done subtracting from POS the IDX value, but it will require a complex circuit for each physical register. However, the same effect can be obtained performing a XOR between IDX and POS since the physical order of the registers is not important and does not have to match with the logical one. The important aspect – all registers to have different tags, between 0 and N-1 – is preserved and the register holding the last value becomes Register 0.

5. POWER – PERFORMANCE TRADE-OFF ANALYSIS

In this paper, we propose dynamic work reuse as a viable solution for power efficient microarchitecture. Using techniques like guarded evaluation and clock gating for shutting down the front-end (while issuing instructions from the trace-cache) should allow us to achieve significant reduction in the power consumption of the overall microarchitecture.

From a performance point of view, this microarchitecture has both strong points as well as weak points. From the conceptual structure, presented in Figure 1, it is obvious that the alternative execution path is shorter than the normal pipeline. This aspect considerably reduces the mispredict penalty when the next trace is found in the trace-cache, and is a definite advantage when executing programs with a bad branch predictability. This advantage should increase, as the current trend is to use deeper pipelines. However, although placing the trace-cache deep in the pipeline creates a shorter critical execution path, there are some caveats associated to it. First, each time instructions are issued from the trace-cache, we cannot make use of the normal branch predictor from the Fetch stage, and the branch is speculated based on the trace. Given this, during execution from the trace-cache, the branch prediction algorithm is equivalent to a 1-bit predictor[1], which will predict branches the same way as during the trace-build phase. Because we use longer traces, with a big number of potential branch instructions, a next-trace predictor would be very complex and would require a lot of power. However, since we cannot afford a very big trace-cache (because of the power concerns), creating more traces from the same program address for multiple possible branch predictions is not an option here. In addition, complex multiple branch prediction hardware is likely to offset any power savings obtained by work reuse.

Furthermore, when traces are created (or executed from the trace-cache), the parallelism obtained by out-of-order execution is exploited only within the bounds of a trace. If the length of the trace is small (e.g. not well-behaved programs), this fact can reduce the potential parallelism that otherwise could have been exploited by the normal microarchitecture. However, our target here is to obtain a structure with a better power efficiency and this should be possible even if we allow a small drop in performance. There are a number of parameters that can be varied in order to tune the microarchitecture for better performance or better power efficiency.

[1] Actually, it is more complex, almost like a 2-bit predictor due to the interference of the trace removal algorithm, detailed in Section 3.1.

6. EXPERIMENTAL SETUP

To validate our approach, we implemented two models based on the Wattch simulator [12]. We have modified the SimpleScalar microarchitecture in order to support our register file and trace cache models, a larger number of stages and a buffer based model for inter-stage communication. We implemented both the modified microarchitecture (with the trace-cache placed between the Issue and Execute stages) and a corresponding normal superscalar pipeline as a base for comparison. Both structures were based on the same pipeline and did not use some advanced features, like fetching from multiple cache blocks, etc. The main purpose of modeling those microarchitectures was to compare their behavior in terms of performance and power requirements.

We have considered 2 versions of the pipeline – one shorter, with 8 stages and another one, with 14 stages. The 8-stage pipeline was chosen because it is a common depth used in the today's processors. Its conceptual structure is:

Fetch	Decode	Decode	Rename	Issue	Execute	Write Back	Retire

Figure 4. Short pipeline microarchitecture

However, the current trend for achieving higher clock rates dictates an increase in the pipeline depth. For this purpose, we have also considered a 14-stage pipeline:

Fetch	Fetch	Dec	Dec	Dec	Rename	Dispatch	Issue	Issue	RD	Exec	WB	Retire	Retire

Figure 5. Long pipeline microarchitecture

For all tests we have used 4-ways pipelines, with the default configuration provided by the SimpleScalar toolset: 16k L1 I-Cache, 16k L1 D-Cache, 256k unified L2 cache, no penalty for accessing the L1 cache, 6 cycles for accessing the L2 Cache, 32 cycles for going to memory. The trace-cache is configured with 100k memory for the data array and 16k for the tag array. For looking up a trace in the tag array we have considered a penalty of 1 cycle. In both cases, we have accounted for the difference in global clock power due to an increased number of pipeline registers that have to be clocked. We have used the SPECint-95 and SPECfp-95 benchmarks to validate our results.

7. EXPERIMENTAL RESULTS

For an 8-stage pipeline, our microarchitecture performs up to 10% faster than the basic one for benchmarks like GCC, PERL or TURB3D and up to 5% slower for SWIM. All the values presented below are normalized with respect to the normal microarchitecture.

Overall, the proposed structure is 1.2% faster than the normal one. These results may not show the same trend as earlier results on using trace-cache [1]. The microarchitecture was tuned so as to yield approximately the same performance as the basic one, but requiring less power.

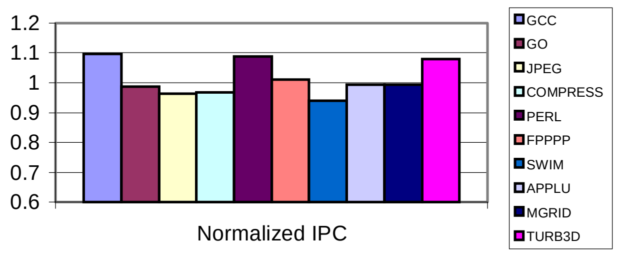

Figure 6. IPC variation for the 8-stage pipelines (100k trace cache)

To report the power consumption values, we have considered the energy required for committing one instruction (EPI). Since we assume clock gating is used for idle modules in the pipeline, we use the same technique implemented in Wattch for measuring the power consumption – by adding to the power required by the active modules a fraction of the power required by those unused. We report the results for 10%, 5% and 0% power overhead for unused modules. However, since most of the time instructions are executed from the trace-cache, completely shutting off the front-end could be a viable alternative, and thus the 0% overhead becomes realistic.

The power consumption is reduced for our microarchitecture by as much as 28% (48% for 0% overhead) for MGRID or TURB3D. Overall, the energy per committed instruction was reduced by 21% (for the 5% overhead model) and 36% if 0% overhead was considered. In Figure 7, all these values are normalized with respect to the power requirements of the original pipeline organization.

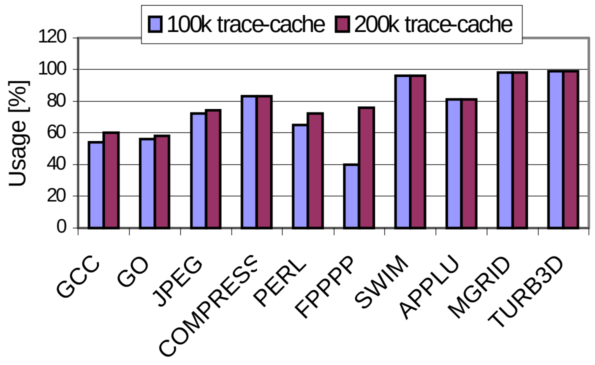

Figure 7. Normalized EPI for an 8-stage pipeline (100k trace-cache)

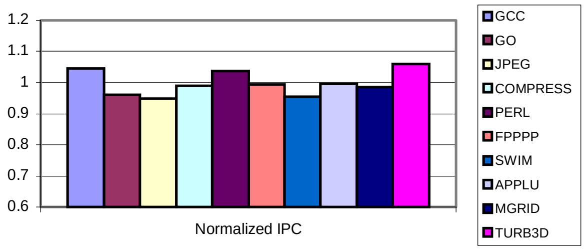

Figure 8. IPC variation for the 14-stage pipelines (100k trace-cache)

For the longer pipeline (14 stages), the IPC for both structures is smaller, due to the increased mispredict penalty. Here, the proposed microarchitecture is about 0.3% slower than the normal superscalar one. However, for the case of normalized performance, the trend is similar to the 8-stage case.

From the power consumption point of view, we find a similar situation. For the 0% overhead model, the gain varies between 23% for FPPPP and as much as 52% for TURB3D. Overall, the power consumption per committed instruction is reduced by 20% for the 5% overhead model or by as much as 40% if the more aggressive, 0% overhead assumption is considered.

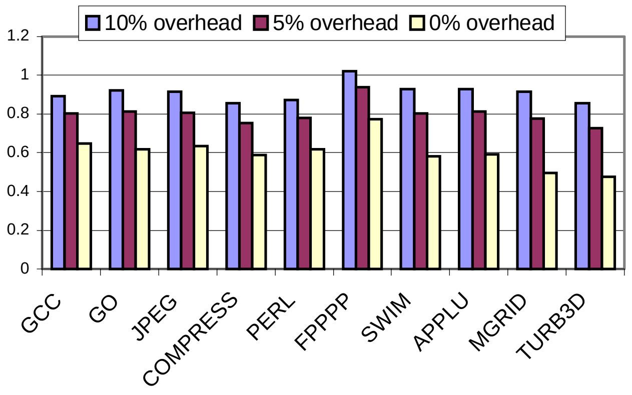

Figure 9. Normalized EPI for a 14-stage pipeline (100k trace-cache)

Increasing the trace-cache size to 200k increases its usage and thus its efficiency. We run a test with this new configuration and the results were significantly better. The alternative execution path was used 76% of the total execution time and the IPC increased to 8% over the normal microarchitecture.

Simulating again all the benchmarks for the new trace-cache (on the 8-stage pipeline), the usage of the alternative execution path was better for some applications, but other cannot benefit from the larger cache.

An interesting case in all these tests is FPPP. We have performed several tests in different configuration and discovered that this benchmark is composed from one big basic block. If we double the trace-cache size (up to 200k), the alternative path usage is much better. In the case of FPPP, if the 100k size trace-cache is used as in the rest of the test cases, the processor issues instructions from the trace-cache only 40% of the time (compared to up to 99% of the time for TURB3D).

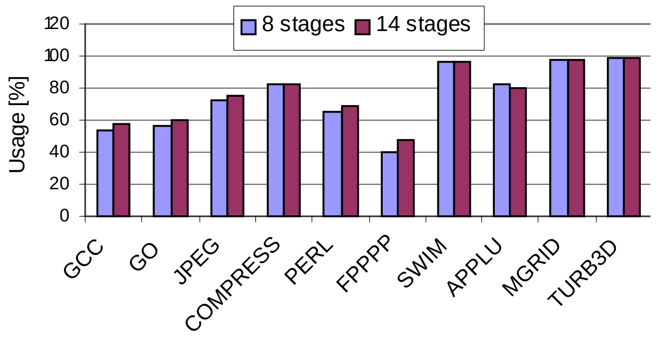

Figure 10. Alternative execution path usage (100k trace-cache)

In such situations, the trace-cache may become inefficient since it is not capable to accommodate the entire basic block. As seen in Figure 10, for most of the benchmarks instructions are issued from the trace-cache more than 60% of the time, whereas in case of FPPP the trace-cache usage is only 40%.

For FPPP, the energy per instruction computed with respect to the normal microarchitecture dropped to 90.9% (10% overhead), 80.5% (5% overhead) and 61.3 (0% overhead).

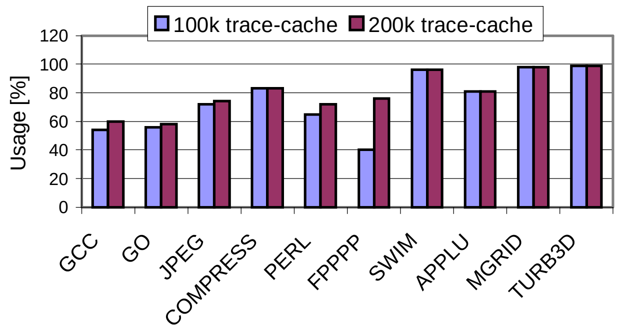

Figure 11. Comparative view of the alternative execution path

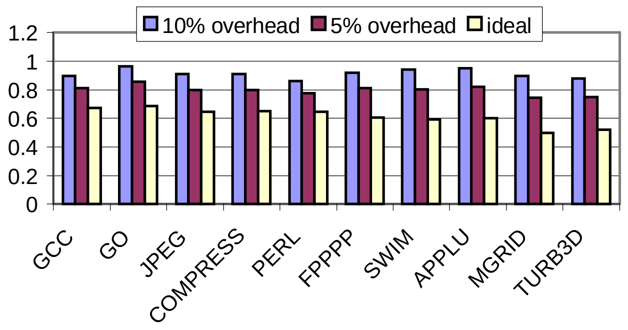

Figure 12. Normalized EPI for an 8-stage pipeline (200k trace-cache)

Overall, we obtained a slight increase in IPC (1.5% over the smaller trace-cache version) but a slight decrease in energy efficiency (1%).

8. CONCLUSION AND FUTURE WORK

In this paper, we propose a new microarchitecture that yields comparable performance to the usual superscalar microarchitecture while using significantly less power. Our experiments show that, depending on the overhead model considered for the unused modules, by using this approach we can achieve between 20% and 40% reduction in power consumption.

Experimentally, we noticed that there are a few ways we can tune the design for power or performance. For example, by further decreasing the length of the traces, the number of instructions executed per cycle (IPC) decrease by about 3-5% while the energy required per instruction (EPI) decreases also by as much as 10% below the values we have presented. Furthermore, if we don't update the branch history buffer while executing from the trace-cache, we have a drop in IPC of about 2% and a decrease in EPI of 5%. In the experiments we presented here, we tuned our test microarchitecture for a performance within 3% from the equivalent superscalar architecture.

A drawback of our trace-cache organization is the relative inefficiency in the space usage. As we want to be able to store and retrieve the instructions as fast as possible, we are storing them in the trace cache exactly as they come out from the issue window. In this respect, on a 4-way processor we will try to store as much as 4 instructions in every entry of the data array. If during a clock cycle the processor is not able to find and issue 4 independent instructions, this will result in some empty slots in the trace. So, the overall usage of the trace cache (in terms of memory space) could be below 100%. In a future implementation, we will try to come up with a way to compress the instructions for a better efficiency in the data array usage.

9. ACKNOWLEDGEMENTS

The authors would like to thank John Shen of Carnegie Mellon University/Intel Corp. for many stimulating discussions, as well as for offering many insights into the TurboScalar microarchitecture.

10. REFERENCES

[1] B. Black and J. P. Shen – "*TurboScalar: A High Frequency, High IPC Microarchitecture*" - International Symposium on Computer Architecture, June 2000

[2] B. Black, B. Rychlik, J. P. Shen – "*The Block-based Trace Cache*" - International Symposium on Computer Architecture, May 1999

[3] B. Black, J. P. Shen - "*Scalable Register Renaming via the Quack Register File*" – Technical Report CMuART-2000-01

[4] E. Rotenberg, S. Bennett, J.E.Smith – "*A trace Cache Microarchitecture and Evaluation*" - IEEE Trans. on Computers, February 1999

[5] V. Tiwari, S. Malik, P. Ashar – "*Guarded Evaluation: Pushing Power Management to Logic Synthesis / Design*" - International Symposium on Low Power Design, April 1995

[6] V. Tiwari, D. Sigh, S Rajgopal – "*Reducing Power in High-performance Microprocessors*" – Design Automation Conference, June 1998

[7] T. D. Burd, R. W. Brodersen – "*Energy Efficient CMOS Microprocessor Design*" - 28th Hawaii International Conference on System Sciences, Jan. 1995

[8] Kin, J.; Munish Gupta; Mangione-Smith, W.H - "*The filter cache: an energy efficient memory structure*" - IEEE Micro, December 1997

[9] N. Bellas, I Hajj - *Architectural and Compiler Techniques for Energy Reduction in High Performance Processors*" - International Symposium on Low Power Electronics Design, August 1998

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

[11] T. Austin, "*The SimpleScalar Architectural Research Tool Set, Version 2.0*" - Computer Sciences Technical Report, June 1997

[12] D. Brooks, V. Tiwari, M. Martonosi – "*Wattch: A Framework for Architectural-Level Power Analysis and Optimizations*" - International Symposium on Computer Architecture, June 2000

[13] INTEL Corp – US Patent US6170038 "*Trace based instruction caching*"

[14] Ghose, K.; Kamble, M.B. - "*Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation*" - International Symposium on Low Power Electronics and Design, July 1999.

[15] SPEC Benchmarks - www.spec.com

[16] Pentium 4 Microarchitecture – P. De Mone – http://www.realworldtech.com