# Potential of using a reconfigurable system on a superscalar core for ILP improvements

Marcelo Brandalero, Antonio Carlos Schneider Beck
Universidade Federal do Rio Grande do Sul
Instituto de Informática - Av. Bento Gonçalves, 9500
Campus do Vale - Porto Alegre, Brasil
{mbrandalero, caco}@inf.ufrgs.br

***Abstract* — *As technology scaling reduces pace and energy efficiency becomes a new important design constraint, superscalar processor designs seem to be reaching their performance limits under the area and power constraints. As a result, new architectural paradigms have to be developed. This work proposes a new architecture for x86 processors, based on a traditional superscalar design coupled to a reconfigurable array. The architecture explores the fact that few basic blocks are responsible for most of the instructions that execute on the processor, and performs a mapping of these basic blocks into a configuration for the reconfigurable array. The configuration encodes the dependencies between the instructions, so that when a loop is executed multiple times, fetch, decode and dependency checks on the instructions are bypassed, thus improving instruction throughput. Our study of the potential of the architecture shows that performance gains of up to 2.5 times with respect to a traditional superscalar can be presented.***

***Keywords* — *x86; instruction-level parallelism; reconfigurable architectures;***

## I. INTRODUCTION

The growing demand for more performance on computer systems has been challenging processor designers to develop solutions that reach beyond traditional architectures. Energy efficiency is finally becoming a first order design constraint for all market segments: embedded systems need to present low power to preserve battery life, general-purpose processors are designed with strict thermal dissipation power limitations and even processors for high-end servers are being optimized for energy efficiency to fit in the Green Computing concept. This power limitation restricts the use of some architectural solutions that optimize performance. Besides, technology scaling, which has been one of the major drivers for performance improvements over the last 20 years, is reaching its limits. Improved performance and energy efficiency must come, therefore, from technological advances in processor microarchitecture [1] [2] [3].

The key to achieving more performance is to efficiently exploit on chip the parallelism available from software. One fundamental form of parallelism is Instruction Level Parallelism (ILP), which reflects how often processor instructions can be executed concurrently. Even though there is an upper bound to the amount of ILP available from applications, given by data dependencies which are a natural part of any computation [4], this bound can hardly be reached by modern processor designs, as it comes to a point at which the marginal increases in area and power do not make up for the gains in performance. As some studies suggest, performance of single-threaded applications will increase very little in the following decades, due to the aforementioned discussion [3].

In order to improve performance and reduce energy consumption, most modern processors employ methods to reuse parts of computation that were previously performed. Recent generations of Intel processors, for instance, have been employing a method to exploit recurring loops in code. The Loop Stream Detector (LSD) [5][6] is an instruction cache for loops, located inside the processor pipeline. On the first loop execution, the cache is filled with the instructions; on the subsequent executions, the instructions are streamed directly from this cache. By skipping the first pipeline stages, performance gains are achieved. Figure 1 illustrates this concept. In the Core2 microarchitecture, the LSD was placed after the fetch stage; in the Core i (Nehalem) microarchitecture, it was placed after the decode stage. Our system aims to take this approach one step further and caches the entire dynamic scheduling of the instructions in the loops, i.e. its dependencies, register allocations and order of execution within the basic block.

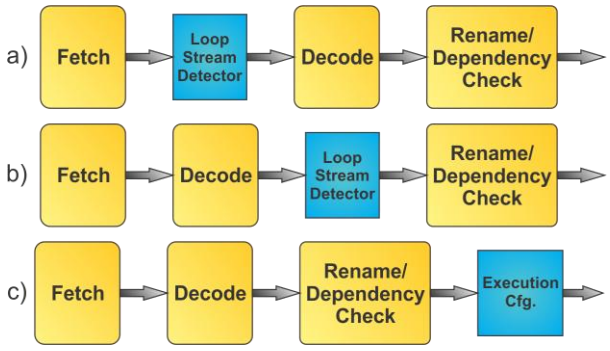


**Figure 1. Loop Stream Detector in the (a) Core2 architecture and the (b) Core i architectures, and (c) the approach proposed in this work.**

To exploit this concept, we consider the use of reconfigurable systems, because they have already been shown to be a promising approach to improving both performance and energy efficiency for a variety of applications [7]. A reconfigurable system employs, along the processor, a circuit whose function can be dynamically modified, such as a Field Programmable Gate Array (FPGA) or an array of functional units. This circuit implements the hotspots in code (i.e. code that is frequently executed, such as loops) using combinatorial logic. These code sequences can then be executed on the circuit, providing energy savings by eliminating intermediate registers and dependency-checking between instructions.

In this work, we present a new microarchitecture for x86 processors that uses the ideas of the Loop Stream Detector and reconfigurable computing, targeted towards improved performance as well as energy savings. We use the micro-ops generated by the x86 instruction decoder as input to a binary translation mechanism, which performs the mapping of the micro-ops into a configuration for a reconfigurable array. By caching decoded instructions ahead on the pipeline, the need to fetch and decode instructions for the same code sequence is eliminated, providing performance gains and energy savings. The proposed mechanism excels over the LSD because cached configurations also store the dependencies between instructions, avoiding the need to repeatedly check for them when a loop executes.

This paper proceeds as follows. In section 2, a review of work regarding instruction level parallelism and reconfigurable computing, as well as characteristics of x86 processors is presented. In section 3, we present the proposed organization for the system, providing a brief overview of the reconfigurable system. Section 4 presents results on its potential for performance improvements, comparing the results achieved with the performance of a traditional superscalar architecture. Section 5 discusses further work to be done on the architecture and concludes this paper.

## II. Related work

Early studies on instruction level parallelism have determined there are upper bounds on the amount of parallelism available from applications. Wall [4] presents a study on these limits for a given set of applications and an architecture. It is shown that the limits of ILP could be as high as 50 instructions per cycle; on a real processor, however, this bound is much stricter than that. We conduct later on this paper an experiment similar to the one of Wall, but considering also the possibility to execute multiple dependent ALU instructions in one cycle. This is made possible by replacing traditional, sequential code execution with combinatorial logic.

With respect to reconfigurable computing, vast literature has been produced [8]. It is a concept that fills a gap in computation: it is typically unfeasible to achieve high performance and simultaneously provide flexibility. Hardwired solutions, such as application-specific integrated circuits (ASICs), provide high performance but need to be totally redesigned for each different application. Microprocessors, on the other hand, serve a wide variety of applications but lack the performance provided by an ASIC. Reconfigurable systems can be configured at runtime to better suit the application to be run; better performance is achieved than with microprocessors, while still providing a higher flexibility than with ASICs. A simple example of a reconfigurable system is one composed of a microprocessor coupled to an FPGA, which the processor can program and use for execution. A survey on aspects of reconfigurable computing is presented by Compton and Hauck in [9]. System classification with respect to processor coupling, reconfiguration times and granularity of the execution units are discussed in their work, but no experimentation is performed.

Most studies on reconfigurable computing express the need to determine critical parts of computation that are to be mapped into hardware during the application development phase. This approach is named static discovery, and requires the use of special compilers. Using methods such as binary translation [10] [11], it is possible to perform this mapping dynamically at runtime. This way, backwards binary compatibility can be achieved, which is a key design issue when further developing an architectural family, and one of the reasons behind the success of the x86 architecture.

Our interest lies within systems that allow the dynamic discovery of instructions, because they maintain binary compatibility with code already deployed. Lysecki, Stitt and Vahid [12] present one of the first works in the field, defining a new design named warp processor, in which an application binary's critical regions are dynamically determined at runtime and mapped into a custom hardware circuit in an FPGA. The hardware must include a special processor that runs a simplified CAD algorithm to perform the mapping of critical regions to the FPGA. Clark et al. [13] present the use of the Configurable Compute Accelerator (CCA), a specialized unit that optimizes the execution of critical computation sections determined from an application's dataflow graph. The CCA is organized as a matrix of functional units, since this is a natural way of exploring both instruction level parallelism and the propagation of data between functional units. The paper discusses ways to integrate the reconfigurable fabric into the processor and presents performance results when using static or dynamic subgraph discovery.

Beck et al. [7] [14] present the use of a coarse-grained reconfigurable array tightly coupled to a MIPS processor. Performance improvements of up to 2.5 times were achieved, while presenting energy reductions and maintaining backwards compatibility with respect to the MIPS code. This approach requires the underlying ISA to provide simple instructions, such as the one provided by MIPS. For the proposed system to work with other architectures, extensions have to be made. On Fajardo et. al. [15], a two-level binary translation system is used to transform x86 code into MIPS code and then optimize it for execution on the reconfigurable array. The goal of that system is to provide support for multiple architectures with binary compatibility, and therefore presents no performance or energy gains.

As for the x86 architecture, we address three of its characteristics that are explored by our system. X86 is a CISC architecture (Complex Instruction Set Computing), meaning that multiple low level operations, such as memory accesses followed by arithmetic operations, can be encoded within a

single instruction. In contrast with RISC instructions (Reduced ISC), CISC instructions are hard to pipeline, because they are usually variable length, each of them performs a different number of operations and operands can reside in memory. To cope with it, x86 processors use a scheme in which CISC instructions are decoded into multiple RISC-like instructions, named micro-ops [16]. Because each micro-op represents a single operation, these are not only simpler to pipeline, but also simpler to map into a reconfigurable array.

One characteristic feature of some families of x86 processors is the presence of a trace cache. A trace cache works similarly to an instruction cache, but rather than caching instructions that are sequential in memory, a trace cache stores entire sequences of basic blocks that appeared sequentially during program execution. When a branch terminating a basic block is biased towards an address, the basic block corresponding to that address is cached on the subsequent block of the trace cache. This way, higher instruction throughput to the decode stage is provided [17]. On the Pentium 4, this concept was further extended such that the trace cache stored micro-ops rather than regular instructions [18]; this way, it alleviates work of the decode unit when executing the same basic block multiple times.

On the latest editions of x86 processors, another characteristic feature has been added. The Loop Stream Detector (LSD) [5] [6] is a mechanism that detects small, recurring loops in code. In case of the Nehalem (Core i) microarchitecture, this mechanism stores the micro-ops that correspond to a loop in a small memory inside the processor pipeline, after the decode stage. When a loop is detected by this mechanism, the fetch and decode pipeline stages are disabled and the instructions are fetched from this new memory, saving time and providing energy savings.

## III. Proposed Architecture

Our architecture exploits the fact that short, recurring loops are very common in software. As shown in [7], there are applications in which a few dozen basic blocks cover more than 90% of the instructions that execute on the processor. When executing these basic blocks, data dependencies are continuously checked between the instructions, even though the same instructions are executed over and over again. In [19], it is shown that these dependency checks are responsible for an average 25% of the energy consumed in processor cores.

Figure 2 presents an overview of the behavior of the system proposed in this work. The blocks on the upper part of each figure represent the typical stages that compose the pipeline for superscalar processors, namely fetch, decode, dispatch, issue, execution and commit. When a basic block is executed for the first time (Figure 2a), after it is fetched from memory its instructions are decoded and executed as usual on the processor pipeline. At the same time, the decoded instructions are fed into a binary translation (BT) mechanism that performs the mapping of the micro-ops into a configuration for the reconfigurable array. This configuration is stored in the configuration cache. When the same basic block is executed again (Figure 2b), the configuration is read from the

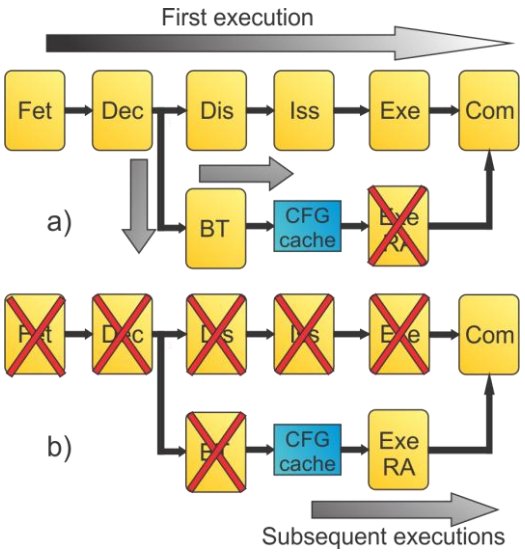


**Figure 2. Execution on the system when a) a basic block is executed for the first time and b) when the basic block is already stored in the configuration cache.**

configuration cache and the basic block is executed on the reconfigurable array. This way, all the logic required to access memory, decode instructions, execute register renaming and dependency checking can be disabled. Instructions continue to stream from the reconfigurable array until a branch instruction with target address outside that basic block is executed. As shown in Figure 1, this mechanism replaces the Loop Stream Detector.

In the next section, we describe in details how the reconfigurable array works, and next we give details of the microarchitecture of the system.

### A. *Reconfigurable Array (RA)*

A general overview of the array organization, as proposed in [7], is presented in Figure 4. The array consists of a matrix of functional units, in which each instruction is allocated to one cell. In this matrix, columns represent parallel execution whereas lines represent sequential execution, or the flow of time. Each level represents one processor cycle; the latencies of the functional units are implementation-dependent. In the figure shown, up to three sequential ALU operations may be performed in one cycle, and up to four ALU operations can be scheduled to each line. Similarly, up to two loads or stores may be performed per cycle and one multiplication operation. An instruction depending on a value produced previously can only be allocated on a row above that of the instruction producing the value.

The input context of the array consists of buses connecting every register to the inputs of the functional units on the first level. Multiplexers are responsible for choosing the appropriate input to each functional unit. Inside each level, multiplexers are also present, and may select as input to each functional unit any of the results on the line below. On the output context, multiplexers select the correct values produced on the last level of the array to be written back to the register bank.

The array has the potential to speed up applications, when compared with a traditional superscalar architecture. Two are the reasons. First, it potentially eliminates functional unit contention, because the operation of each functional unit and the data propagation between them can be modified for each basic block executed. Second, multiple dependent ALU operations can be performed in the same processor cycle, unlike traditional architectures. Because of the flexibility provided, the use of a RA on general purpose systems can increase the average performance for every application, unlike the use of an ASIC.

### *B. Coupling the RA to the x86 processor*

The microarchitecture of our system is composed of the x86 processor, with the RA tightly coupled to the pipeline as another functional unit. Our model of the superscalar pipeline is based on the architectural simulator we used in our performance evaluations, Multi2Sim [20]. The microarchitecture is shown in Figure 3. This model for the x86 pipeline is composed of 6 stages: instruction fetch, decode, dispatch, issue, write-back and commit. On the fetch stage, x86 instructions are read from the instruction cache and passed on to the decode stage. On the decode stage, complex x86 instructions are decoded into micro-ops and put in a queue for the dispatch stage. At the same time, these instructions are fed into a binary translation mechanism, which performs a mapping of the micro-ops into a configuration for the RA. Once a branch instruction is found, the translation is terminated and the configuration is saved in the configuration cache, which replaces the trace cache. With this replacement, we expect the area overhead due to the addition of the

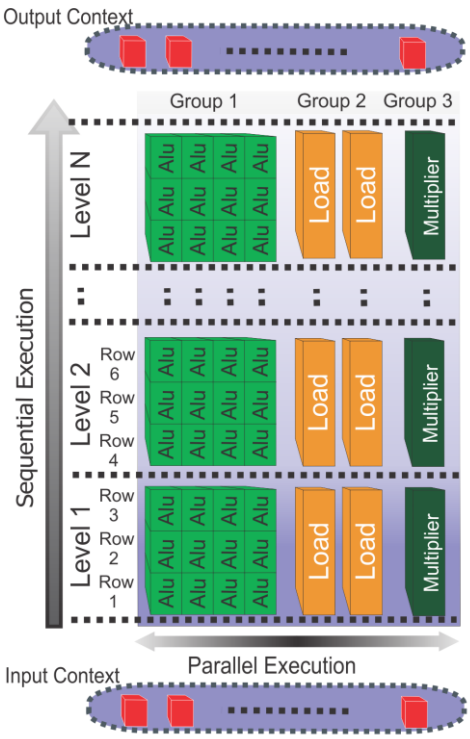


**Figure 4. Overview of the reconfigurable array.**

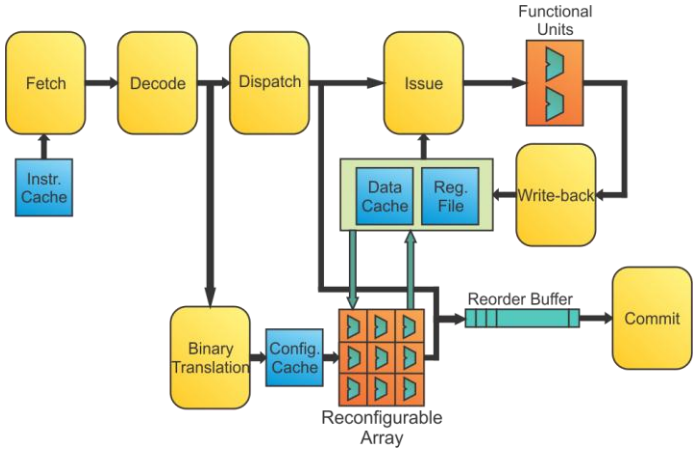


**Figure 3. Overview of the microarchitecture.**

configuration cache to be minimal. The configurations are indexed by the memory address of the first instruction in the basic block.

Execution of the micro-ops generated in the decode stage proceeds normally through the dispatch, issue, writeback and commit stages. On the dispatch stage, false dependencies between micro-ops are eliminated and the micro-ops are fed into the reorder buffer, as well as onto two queues: one for micro-ops performing memory accesses and one for all other operations. On the issue stage, a certain number of operations are executed on the functional units, considering the true data dependencies and the availability of the functional units. On the writeback stage, results are written to the register file or data cache. Finally, on the commit stage, the instructions are removed from the reorder buffer as soon as they are confirmed to be non-speculative.

When a branch instruction is executed and its target address is a basic block which is already in the configuration cache, then the fetch, decode and dispatch stages are disabled and the configuration is loaded to the array. The input operands are fetched from the register file or data cache, the instructions are placed in the reorder buffer and executed. When the instructions are confirmed to be non-speculative, they are ready to commit and are removed from the reorder buffer.

## IV. Speedup Potential

We present a study on potential of the proposed system for performance improvements. The goal is to compare the performance of the microarchitecture proposed in this work with that of a typical x86 processor. To achieve this goal, we chose a set of benchmarks and estimated the average number of instructions executed per clock cycle (IPC) for each application in the suite for both architectures.

As our benchmark suite, we chose MiBench [21] because it covers a variety of applications, both control- and data-oriented. Each of the benchmarks was compiled on a Linux operating system using gcc v4.4 with -static, -O3 and -m32 flags. The benchmarks were executed on two simulators: one modeling the x86 processor (Multi2Sim [20]) and the second modeling our system.

**Table 1. Different setups considered for the RA.**

| Parameters | Setups | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| ALU op. latency (cycles) | 1 | 1/2 | 1/3 | 1/3 | 1/3 | 1/3 |
| Mem. ops. per cycle | Unlim. | Unlim. | Unlim. | 4 | 2 | 1 |

To evaluate the potentials of our architecture, we developed a simulator for the architecture which extended the Multi2Sim simulator. The extended simulator implements an instruction scheduler for the RA. We executed the applications on Multi2Sim, which generated an execution trace of micro-ops for each application. This trace is fed into our simulator which reads entire basic blocks from the trace and schedules them for execution. The simulator assumes there is an infinite amount of functional units available, and that every instruction can be executed on the RA; this way, instructions are scheduled based on true data-dependencies only, and we can get an upper bound on the potential of using the array for performance improvements. We considered multiple setups, in which two parameters were varied: the latency of each ALU instruction (i.e.: how many dependent ALU instructions can be executed in one cycle), and the number of memory operations that can be executed per cycle. These setups are presented in Table 1.

In Table 2, the average IPC observed for each benchmark executing on the RA is presented, under the six setups. In this experiment, only one basic block can be executed at a time on the RA, thus ignoring speculative execution. For setups 1 to 3, one can notice the increase in IPC that is obtained when allowing multiple ALU operations to be executed per cycle. An average of 10% increase in IPC is observed when allowing up to three ALU instructions to be executed per cycle. Little benefit should be expected from expanding this value further. As for setups 4 to 6, one can notice the huge impact of allowing only a small number of memory operations to execute within the same clock cycle. When comparing setup 5 with 3, average IPC goes down 15% and when comparing setup 6 with 3, the average decrease is of 30%. This table provides a good insight into the memory behavior of each application, as well as into the amount of computation they perform. However, it is not suitable for comparison with a superscalar architecture, since we model in this experiment only the execution unit, rather than the entire processor pipeline. Also, the assumptions made for each system are different.

To compare our results with execution on the superscalar processor, we change a few assumptions about both systems in order to put them on the same baseline for comparison. First, since our simulator does not consider memory access latency, we configured Multi2Sim such that every memory results in a cache hit. Second, Multi2Sim, as most superscalar processors, performs speculative execution. On the previous experiment, only one basic block could execute at a time, therefore ignoring speculative execution. We must consider additional setups in the RA on which multiple basic blocks may be executed simultaneously. Because we ignore reconfiguration times on the RA and work with execution traces, we also configured the branch prediction scheme on Multi2Sim to always hit. Multi2Sim was configured to issue up to 4 memory instructions and 4 non-memory instructions per cycle; to match this, the RA setup taken as base for this comparison was setup 4, with 4 memory instructions per cycle and up to three dependent ALU operations per cycle.

Figure 5 presents a comparison of the results on the RA and on the superscalar simulators, considering the aforementioned discussion. The average IPC values obtained for execution on the RA were normalized with respect to IPC from execution on Multi2Sim (superscalar); values higher than one indicate, therefore, performance gains over the superscalar model. As can be seen, for most applications no gain is provided when only one basic block is allowed to execute at a time. This is expected, because the superscalar processor is executing multiple basic blocks simultaneously. As we increase the amount of speculation performed on the array, by increasing the amount of basic blocks executed at a time, performance gains start to show up. When speculating up to two basic blocks, 8 out of 20 applications already present performance gains, with an average normalized IPC value for the entire benchmark set of 1.07. As we further increase the amount of BBs speculated from 3 till up to 5, the average normalized IPC values are of 1.32, 1.53 and 1.68. When speculating up to 5 BBs, all applications present performance gains, with some applications, such as susan and jpeg decoder, performing twice as fast as the superscalar processor. It should be noted that considering the reorder buffer on the superscalar simulator was taken large (100+ positions), and assuming basic blocks take an average size of 13 micro-ops for the tests performed, then the

**Table 2. IPC for each benchmark executing on the RA, considering the six different setups.**

| Benchmarks | Array (different setups) | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 |
| adpcm enc | 1.67 | 1.71 | 1.71 | 1.71 | 1.69 | 1.56 |
| adpcm dec | 1.69 | 1.76 | 1.76 | 1.76 | 1.71 | 1.54 |
| basicmath | 2.24 | 2.47 | 2.51 | 2.49 | 2.33 | 1.95 |
| bitcount | 1.94 | 2.26 | 2.27 | 2.27 | 2.25 | 2.09 |
| blowfish enc | 1.86 | 2.02 | 2.07 | 2.06 | 2.00 | 1.74 |
| blowfish dec | 1.88 | 2.04 | 2.09 | 2.08 | 2.01 | 1.75 |
| CRC32 | 1.85 | 2.02 | 2.02 | 2.02 | 1.95 | 1.64 |
| Dijkstra | 1.32 | 1.42 | 1.43 | 1.42 | 1.42 | 1.39 |
| FFT | 2.47 | 2.71 | 2.76 | 2.73 | 2.50 | 2.02 |
| FFT inv | 2.38 | 2.63 | 2.67 | 2.65 | 2.48 | 2.06 |
| gsm enc | 2.69 | 3.02 | 3.12 | 2.97 | 2.66 | 2.16 |
| gsm dec | 1.63 | 1.72 | 1.73 | 1.73 | 1.67 | 1.53 |
| jpeg enc | 2.35 | 2.46 | 2.46 | 2.32 | 2.14 | 1.72 |
| jpeg dec | 4.25 | 4.34 | 4.35 | 4.08 | 3.38 | 2.19 |
| patricia | 2.19 | 2.39 | 2.45 | 2.42 | 2.24 | 1.84 |
| qsort | 2.21 | 2.36 | 2.39 | 2.34 | 2.19 | 1.82 |
| stringsearch | 1.90 | 2.24 | 2.29 | 2.28 | 2.18 | 1.95 |
| susan corners | 3.84 | 4.35 | 4.48 | 3.89 | 3.14 | 2.14 |
| susan edges | 5.96 | 6.92 | 7.21 | 5.31 | 3.74 | 2.28 |
| susan smoothing | 2.69 | 2.94 | 2.95 | 2.93 | 2.90 | 2.38 |
| **Average** | **2,45** | **2,69** | **2,73** | **2,57** | **2,33** | **1,89** |

consideration that up to 5 basic blocks can execute simultaneously on the RA is a reasonable assumption for this comparison.

## V. Conclusion

In this work, we presented a new architecture for x86 processors which aims to improve overall performance for all applications and provide energy savings. On our preliminary study, performance gains of up to 2.5x for some applications may be achieved, with an average gain of 1.68x. Currently, we are working on a prototype for the system, which will be used to take area and power estimations. We shall then analyze performance, area and power and compare all parameters with the superscalar architecture, in order to evaluate implementation tradeoffs.

## VI. References


[1] M. J. Flynn and P. Hung, "Microprocessor Design Issues: Thoughts on the Road Ahead," *IEEE Micro*, vol. 25, no. 3, pp. 16–31, May 2005.

[2] K. Olukotun and L. Hammond, "The future of microprocessors," *Queue*, vol. 3, no. 7, p. 26, Sep. 2005.

[3] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, p. 67, May 2011.

[4] D. W. Wall, "Limits of instruction-level parallelism," *ACM SIGPLAN Not.*, vol. 26, no. 4, pp. 176–188, Apr. 1991.

[5] M. Dixon, P. Hammarlund, S. Jourdan, and R. Singhal, "The Next Generation Intel® Core™ Microarchitecture," *Intel Technol. J.*, vol. 14, no. 3, pp. 8–28, 2010.

[6] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual." Intel, p. 660, 2014.

[7] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Proceedings of the conference on Design, automation and test in Europe - DATE '08*, 2008, p. 1208.

[8] A. C. S. Beck and L. Carro, *Dynamic Reconfigurable Architectures and Transparent Optimization Techniques*. Springer, 2010, p. 225.

[9] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, Jun. 2002.

[10] E. R. Altman, D. Kaeli, and Y. Sheffer, "Welcome to the opportunities of binary translation," *Computer (Long. Beach. Calif).*, vol. 33, no. 3, pp. 40–45, Mar. 2000.

[11] A. C. S. Beck, C. A. L. Lisba, and L. Carro, "Adaptable Embedded Systems," Nov. 2012.

[12] R. Lysecky, G. Stitt, and F. Vahid, "Warp Processors," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, no. 3, pp. 659–681, Jul. 2006.

[13] N. Clark, M. Kudlur, S. Mahlke, and K. Flautner, "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, 2004, pp. 30–40.

[14] A. C. S. Beck, M. B. Rutzig, and L. Carro, "A transparent and adaptive reconfigurable system," *Microprocess. Microsyst.*, vol. 38, no. 5, pp. 509–524, Jul. 2014.

[15] J. Fajardo, M. B. Rutzig, L. Carro, and A. C. S. Beck, "Towards a multiple-ISA embedded system," *J. Syst. Archit.*, vol. 59, no. 2, pp. 103–119, Feb. 2013.

[16] J. L. Henessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011, p. 856.

[17] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," pp. 24–35, Dec. 1996.

[18] I. C. Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Desktop Platforms Group, "The Microarchitecture of the Pentium 4 Processor."

[19] D. Folegnani and A. Gonzalez, "Energy-effective issue logic," in *Proceedings 28th Annual International Symposium on Computer Architecture*, 2001, pp. 230–239.

[20] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: a simulation framework for CPU-GPU computing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT '12*, 2012, p. 335.

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.


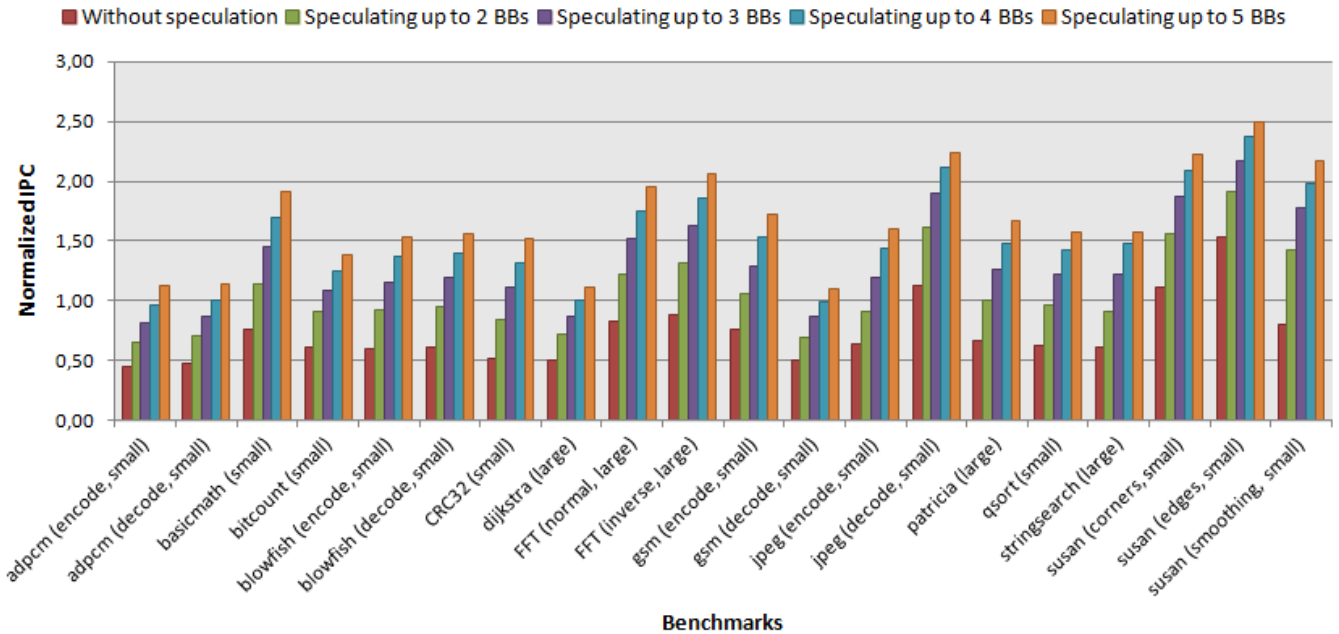


**Figure 5. IPC for different benchmarks executing on the RA, considering speculation, normalized with respect to execution on superscalar processor.**