UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

MARCELO BRANDALERO

A reconfigurable array for superscalar processors

Monograph presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering.

Advisor: Prof. Dr. Antonio Carlos S. Beck Filho

Porto Alegre

2014

ACKNOWLEDGEMENTS

I would like to thank everyone who contributed for the development of this project, directly or indirectly, and also those who supported me during my undergrad studies.

A special thanks to those contributing indirectly to the work: to my mom, Veronice, for always being there for me in the good and in the difficult times; to my dad, Luiz, for always pushing me forward in my studies; to my girlfriend, Amanda, for all her understanding and support; and all friends who made my university years an enjoyable experience.

A special thanks for those whose contributions were essential to the development of this work: to my advisor, Prof. Caco, for all his constructive criticism, incentives and support during the last two years; to all my friends at the embedded systems lab.

ABSTRACT

The performance of microprocessors is closely related to their ability to exploit the parallelism from applications. The superscalar model has long been the state-of-the-industry microarchitectural paradigm for exploiting instruction-level parallelism; however, they reach their scalability limits under the strict area and power constraints posed by modern designs. This work proposes a new microarchitecture for x86 processors, based on a traditional superscalar design tightly-coupled to a reconfigurable array. The array implements critical computation parts using combinational logic, improving the amount of parallelism exploited. The system detects recurring code sequences at runtime and employs dynamic binary translation to prepare these sequences for execution on the reconfigurable array; the next time the code sequence has to execute, the array is employed. Two major advantages of this solution are that it is transparent to the programmers, because binary compatibility is maintained, and it is simpler to implement (compared to other novel microarchitecture solutions), because it is based on a traditional superscalar design. Additionally, by targeting the x86 architecture, one additional advantage emerges: the burden on the x86 instruction decoder, which has to constantly translated CISC instructions into simpler micro-ops, is alleviated. The microarchitecture was modeled using a cycle-accurate simulator and performance results were collected. It is shown that the proposed system presents higher potential to explore instruction-level parallelism than the superscalar.

Keywords: x86. Instruction-level parallelism. Trace-level reuse. Reconfigurable architectures. Binary translation.

Um array reconfigurável para processadores superescalares

RESUMO

A performance de microprocessadores está intimamente relacionada à sua capacidade de explorar o paralelismo presente nas aplicações. O modelo superescalar tem sido, por muito tempo, o estado-da-industria em termos de paradigma microarquitetural; contudo, sob as restrições de área e potência impostos pelos projetos atuais, eles atingem seus limites de escalabilidade. Este trabalho propõe uma nova microarquitetura para processadores x86, baseado em um sistema superscalar ao qual um array reconfigurável é acoplado. O array implementa trechos críticos da computação utilizando lógica combinacional, o que aumenta a quantidade de paralelismo explorado. O sistema detecta trechos recorrentes de código em tempo de execução e utiliza tradução binária dinâmica para preparar esses trechos para execução no array reconfigurável; na próxima vez que o trecho precisar ser executado, o array é utilizado. Duas vantagens dessa solução são que ela é transparente para os programadores, pois é mantida a compatibilidade binária, e ela é simples de ser implementada (frente a outras soluções microarquiteturais), pois é baseada em um projeto superescalar. Adicionalmente, por utilizar-se a ISA x86, surge uma outra vantagem: a pressão em cima do decodificador, que necessita constantemente transformar instruções CISC em micro-ops, pode ser reduzida. O sistema foi modelado utilizando um simulador com precisão de ciclos, e resultados de performance foram coletados. Observa-se que o sistema apresenta maior potencial de exploração de paralelismo a nível de instruções que o superescalar.

Palavras-chave: x86. Paralelismo a nível de instruções. Reúso de traços de execução. Arquiteturas reconfiguráveis. Tradução binária.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ALU	Arithmetic-logic unit
ASIC	Application-specific integrated circuit
BB	Basic block
BT	Binary translation
CAD	Computer-aided design
CCA	Configurable compute accelerator
CISC	Complex instruction-set computing
FPGA	Field-programmable gate array
ILP	Instruction-level parallelism
IP	Instruction pointer
ISA	Instruction-set architecture
LSD	Loop stream detector
RA	Reconfigurable array
RISC	Reduced instruction-set computing
ROB	Reorder buffer
TDP	Thermal design power
TLP	Thread-level parallelism

SUMMARY

1 INTRODUCTION

The rapid growth of transistor integration density within processors, as predicted by Moore's law (Moore, 1965), presents a challenge to architectural designers. When a new process generation is to be launched, designers must choose the best possible way to utilize the available die area in order to match the processor performance with the users' expectations. Besides area, energy efficiency has recently become an important design constraint for all market segments, because processors are designed with a TDP (Thermal Design Power) limitation. This poses a serious limitation to the employment of techniques that improve performance, because these usually require additional area and energy. However, the strongest constraint in architectural development is that of maintaining binary compatibility, which allows applications that were previously deployed to continue executing on a new processor without the need to recompile, retest and redeploy. Thus, new processors must be built upon an ISA (Instruction Set Architecture) that extends the previous one, and most improvements over an existing processor generation need be implemented at the microarchitectural level.

The best way to exemplify the aforementioned discussion is by considering the Intel x86 processor family, which dominates in the general-purpose computer systems domain and has been a market leader in that segment for over 30 years. In this time, processor performance has improved more than 1000 times, in such a way most of the architectural improvements were purposely made transparent to the users (Olukotun & Hammond, 2005).

To improve performance, architectural solutions rely on better exploiting on chip the parallelism available from software. Applications present, by construction, parallelism at different levels, such as instruction and thread. Performance is constrained both by the inherent parallelism that the application presents and the hardware features that are implemented to exploit it. One of the fundamental forms of parallelism is ILP (Instruction Level Parallelism), which reflects how often program instructions can be executed concurrently. To exploit ILP, most processor families employ a superscalar microarchitecture. In this microarchitecture, functional units are replicated in the execution stage of the pipeline in order to exploit the parallelism that the application presents. Although there is a theoretical upper bound on the amount of ILP that can be exploited by hardware, given by the application itself, this bound cannot be reached by superscalar designs, as it comes to a point in the design space in which a marginal increase in performance implies great area and power overheads. According to Olukotun & Hammond (2005), the complexity of the additional logic required

to determine parallelism among instructions is roughly proportional to the square of the amount of instructions that can be executed concurrently, which poses a serious scalability problem.

Processors in the x86 family employ superscalar designs, and thus have also reached the limits on ILP. Besides the aforementioned problem, the x86 architecture suffers from one additional drawback: it is a CISC (Complex Instruction-Set Computing) architecture. In such an architecture, instructions may present variable-length encodings and many different memory access modes (direct and indirect, for instance), all of which make them hard (or even impossible) to pipeline. These processors implement a scheme in which these complex instructions are internally decoded into simpler, RISC-like (Reduced Instruction-Set Computing) instructions, named micro-ops, which are then executed on the processor. This decode process is cumbersome; however, since binary compatibility must be maintained, it must be implemented in every processor.

In this work, a new microarchitecture for x86 processors is presented, which provides means to increase the amount of ILP exploited and decrease the burden of decoding CISC instructions, while leaving the ISA unchanged. The solution is targeted towards the x86 architecture, because of its practical relevance as the dominant architecture in the general-purpose computing domain, and makes use of a reconfigurable array.

The monograph is organized as follows: section 2 continues this introduction by motivating the solution towards the problem; section 3 provides a background review on the topics addressed by this work, namely on the limits of ILP exploitation and superscalar processors , reuse techniques and reconfigurable architectures; section 4 presents the microarchitecture proposed; section 5 presents a simulation model of the system, and also a few results; finally, section 6 concludes this work and presents possible future investigation on this theme.

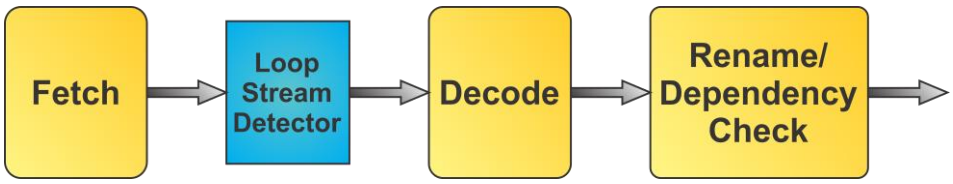
2 MOTIVATION

2.1 General means to improve ILP

One possible approach to improving processor performance is to try to reuse pieces of computation that were previously performed (Sodani & Sohi (1997), Lipasti & Shen (1996), Gonzalez et al. (1999)).

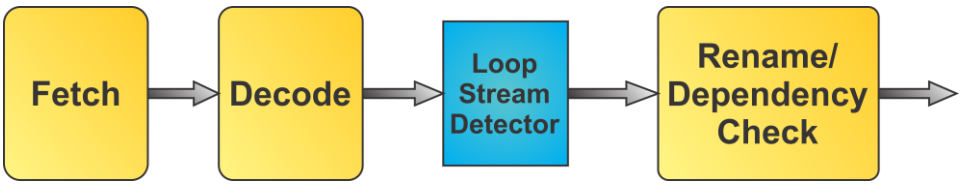
X86 processors, for instance, employ a technique to reduce the instruction fetch and decode overheads when executing instructions in a loop. As noticed by the developers, small loop sequences are very common in software. A typical superscalar processor would be continuously fetching and decoding instructions, even though it is always the same code block being executed. The Intel Core2 microarchitecture introduced the Loop Stream Detector (LSD), a small instruction cache located inside the processor pipeline and capable of holding up to 18 instructions. When a loop executes for the first time, instructions are fetched from memory and the LSD is filled; in the subsequent executions, the instructions stream directly from the LSD, avoiding the fetch stage. In the Intel Nehalem microarchitecture, this concept was taken one step further, by moving the LSD after the decode stage and allowing it to hold up to 28 micro-ops. By doing this, also the overhead of decoding instructions for the same code sequence was reduced (Dixon et al., 2010). Figure 2.1 and Figure 2.2 illustrate this concept in the Intel Core2 and Nehalem microarchitecture, respectively.

Figure 2.1 - Loop Stream Detector in the processor pipeline, as presented in the Core2 microarchitecture.



Source: the author.

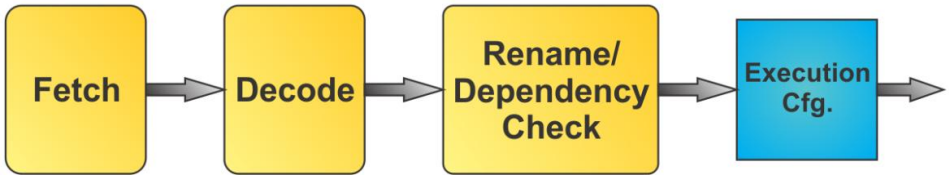
Figure 2.2 - Loop Stream Detector in the processor pipeline, as presented in the Nehalem microarchitecture.



Source: the author.

As mentioned before, one of the limiting factors in further expanding the ILP exploited by superscalar designs is the complexity of the logic responsible for dependency checks, which determines the instructions that can be executed concurrently. Because these dependency checks are continuously performed, even for instruction sequences that have just executed, a lot of redundant computation takes place. One possible solution is to additionally cache the dependency checks for these instructions by moving the LSD ahead in the processor pipeline, after the register renaming and dependency checking stages. This method would solve both the problem of continuously decoding complex x86 instructions and that of checking for parallelism among the instructions. The mechanism is illustrated in Figure 2.3. This information stored is named in the figure as execution configuration, since it stores information on the dynamic scheduling of instructions when they execute.

Figure 2.3 - A mechanism for caching dependency checks amongst instructions.



Source: the author.

By caching this information, there is also a huge potential for energy savings. One important consequence of having less energy consumption is that architects can more freely choose other architectural features that improve performance**,** because processors are usually constrained by the power budget. An energy-efficient design provides more room for designers to pack into the processor features that improve its performance.

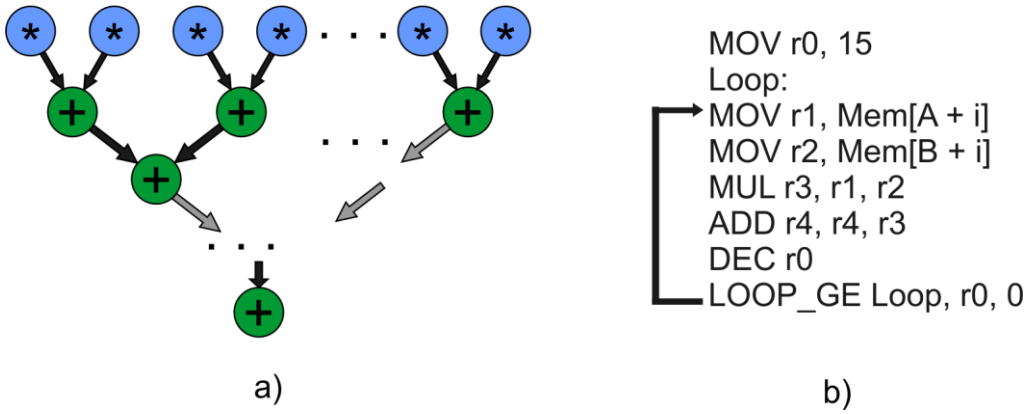
2.2 The data-flow constraint

Besides the burden of decoding instructions, another general limitation of microprocessors comes with the nature of the computation itself. By implementing a design based on sequential logic, these processors force that dependent instructions execute in different clock cycles (a value must first be written in a register before it can be used). This sets a barrier on the maximum ILP that can be extracted from software. In specific fields of application, where high performance or real-time constraints are required, ASICs (Application-Specific Integrated Circuits) are used to overcome this limitation. Rather than

executing instructions, ASICs achieve higher performance by executing a predefined computation, resembling a data-flow execution. This concept is illustrated in Figure 2.4, where (a) shows how computation is performed on an ASIC and (b) how it is performed on a processor, which has to interpret the instructions before performing the operations that they encode. Transforming sequential code execution into combinatorial logic allows the ILP barrier to be broken, because data-dependent instructions can be executed within the same clock cycle. Besides, combinatorial logic provides room for energy savings, because intermediate values need not be stored into registers and the computation is completed sooner.

Although more efficient than microprocessors, ASICs lack the flexibility provided by the former solution. A recent concept which aims to address the gap between these two worlds is that of reconfigurable systems (Compton & Hauck, 2002). By employing a special fabric, configurable at runtime, to execute critical application kernels, these systems mimic the presence of a specialized hardware unit. This fabric could be, for instance, a matrix of functional units where all the interconnects are programmable. These solutions usually require the program kernels to be detected statically and the configuration for the hardware unit to be specified in the program binary. However, the program kernels can also be determined dynamically at runtime, by employing binary translation techniques (Altman et al, 2000). This approach addresses binary compatibility, because there is no need to modify the compiled program.

Figure 2.4 - Data-flow execution and instruction-flow execution of a multiply-accumulate algorithm.



Source: the author.

2.3 This work

In this work, a reconfigurable array with dynamic kernel detection is employed to an x86 core as means to implement the mechanism described in Figure 2.3. The system presents following advantages over existing microarchitectures for x86:

- The cumbersome process of decoding CISC instructions into RISC instructions, as well as dependency checks, are performed only once for these code sequences;
- The execution of these code blocks in combinatorial logic provides room for breaking the ILP barrier which is set by superscalar designs;
- Transparency is provided to the programmers, because there is no need to modify the program binary;
- Little design overhead is present in the system, because it is based on an existing superscalar core.

This is the first work on reconfigurable systems in which an array was coupled to a superscalar processor. The details of how the system works will be explained over the next sections.

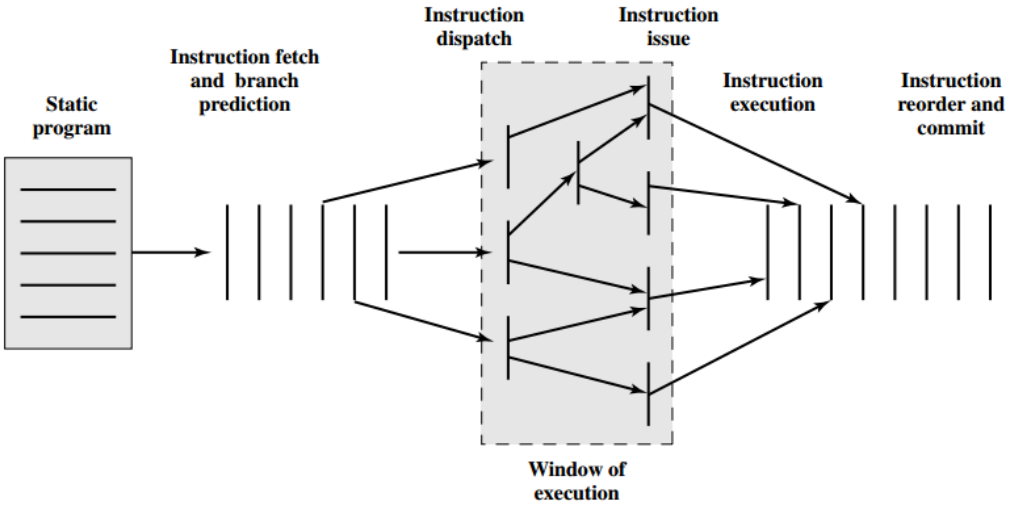
3 BACKGROUND

This section provides a discussion on what is known about the limits of ILP exploitation, some important reuse techniques and finally other reconfigurable systems which inspired this work.

3.1 Limits on ILP

The work by Wall (1991) presents a fundamental study on limits of parallelism available from the applications. It considers five processor models, ranging from a perfect one (perfect branch predictor, perfect memory alias analysis and perfect register renaming) to a bad one (branches always mispredicted, no alias analysis, no register renaming). It is shown that the limit of ILP could be as high as 20 instructions per cycle on the perfect processor, for most of the benchmarks; on a real processor, however, this high ILP cannot be exploited.

Figure 3.1 - Program execution on a superscalar processor.

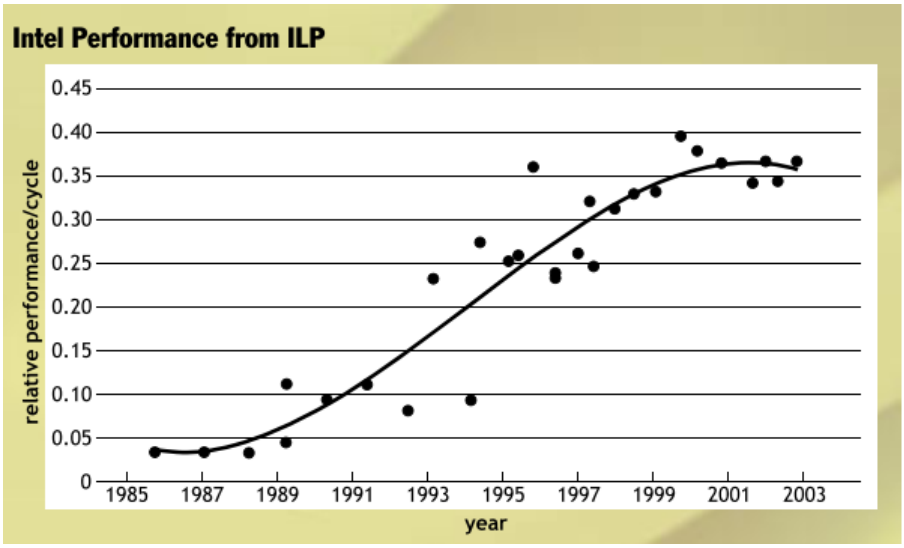


Source: Stallings (2010).

To exploit most of the parallelism that applications present, one microarchitectural paradigm that has been applied to processors for a long time is the superscalar model, as described by Stallings (2010) and shown in Figure 3.1 - Program execution on a superscalar processor.. In this processor model, techniques such as out-of-order execution, register renaming, branch prediction and speculative execution are employed to improve instruction

throughput. These techniques have a high cost, both in area as well as in energy usage. Up to date, even advanced superscalar processors, such as the IBM Power7, do not exceed a limit of 8 instructions per cycle, according to Wendel et al. (2010). The main bottleneck lies within the issue logic, which is responsible for selecting, each cycle, which instructions from the instruction queue may start execution. This selection is based on the true data dependencies between the instructions (i.e., read-after-write dependencies), and thus the input and output operands of all instructions in the queue must be compared one-another. According to Patterson & Henessy (2006), over the years the instruction window size has been maintained in the range of 32 to 126 instructions, which requires over 2000 comparators to check for dependencies. Some processors, such as the Alpha 21264, spend up to half of the total power consumed just with ILP extraction (Wilcox & Manne, 1999). The implications in energy usage were also reported by Folegnani & Gonzalez (2001), who found that up to 25% of energy consumption in processors was being devoted to checking parallelism between instructions.

Figure 3.2 - X86 processor performance associated with ILP extraction over the years.



Source: Olukotun & Hammond (2005).

Despite the high cost in area and energy of superscalar designs, they seem to have become a state-of-the-industry in microarchitectural paradigms and managed to improve processor performance over each new processor generation, up until a limit was reached. Figure 3.2 shows the increase in processor performance related with the exploration of ILP over the years. As superscalar designs started to appear in the 90's, processor performance quickly grew until the curve becomes flat, close to the year 2000. At this stage, marginal

performance improvements implied great area and power overheads; thus, a limit in exploiting ILP was reached. Industry then faced a major change in design focus towards the exploitation of thread-level parallelism (TLP), with the first multicore processors being shipped to customers by Intel in the year 2005 (Intel, 2005).

3.2 Reuse techniques

As discussed in section 2, reusing pieces of computation that were previously performed can improve processor performance. Many different techniques have been proposed in the literature and come with different names. A survey study on the theme is presented by Sodani & Sohi (1998).

Dynamic instruction reuse is a technique proposed by Sodani & Sohi (1997) which is based upon the fact that many instructions that execute on a processor operate on the same input values. These instructions produce the same output values. Sodani proposed caching these outputs whenever an instruction completes in a cache named reuse buffer. When an instruction enters the pipeline, the reuse buffer is checked to see if that instruction is reusable (i.e., its input values are the same as in a previous execution). If it is, then the results are read from the cache and the instruction can be considered ready for retirement. This allows the instruction to skip the remaining pipeline stages, freeing pipeline resources, and also anticipating the execution of instructions which depend on it. It was shown in the study that the rate of repeating instructions can be quite high, reaching about 50% of the dynamic instructions for the benchmarks considered.

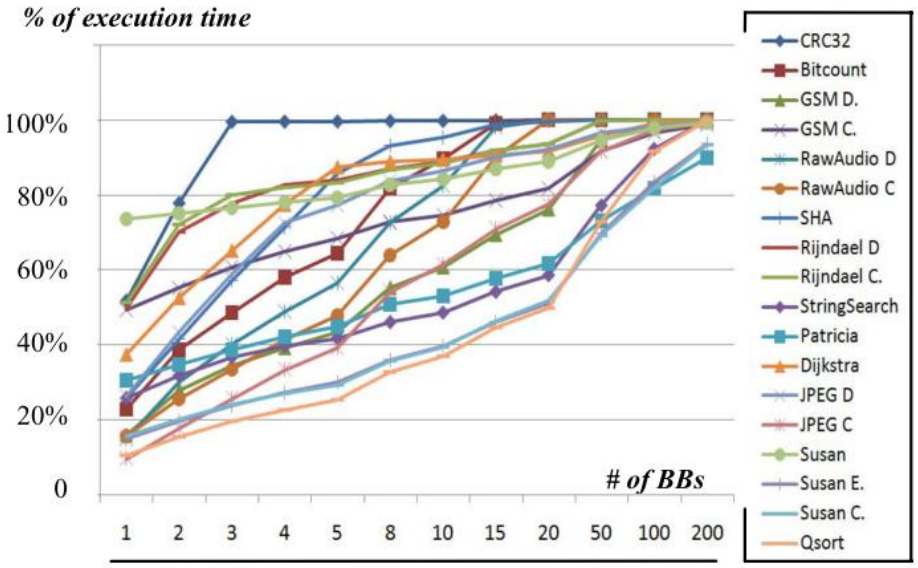
Another important technique is that of value prediction, proposed by Lipasti & Shen (1996). Value prediction is a speculative technique. It essentially consists of speculating on the input operands for an instruction before they have been computed, in order to anticipate the execution of other instructions. This, according to the authors, allows for breaking the data-flow constraint on computation (i.e., the serialization required to execute data-dependent instructions). The speculation on the inputs is naturally based on values previously seen when executing the same instruction; thus this mechanism is also classified as a reuse technique. Because it is speculative, the instructions need to be validated at commit time to confirm if the execution was correct. This subject is still a hot research topic, as can be seen from recent work by Perais & Seznec (2014).

The concept of dynamic instruction reuse presented before can be extended to consider entire sequences of instructions. Gonzalez et al. (1999) proposed a technique named trace-

level reuse. A trace is a sequence of instructions which already executed at least once. Multiple traces presenting the same input context will always produce the same output. A reuse trace memory stores information that allow for the reusability of traces, such as the initial memory address of the trace, the input registers and memory locations, the output registers and memory locations and the next program counter. It is argued by the authors that this method excels over individual instruction reuse because it reduces instruction fetch bandwidth and effectively increases the instruction window.

A study performed by Beck et al. (2008) also addresses the reusability of entire basic blocks. Their study analyzed for a set of applications how many BBs were responsible for a certain fraction of the execution time - or, in other words, how many BBs are required to cover a certain amount of executed instructions. The results, presented in Figure 3.3, show that for some applications less than 10 BBs already cover more than 80% of the executed instructions. The approach taken in their work to exploit this form of reuse is described in the next section.

Figure 3.3 - How many basic blocks are required to cover a certain fraction of the execution time.



Source: Beck et al. (2008).

3.3 Reconfigurable systems

The work by Compton & Hauck (2002) presents a survey on reconfigurable systems. These systems can be defined as a concept filling a gap between software execution (as in software executing on a microprocessor) and hardware execution (a dedicated ASIC

performing a specific task). It is usually required that the application designer specify which code sequences are to be executed on the reconfigurable unit, usually by inserting special instructions in the program binary. In the work at hands, the interest lies within systems that can automatically determine the computation kernels to execute on the reconfigurable unit at runtime, because this allows for maintaining binary compatibility. Only three reconfigurable systems in the literature address the mechanism of dynamic detection; those are described next.

Lysecky et al. (2006) define a concept named warp processor. A warp processor detects, during execution, an application's critical regions, reimplements those regions as a custom circuit in a simplified field-programmable gate array (FPGA), and replaces those regions in the program binary by a call to the hardware implementation of that region. This fine-grained approach allows control of bit-level operations. In their work, speedups of up to 6x and energy reductions of 66% with respect to an ARM7 processor were achieved. The downside of this approach is that it requires complex hardware to execute the computer-aided design (CAD) algorithms which map the code regions into the FPGA, thus implying great design costs and huge area overheads.

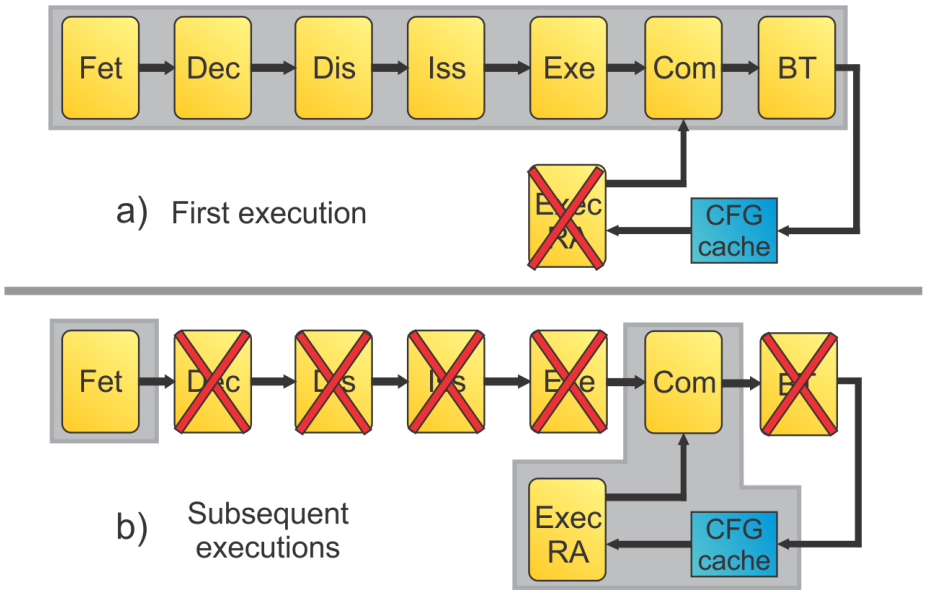
Important results are also presented by Clark et al. (2004). In their work, a coarse-grained reconfigurable unit is used, which operates on entire data units rather than bits. Their design is named CCA (Configurable Compute Accelerator), and relies on building an application's dataflow graph at runtime to perform the mapping of instruction blocks into the CCA. The CCA is organized as a triangular matrix of functional units; for each functional unit, its operation is determined based on the instruction being mapped. The results show that an average performance improvement of 25% both for embedded and general-purpose applications can be achieved.

Lastly, Beck et al. (2008) also employ a coarse-grained reconfigurable unit setup in a tightly-coupled fashion to a MIPS processor. In their setup, performance gains of up to 2.5x with respect to execution on a traditional MIPS processor were achieved, as well as energy savings of 55%. Besides, unlike the previous works, the reconfigurable unit employed also supported memory access operations, improving its range of application. Their work contributed to showing that both data-flow as well as control-flow oriented applications may present improvements when executed on the reconfigurable unit.

4 PROPOSED SYSTEM

As mentioned earlier in section 2.3, this work proposes a new microarchitecture for x86 processors, which exploits the fact that recurring basic blocks are very common in software (see Figure 3.3). 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 this new microarchitecture, a reconfigurable array (RA) is added to the superscalar core and used to improve execution on these recurring basic blocks (BBs).

Figure 4.1 - Behavioral overview of the proposed microarchitecture.



Source: the author.

In Figure 4.1 a behavioral overview of the system proposed in this work is presented. The blocks on the upper part of each figure represent the typical stages that compose the pipeline of superscalar processors, namely fetch, decode, dispatch, issue, execution and commit. A detailed explanation of the role of each stage will be presented afterwards. When a trace (a sequence of basic blocks) is executed for the first time (Figure 4.1.a), the instructions fetched from memory are decoded and executed as usual on the processor pipeline. When the execution is completed, the code sequences are fed into a binary translation (BT) mechanism, which performs the mapping of the micro-ops into a configuration for the RA. This configuration is stored in the configuration cache. When the same trace is executed again

(Figure 4.1.b), the configuration is read from the configuration cache and is executed in the RA. This way, all the logic required to access memory, decode instructions, execute register renaming and dependency checking can be skipped. Instructions continue to stream from the RA until the trace is completed or branch instruction with target address outside of that configuration is executed (i.e., the configuration represents an invalid trace). This mechanism extends the Loop Stream Detector, presented earlier in

Figure 2.2, thus improving the reuse mechanism of x86 processors. In the next section, we describe in details how the RA works, and afterwards the whole microarchitecture is presented.

4.1 Reconfigurable Array

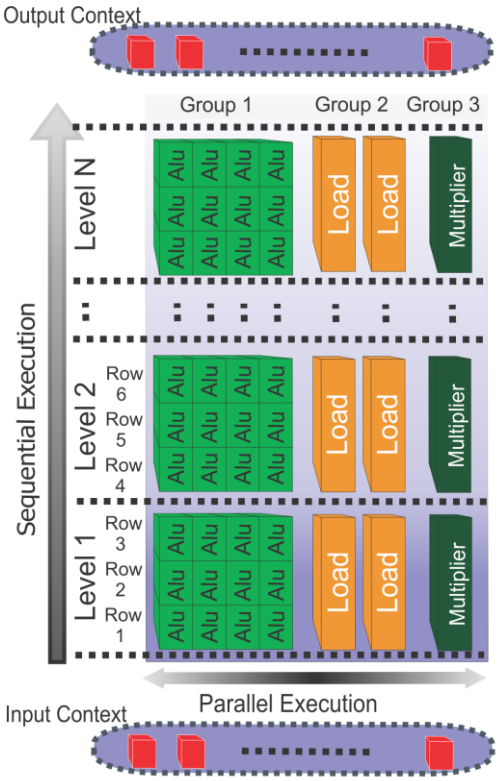
The design of the reconfigurable unit was taken from the work by Beck & Carro (2010) and is presented in Figure 4.2. It is a coarse-grained reconfigurable array, as described earlier. The array consists of a matrix of functional units (FUs), where each cell represents the execution of one instruction. Instructions that are independent may all be allocated on the same row; instructions that depend on others may only be allocated on rows above that of the instruction they depend on. In every processor cycle, one entire level may be executed. Unlike in superscalar execution, multiple dependent instructions may execute on the same clock cycle (because each level comprehends multiple rows).

Many design choices are available, which have a clear impact on the area of the array. For instance, the latency of the FUs impacts the amount of dependent instructions that may execute within the same clock cycle. Different types of FUs may be present, such that certain classes of instructions may only be allocated to certain units. In Figure 4.2, three different FU types are present: one for arithmetic-logic unit (ALU) instructions, another one for multiplications/divisions and another one for memory accesses. Because of the latencies, up to three sequential ALU instructions may be executed per cycle. Similarly, one memory and one multiplication/division operation can be performed per cycle. The amount of parallel FUs is also a design choice. In the figure, four ALU FUs are present per row; per level, two FUs for memory instructions and one FU for multiplication are present.

The input context of the array consists of buses connecting every register to the inputs of the FUs on the first level. To allow for data propagation inside the array, many interconnect schemes are possible. For instance, one could employ an interconnect such as the one presented in Figure 4.3. In that scheme, input multiplexers are present before each FU,

selecting from the input operands available which will be forwarded to each FU. Output multiplexers receive the values produced by the FUs and select which of them propagate to the next row or level. On the output context, multiplexers select the values produced on the last level of the array to be written back to the register bank (or, as is the case in this work, to the reorder buffer). Other more complex interconnect networks could also be employed to allow for any kind of data propagation within the array.

Figure 4.2 - Overview of the reconfigurable array.



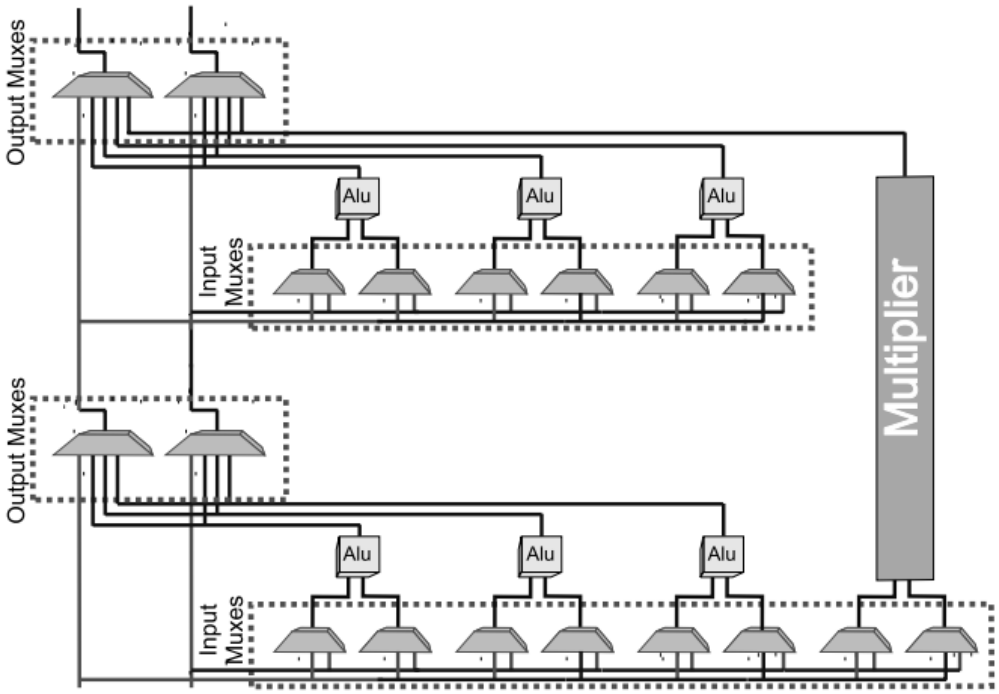
Source: Beck & Carro (2010).

A configuration for the RA is a sequence of control bits that are input to each FU and multiplexer. Each configuration represents the execution of one basic block[1], or possibly multiple basic blocks representing a trace. Each configuration also caches the sequence of micro-ops which it corresponds to. When a code sequence (which may be a single basic block, or a trace) has to be executed on the reconfigurable array, first the respective configuration has to be loaded. When the FUs and multiplexers are setup, the operands are

[1] In this work, a relaxed definition of basic block is employed. A basic block is any sequence of non-control instructions that is terminated by a control instruction. It still guarantees that, given the execution of the first instruction, all instructions up to the control instruction are executed.

fetched from the registers and loaded to the input context. Execution is started and takes a few cycles to complete, depending on the number of levels that the configuration spans. When done, the resulting computation is located in the output context, and values can be written back.

Figure 4.3 - Data propagation inside the reconfigurable array.



Source: Beck & Carro (2010).

The array has the potential to speed up applications, when compared with a traditional superscalar organization. First because multiple dependent ALU operations can be executed in the same processor cycle, unlike in the former organization. Besides, instructions need not be decoded and checked for dependencies before execution, potentially improving performance and reducing energy usage. Because of the flexibility provided, the RA can be employed to general purpose systems, unlike ASICs.

The RA also supports speculative execution. Speculation works by mapping to configurations entire traces, rather than a single basic block. This allows for better exploiting parallelism across basic blocks, which superscalar processors also do. After execution, a mechanism has to determine whether the trace executed is correct or not: the first BB in a trace is always correct, but the subsequent BBs are just a guess. A trivial solution would be to discard the entire execution, in case the trace executed deviated from the correct one. As will

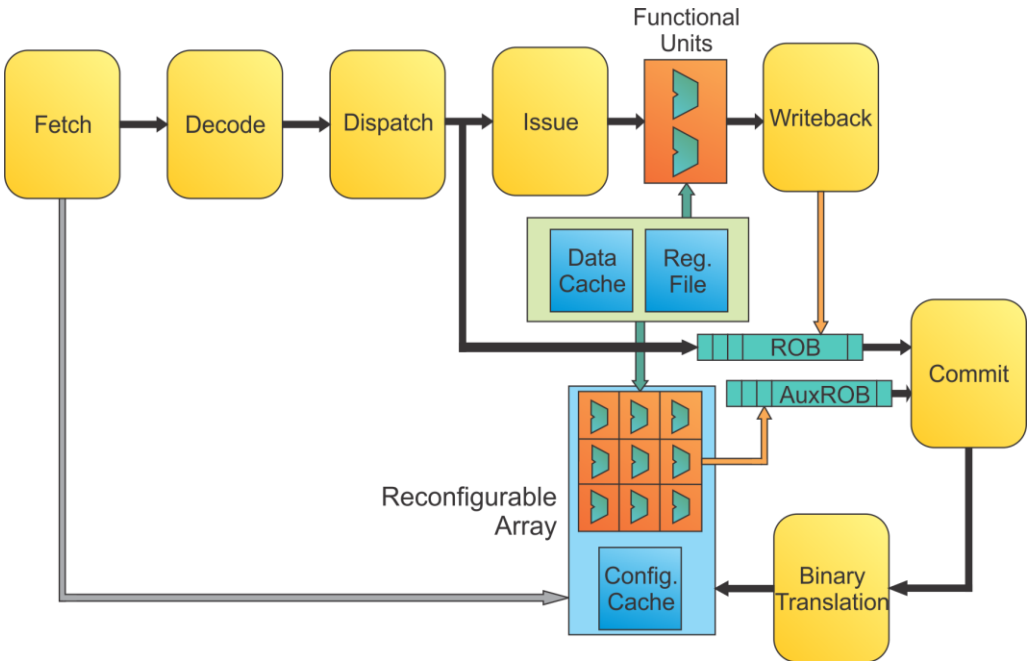
be shown next, the system proposed employs the reorder buffer (ROB) which is already present in the superscalar processor to handle speculation, providing a low-overhead technique to implement speculative execution in the RA.

4.2 Microarchitecture employing the RA in an x86 core

The microarchitecture of our system is composed of the typical superscalar, with the RA tightly coupled to the pipeline. This means the RA is located inside the processor core and has direct access to the register bank and the reorder buffer (ROB). The superscalar pipeline was modeled based on the architectural simulator employed in the performance evaluations, Multi2Sim (Ubal et al., 2012). The microarchitecture for this system is shown in Figure 4.4. The x86 pipeline is composed of 6 stages: instruction fetch, decode, dispatch, issue, write-back and commit. In the new microarchitecture, one additional stage is present which is that of binary translation. As shown in Figure 4.1, execution can proceed in one of two manners. The fetch stage is responsible for performing a lookup on the configuration cache to determine whether or not the instruction sequence addressed by the next instruction pointer (IP) is mapped to a configuration. When a code sequence is not mapped, then execution works as in a regular x86 pipeline. Otherwise, the configuration is loaded and execution takes place in the reconfigurable array. These two execution modes will be described in details.

For a code sequence which is not mapped to an array configuration, following actions take place. In the fetch stage, x86 instructions are read from the instruction cache and passed on to the decode stage. In the next stage, complex x86 instructions are decoded into micro-ops and put in a queue. In the dispatch stage, false dependencies between micro-ops are eliminated (via register renaming) and the micro-ops are fed into the ROB, as well as into one of two instruction queues: one for micro-ops performing memory accesses and one for all other operations. In the issue stage, a pre-defined number of micro-ops are read from the instruction queues and start execution in the functional units, considering the true data dependencies and the availability of the functional units. In the writeback stage, operation results are written to the ROB, in the position associated with each micro-op. Finally, in the commit stage, the micro-ops are removed from the ROB as soon as they are ready and confirmed to be non-speculative, and their results are written to the register file or data cache. After the micro-ops are complete, they are inserted in a queue, from which the binary translation mechanism reads and transforms in a configuration for the RA. This configuration is stored in the configuration cache.

Figure 4.4 - Microarchitecture of the proposed system.



Source: Author.

When executing a basic block which is already in the configuration cache (i.e.: it has already been translated), execution proceeds in a different manner. In the fetch stage, a signal is sent to the RA indicating it should load the next configuration (which is indexed by the current IP). In case the RA is currently executing a configuration, the fetch unit enters a waiting state until the RA is ready. The load process configures the FUs and the multiplexers, and also inserts the micro-op sequence to which that configuration corresponds in an auxiliary reorder buffer (AuxROB). The fetch stage also generates an RA synchronization instruction, which is forwarded to the x86 pipeline. This instruction will be used afterwards to coordinate the start of execution on the array and guarantee in-order commit. In the decode stage, the synchronization instruction is decoded into its respective micro-op. In the dispatch stage, the synchronization micro-op is inserted into ROB and the RA is notified to start execution. The RA may then have to wait until the configuration has finished loading and all data dependencies which compose the input context have been resolved. When ready, it starts executing the configuration. The execution results are written to the AuxROB, which behaves just like a regular ROB. Eventually, after the micro-ops that precede the execution on the RA have committed, the synchronization micro-op (which was inserted in the dispatch stage) will have reached the head of the ROB. This micro-op acts as a pointer to the AuxROB, from which the commit stage will start reading until all micro-ops executed in the RA have written

to their locations. Because that code sequence is already mapped to an array configuration, it needs not be applied to the binary translation stage.

Two key aspects of the microarchitecture, which are how the binary translation is performed, and how speculation is handled, are explained next in details.

4.2.1 Binary translation

The binary translation stage is responsible for transforming micro-op sequences into configurations for the RA. What it essentially performs is to build the dataflow graph of entire traces and then do a best effort to map the dataflow graph into the available functional units. The BT system employed in this work is similar to the one described in the work by Beck et al. (2008).

On each processor cycle, a certain number of micro-ops are read from the BT queue. For each micro-op, its inputs are compared against the outputs of all previously scheduled micro-ops. In case true data dependencies[2] exist, this micro-op may only be scheduled on a row or level higher than that of the micro-op producing the dependency. After determining the level/level, a check is also performed on the availability of functional units for that micro-op on that row/level; if none is available, then the operation is scheduled to the next first available row/level.

The micro-ops are continuously read from the BT queue until a branch micro-op is found. After the branch micro-op is scheduled, the configuration is stored in a temporary buffer. The algorithm may choose to continue scheduling the next BB, generating a speculative configuration, or terminate the process and move the configuration from the buffer to the cache. Different policies for this situation could be considered; for instance, the system could keep translating BBs until all functional units are filled, or terminate the configuration based on the ratio of micro-ops per level (which determine the ILP exploited), or attempt to map a fixed number of BBs per configuration (in this work, the latter policy was adopted).

The translation process may also terminate for other reasons. An unsupported operation may be input to the process, for example. Because no functional unit in the RA can execute that operation, the whole BB cannot be turned in a configuration. In this case, the

[2] Write-after-read and write-after-write hazards are automatiy eliminated, because no intermediate values are stored.

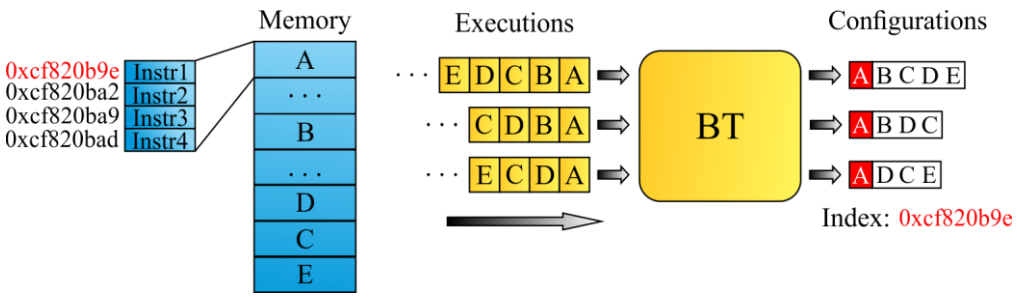
translation process is terminated, and the mechanism may choose to store only what has been previously saved on the temporary buffer or discard the entire configuration (in this work, the former policy is employed). The same happens if, during the translation process, no functional unit can be allocated for an operation (because the configuration is full).

4.2.2 Speculative execution

One of key aspect of the microarchitecture is the mechanism for handling speculative execution. The ability to start executing the next BB before the target of a branch instruction has been resolved increases the amount of independent instructions and allows the system to exploit parallelism between BBs.

As mentioned before, each configuration represents the execution of a trace, i.e. a sequence of basic blocks. Consider Figure 4.5, where A, B, C, D, E are basic blocks. Each of the sequences ABCDE, ABDC or ADCE are traces that were previously executed and generated a different configuration. In the microarchitecture proposed, each configuration is indexed by the IP of the first instruction in the first basic block, so these three would be indexed the same way as they start with basic block A. This way, only one configuration starting from A can exist at any time in the cache, which simplifies its design (because no additional mechanism for selecting a configuration is required).

Figure 4.5 - Configurations generated from distinct traces may be indexed by the same value.



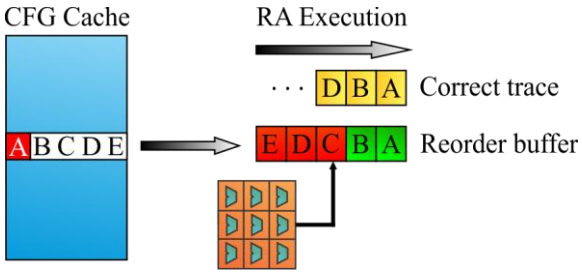
Source: the author.

When the fetch stage performs a lookup in the configuration cache, it searches for a configuration indexed by the current instruction pointer. Whenever a configuration is loaded to the RA, all of the basic blocks which compose it will execute. This means an incorrect trace could be executed. Consider, for instance, that ABCDE was the configuration loaded and the RA starts execution. The branch terminating basic block A executes and its target is

block B; the branch terminating B then executes but its target is block D, rather than block C. A mechanism is needed for solving the complications which arise: first, all operations already executed which come from blocks C, D or E need be squashed. Second, blocks A and B may have not yet been fully executed (because the branch may execute sooner than the other micro-ops in the block). In this scenario, the configuration must be executed until the end and execution should resume afterwards from block D, which is the target of the last executed branch.

Figure 4.6 clarifies the aforementioned example and also introduces the mechanism employed to handle speculation. In the proposed microarchitecture, it is handled in a similar way to that of the superscalar processor, with use of the reorder buffer. As such, coupled to the RA is an auxiliary reorder buffer (auxROB) which is accessible only by the RA. When a configuration is loaded, all micro-ops which are mapped to the configuration are also loaded (in their program order) to the auxiliary ROB. For each operation performed on the RA, its results are written to the corresponding auxiliary ROB entry. Each entry may only be committed after its entire BB has been confirmed to be non-speculative (after resolving the target of the previous branch instruction). When a branch target is not the next BB in the configuration, a situation of misspeculation ocurrs (just like in superscalar execution), and every instruction succeeding that branch is marked invalid in the auxiliary ROB. Nonetheless, as argued before, the configuration must be executed until the end. Afterwards, only the instructions marked as valid are committed from the auxiliary ROB.

Figure 4.6 - Use of the ROB to handle speculative execution.



Source: the author.

In order to avoid successive misspeculations in the same configuration, a second-chance algorithm is employed. Whenever a misspeculation occurs, a counter associated to the corresponding configuration is incremented. If there is no misspeculation the next time the configuration executes, then the counter is decreased; otherwise, it is increased. When the

counter reaches two, the configuration is removed from the cache. This ensures that configurations missing too many times will be removed and will not slow the processor down.

5 EVALUATION

Implementing a real processor is expensive (or impossible, in early-stage research). For this reason, simulation techniques have been employed to validate the proposed system. Simulation requires three components: a simulator, which models the real system; a set of benchmarks, which are representative of what the system will execute; and a set of metrics to evaluate the system. A discussion on performance evaluation is provided in the work by Jain (1991).

Multiple simulation platforms that model the x86 architecture are available. Three of them were analyzed: gem5 (Binkert et al., 2011), MARSSX86 (Patel et al., 2011) and Multi2Sim (Ubal et al., 2012). The first two are full-system simulators, while the latter is application-only, and the first two present a richer set of features than the latter. Nevertheless, Multi2Sim was chosen for its simplicity (thus presenting an easier learning curve). Besides, it was thought to provide a good microarchitectural model of the x86 architecture, had good documentation and was modularly written in C (making it easier to alter afterwards).

As for the benchmarks, the MiBench (Guthaus et al., 2001a) benchmark suite was used. It comprises a wide range of applications, from control- to data-flow oriented, and is targeted towards the embedded system domain. Because the x86 ISA implemented by Multi2Sim does not support recent instruction set extensions or 64-bit binaries, programs had to be compiled with gcc version 4.4 (older than the current release) and using the -m32 flag (to generate 32-bit binaries). Additionally, the O3 optimization flag was used. Of the 24 benchmark applications in the suite, only 13 could be compiled without modifications; these were selected for the simulations.

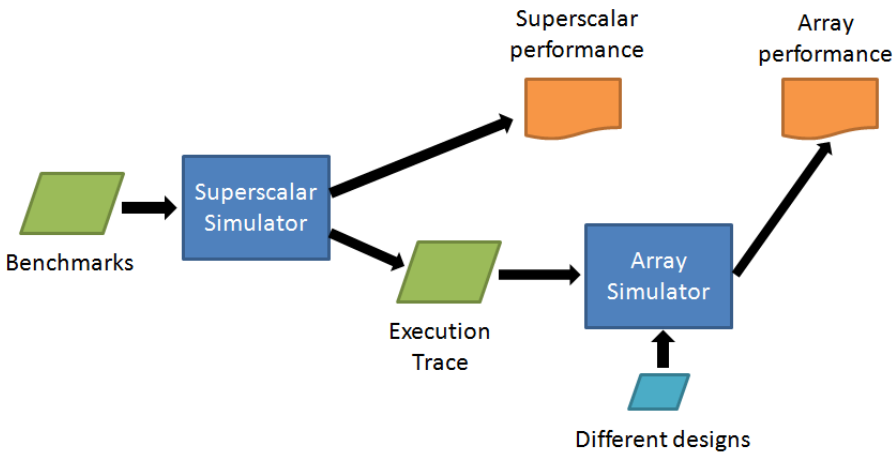
With respect to metrics, the most accurate measurement for comparing different computer systems is execution time. However, it is possible to compare both using instructions per cycle (IPC), or even micro-instructions (micro-ops) per cycle (uIPC), because both the original superscalar machine and the system proposed run at the same clock speed and implement the same underlying ISA. In this study, the latter metric (uIPC) was used for comparing the systems, because the RA executes micro-ops, rather than instructions. Other metrics were also collected, with the intention to identify possible performance bottlenecks. These other metrics will be presented along with the results. Two analyses were performed, one for potential and one for detailed behavior.

In the first analysis, a comparison of a program executing on the x86 simulator against the same program executing only on the RA was performed. The objective of this potential

analysis was to perform a quick, despite low-precision, evaluation on whether the system proposed could perform better than the superscalar. To do so, a trace-driven simulator for the RA was developed, which was written in C (for fast simulation times) and comprised over 1500 lines of code. The simulator implements an algorithm that reads BBs from a trace file, transforms them into configurations for the RA and counts how many cycles the execution would take. The simulator also includes a visualization tool which can convert RA configurations into DOT files (a graph description language), which can be graphically plotted using tools such as Graphviz (2014).

To generate the trace file, each of the applications in the benchmark set was executed on Multi2Sim. During this execution, the number of uIPC for the superscalar processor was obtained. When performing the trace-driven simulation on the RA, different designs were considered. From this execution, performance results were extracted and then compared with execution on the superscalar processor. This methodology is illustrated in Figure 5.1.

Figure 5.1 - Methodology for the potential analysis on the system performance.



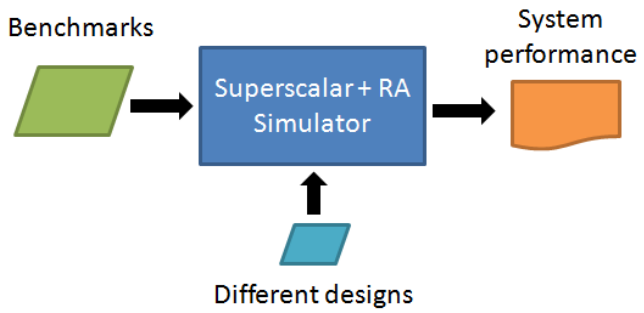
Source: the author.

In the detailed analysis, the Multi2Sim simulator was extended to incorporate the RA and model the entire system as depicted in Figure 4.4. First, the RA was added to the system. Next, some of the existing pipeline stages in the Multi2Sim simulator had to be modified to coordinate execution within the array, for example the communication mechanism between the fetch engine and the RA, and between the dispatch stage and the RA. About 2000 code lines were added to or modified in the simulator (Appendix A shows a simplified view of the source code tree of the simulator, with the added and modified modules). Because the system is complex and would require significant amount of time to properly implement and validate,

a few assumptions on the implementation were performed (they will be described afterwards, in section 5.3).

In this latter analysis, the benchmarks were executed directly on the simulator, generating performance reports. Just like before, different design choices for the RA block were considered. The methodology is shown in Figure 5.2.

Figure 5.2 - Methodology for the detailed analysis on the system performance.



Source: the author.

The next section describes the Multi2Sim simulator, which models the behavior of the superscalar processor.

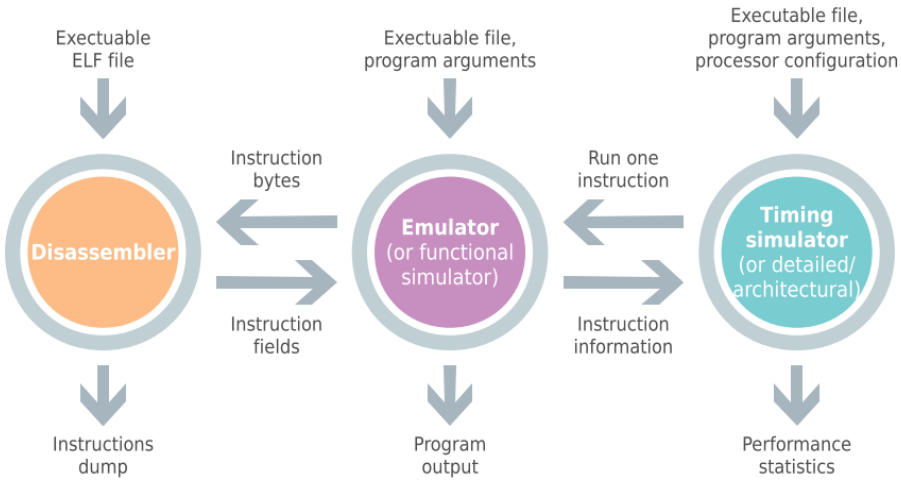
5.1 The Multi2Sim simulator

Simulation works by modeling the behavior of a real system. Because these systems are usually complex, simulation models key aspects of the system and makes a few assumptions about the details. This section describes the Multi2Sim simulator and its model of the x86 superscalar architecture, as presented in the user guide (Barton et al., 2014). A description of this model is important to understand the boundaries of this work.

To start with, the Multi2Sim simulation paradigm is described in Figure 5.3. Three distinct modules comprise the simulator: the disassembler, the emulator (or functional simulator) and the timing simulator (or detailed simulator). In essence, the disassembler is responsible for reading bit streams and interpreting them as machine instructions; the emulator models instruction behavior from an input/output point of view and the simulator models the flow of instructions inside the machine as they execute. The entire simulation framework was modularly designed, such that each module in Figure 5.3 requires all modules to the left in order to work (the emulator requires the disassembler, and the simulator requires the emulator). When programs execute on the timing simulator, it requests the functional

simulator to execute an instruction. The functional simulator reads the program binary (if necessary) and passes the instruction bytes to the disassembler, which returns the instruction fields. The functional simulator executes the instructions and passes execution information to the timing simulator.

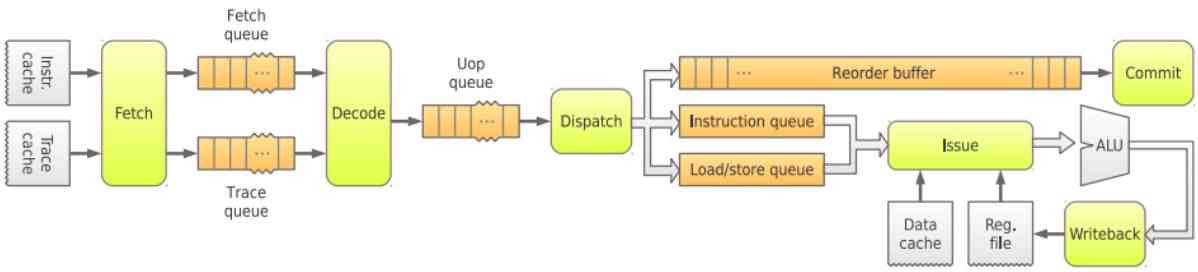
Figure 5.3 - Multi2Sim's simulation paradigm.



Source: Barton et al. (2014).

The internal structure that the timing simulator models is the superscalar pipeline for the x86 architecture, which is depicted in Figure 5.4. This model is just like the upper part of Figure 4.4, except it is shown here with slight more detail. Each pipeline stage plays the role that was already described in section 4.2; however, the simulator simplifies many aspects of the pipeline implementation which have a clear impact on performance.

Figure 5.4 - The superscalar pipeline, as modeled in the Multi2Sim simulator.



Source: Barton et al. (2014).

The fetch stage holds the current instruction pointer (IP), which points to the next instruction to be executed. When fetching, a call is performed to the functional simulator, which is requested to execute the instruction pointed by the IP. After execution, the functional simulator returns to the fetch mechanism a list of the micro-ops which were executed, with all information regarding its execution (the next instruction, the size of the instruction which generated the macro-op, the dependencies, and others). The timing simulator then pretends that no information is yet known on these instructions, and forwards them to the pipeline. The decode stage pretends to decode the instructions and forwards them to the next pipeline stage.

The dispatch and issue stages are responsible for checking dependencies among the micro-ops. Specifically in the dispatch stage, register renaming (the mapping of logical to physical registers) is performed, which eliminates all write-after-read and write-after-write dependencies. When done, micro-ops are inserted in program order either into the instruction queue or into the load/store queue (depending on the micro-op), and into the reorder buffer as well. The issue stage selects micro-ops that have their input operands ready and do not present read-after-write dependencies, and sends them to the functional units. Instructions are selected for out-of-order execution from the load/store queue, ignoring possible memory aliasing problems.

In the execution phase, the functional units are modeled as a pool of specialized resources. Each functional unit is specialized for a class of micro-ops, such as integer, logic, branches, floating point and vector, and their number and latency may be configured. When an instruction is issued for execution, a functional unit is allocated for it and the instruction is put in an event queue. After a certain number of cycles, determined by the operation latency, the instructions are removed from the event queue and the corresponding functional unit is freed. The results are written to the register file (in contrast to a typical superscalar, where the results are written to the reorder buffer). This is the write-back stage. There is no control on the number of operations simultaneously writing to the registers, which is the same as assuming there is an unlimited number of ports on the register bank.

In the commit stage, a finite number of operations is read from the reorder buffer and marked as completed. It is also at this stage that branches that caused misspeculation are resolved; when a misspeculation is detected, the entire pipeline is cleaned and execution proceeds from the target branch address.

Multi2Sim allows the user to specify parameters for the memory system, such as the amount of cache memory available, the amount of cache levels, the latencies, and number of memory ports. However, it was noticed during the simulations that the number of memory

ports specified presents no impact on performance. Likewise, it was noticed that the memory operations may take longer than the latency specified.

Summarizing the above discussion, the structures which are not correctly modeled in Multi2Sim are as follows:

- The functional and timing simulations are decoupled. Execution takes place when fetching instructions, and all micro-ops information is already available at the fetch stage;
- Memory aliasing problems are ignored, i.e., loads or stores may be freely reordered even though they access the same memory location.
- There is no control over the number of simultaneous accesses to the register bank.
- There is no control over the number of simultaneous accesses to the memory (or the simulator restricts this to a fixed number, regardless of what the user specifies).
- Memory accesses may take longer than the latency specified.

5.2 Potential analysis

For the potential analysis, a simulator for the RA alone was implemented. The simulator implements a simple algorithm to determine the potential that the RA has for ILP extraction. It reads instructions from the trace file until a branch instruction is found. Each instruction is scheduled to an appropriate functional unit within the RA, based on the true data dependencies. When the scheduling is done for a basic block, two global counters are incremented: one for the amount of micro-ops that were mapped and the other for the amount of cycles that the execution took (which is given by the number of levels with at least one instruction). After all basic blocks have been mapped, the potential for ILP extraction is the average number of uIPC, given by the ratio of the two counters.

As mentioned in section 4.1, different design choices are available concerning the size of the RA, the functional units available, the latencies and so on. At first, interest was on evaluating the impact of each design choice on the performance the applications. It was assumed that an infinite number of FUs is available, so that only the data dependencies impact the performance. The amount of different functional units was fixed to three, and all operations were supported. The three types are as follows: one for ALU operations (additions, subtractions, register moves, sign change, logic operations - shifts are not included here); one for memory operations (loads and stores); and one for all remaining operations. The latency

of the ALU FUs was varied considering 1, 1/2 and 1/3 of a cycle, because previous work have already shown that multiple simple operations can be executed in sequence within one clock cycle (Clark et al. (2004) and Beck et al. (2008)). All other FUs operations had their latency fixed to one cycle latency. One additional parameter was analyzed, which is the number of memory operations allowed to be scheduled for each level in the RA. Because a real memory has a fixed number of access ports, it is reasonable to limit the amount of simultaneous accesses. This parameter was varied from an infinite number to 4, 2 and 1 operation per level. These descriptions correspond to the six different design considered, which are shown in Table 5.1.

Table 5.1 - Different array designs considered on the potential analysis.

Parameter	Design 1	Design 2	Design 3	Design 4	Design 5	Design 6
ALU operation latency	1 cycle	1/2 cycle	1/3 cycle	1/3 cycle	1/3 cycle	1/3 cycle
Max. memory ops. per cycle	Unlim.	Unlim.	Unlim.	4	2	1

Source: the author.

The results for the execution of each benchmark on the RA alone is presented in Table 5.2. It shows the average uIPC observed for each benchmark under the 6 different designs, and orders the benchmarks by the average BB size (it is expected that large BBs contain more parallelism than small ones, because the instruction window analyzed is larger). These results do not include yet a comparison with execution on a superscalar machine; they do, however, provide interesting insight into the nature of the applications in the benchmark. For instance, moving from designs 1 to 3, the latency of each ALU functional units (and, therefore, the amount of dependent ALU operations that may be executed in a single cycle) was varied from 1 to 1/3 of a cycle, which caused the average uIPC to raise by 11% only within a single BB. Some benchmarks increased well above average when moving from design 1 to 3, such as susan-s (20% increase), which suggests these may benefit more from implementing their algorithms in combinatorial logic, while others increased below average, such as adpcm (2% increase), which suggest little benefit from this approach. When moving from design 3 to designs 4, 5 and 6 it was expected that the benchmarks which presented a high rate of memory operations would present a strong decrease in the number of uIPC. On average, design 4 performed 6% worse than design 3; design 5 performed 15% worse and design 6 about 30% worse than design 3. Benchmarks of the susan family present the most significant

decrease, with susan-e decreasing 27% when moving from design 3 to 4 and almost 70% decrease when moving from design 3 to 6.

Table 5.2 - Average BB size for each of the benchmarks and uIPC values for execution on the RA under six different designs.

Benchmarks	Avg. BB size	uIPC					
		Design 1	Design 2	Design 3	Design 4	Design 5	Design 6
dijkstra (large)	5.49	1.32	1.42	1.43	1.42	1.42	1.39
adpcm (encode, small)	5.80	1.67	1.71	1.71	1.71	1.69	1.56
adpcm (decode, small)	6.02	1.69	1.76	1.76	1.76	1.71	1.54
stringsearch (large)	6.78	1.90	2.24	2.29	2.28	2.18	1.95
bitcount (small)	6.85	1.94	2.26	2.27	2.27	2.25	2.09
gsm (decode, small)	8.11	1.63	1.72	1.73	1.73	1.67	1.53
qsort (small)	8.25	2.21	2.36	2.39	2.34	2.19	1.82
CRC32 (small)	9.42	1.85	2.02	2.02	2.02	1.95	1.64
patricia (large)	9.57	2.19	2.39	2.45	2.42	2.24	1.84
blowfish (encode, small)	9.86	1.86	2.02	2.07	2.06	2.00	1.74
blowfish (decode, small)	9.93	1.88	2.04	2.09	2.08	2.01	1.75
basicmath (small)	9.94	2.24	2.47	2.51	2.49	2.33	1.95
FFT (inverse, large)	10.89	2.38	2.63	2.67	2.65	2.48	2.06
FFT (normal, large)	11.30	2.47	2.71	2.76	2.73	2.50	2.02
jpeg (encode, small)	12.17	2.35	2.46	2.46	2.32	2.14	1.72
gsm (encode, small)	17.50	2.69	3.02	3.12	2.97	2.66	2.16
jpeg (decode, small)	22.65	4.25	4.34	4.35	4.08	3.38	2.19
susan (smoothing, small)	27.17	2.69	2.94	2.95	2.93	2.90	2.38
susan (corners, small)	28.15	3.84	4.35	4.48	3.89	3.14	2.14
susan (edges, small)	37.47	5.96	6.92	7.21	5.31	3.74	2.28
Average	13.17	2.45	2.69	2.73	2.57	2.33	1,89

Source: the author.

It is important to note that in this first experiment the parallelism between basic blocks was not exploited, because each RA configuration was composed of only one BB at a time. It is expected (although this exact experiment was not performed) that by allowing multiple basic blocks to execute simultaneously within the RA the increase in uIPC seen when moving from design 1 to 3 would be greater than 11%. Similarly, restricting the amount of memory operations per cycle caused a steep fall in the average uIPC, because additional levels were required within the array to accommodate all memory operations. In a speculative execution setup, however, the additional cycles generated could be used to execute instructions from the next BB which do not depend on that memory access, alleviating the uIPC drop.

The next experiment aimed to compare execution on the superscalar machine against execution on the RA alone; for this, speculative execution was also considered on the RA. Design 4 was taken and extended to support speculative execution in 5 distinct setups where

the number of BBs speculated was ranged from 1 to 5. As for the superscalar simulator, Multi2Sim presents many configurable parameters with respect to the branch prediction mechanism, size of the structures in the pipeline, functional units count and latency and also for the cache and memory system. The parameters used in this analysis are shown in Table 5.3. The trace file used by the array simulator contains no information on the latency of the memory accesses executed, and does not include speculative code sequences (i.e., instructions that were executed but later turned to be on the wrong execution path). In order to put both the RA and the superscalar on the same baseline for comparison, the branch prediction in the superscalar was setup to always predict the right path. For the same reason, memory accesses are considered as taking exactly one clock cycle in the array and in the superscalar simulators.

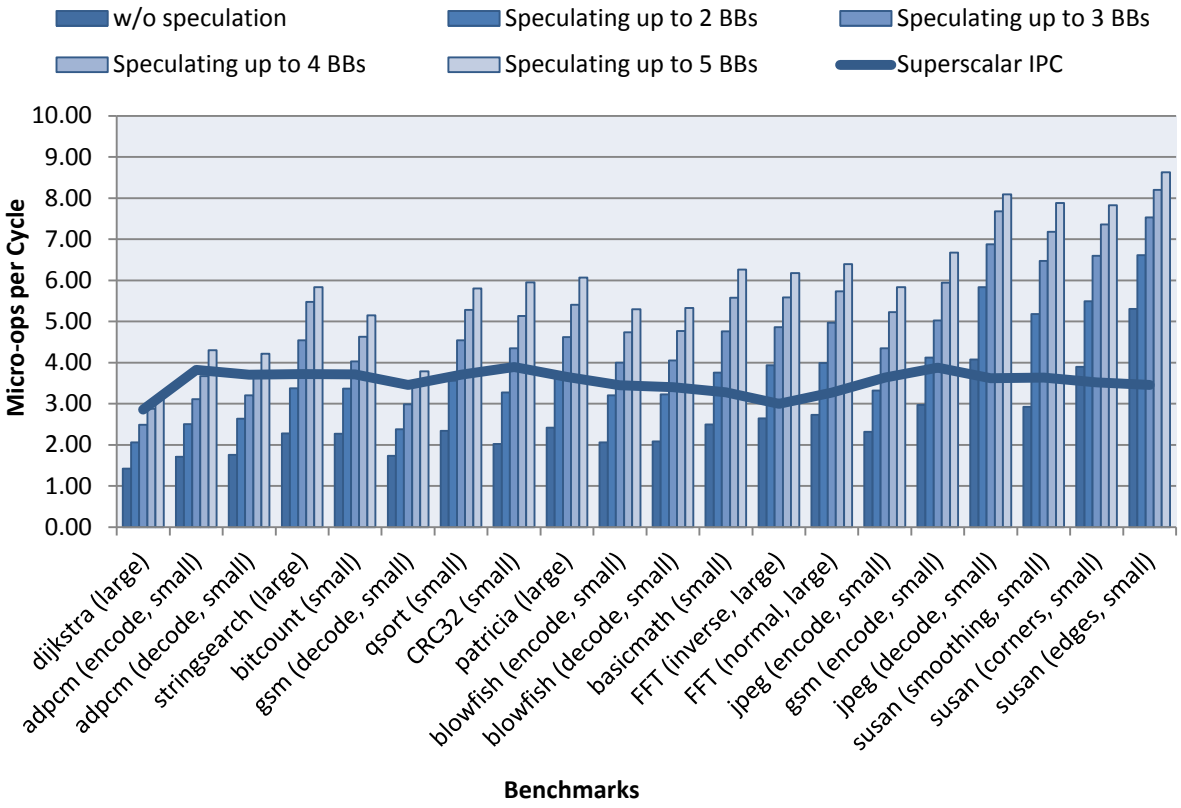
The results are presented in Figure 5.5. In the chart, the number of uIPC observed for each benchmark is displayed. Each different bar represents execution on the RA, using a different degree of speculation; for instance, speculating up to three basic blocks means each configuration of the array spanned 3 adjacent basic blocks. The black line represents the number of uIPC observed on the superscalar machine executing with the design parameters given in Table 5.3. In the graph, the benchmarks are ordered increasingly by their BB sizes, just like in Table 5.2.

Table 5.3 - Configuration of the superscalar processor employed in the potential analysis.

Branch predictor	
Kind	Perfect
Pipeline	
Dispatch width	4 micro-ops per cycle
Issue width (from each queue)	
Commit width	
Structures	
Micro-op queue	40 micro-ops
Re-order buffer	128 micro-ops
Instruction queue	36 micro-ops
Load/Store queue	32 micro-ops
Register file	64 registers
Functional Units	
Integer	3 units
Logic	3 units
Memory	
Access latency	1 cycle
Number of ports	4

Source: the author.

Figure 5.5 - uIPC for different benchmarks executing on the RA, considering speculative execution.



Source: the author.

As can be seen in the chart, as the amount of speculative execution increases so does the uIPC. This is expected, because increasing the speculation degree also increases the amount of instructions, which increases the amount of parallelism available. What is interesting to note is that, for most of the benchmarks, executing only three basic blocks simultaneously is already enough to perform better than the superscalar processor. On average, speculating up to two basic blocks is already enough to provide 6% higher uIPC (average) than that of the superscalar; for 3, 4 and 5 BBs speculated the numbers are of 32%, 52% and 67%, respectively. Another interesting point is a tendency for the uIPC to grow when moving from left to right on the charts, crossing multiple benchmarks, but only for the RA (the superscalar uIPC presents a flat line). This happens because the as the BB size increases, so does the instruction window size (as seen from the BT algorithm), and thus the amount of parallelism available to be exploited by the RA. The superscalar processor employs a fixed size instruction window, having less performance influence from the basic block size.

The same can be said about the performance improvements seen when increasing the amount of speculation.

It is fair to compare execution on the superscalar with a 5-BB speculation scheme on the RA, considering the average BB size for each benchmark (Table 5.2) and the size of the structures on the superscalar processor (Table 5.3). Given the size of the instruction queue (36), from which independent instructions for execution are selected, and the ROB (128), which represents the amount of operations that can be in-flight simultaneously, up to five BBs at a time may be executing on the superscalar machine. For the applications which have larger BBs, just three BBs may already contain enough micro-ops to fill the instruction window or ROB, but these applications already present performance gains when executing only one or two BBs simultaneously in the RA.

It is clear, from the potential analysis, that the array has higher potential to exploit ILP than the superscalar. The analysis, assuming perfect branch prediction and one-cycle memory access latency for both the superscalar and the RA, opens the question on how the array would perform under a real scenario. The assumption that every basic block executes on the RA is not true, because each basic block must execute at least once on the superscalar pipeline before being translated into a configuration. Given a large enough configuration cache size, however, and considering that some traces are recurring in program execution, one can predict that most of the basic blocks actually execute on the RA. The detailed analysis of the RA coupled to the processor pipeline aims to provide better insight into the system performance.

5.3 Detailed analysis

In this analysis, Multi2Sim was extended to include the RA. A few implementation assumptions were made: first, the input and output context mechanism was not implemented, neither the time delay required to load a configuration. The assumption that configurations can be loaded in one cycle is not so restrictive, though; as mentioned before, a configuration is detected at the fetch stage, but starts execution only on the dispatch stage. This gives the system a few cycles to perform the load process. The second assumption is that a configuration may start executing as soon as it is loaded, ignoring any data dependency in the input context which may not yet been resolved (although proper in-order commit is still guaranteed by use of the ROB). Just like before, it is assumed that memory accesses take exactly one cycle and that the branch predictor always hits (although a configuration, once loaded, may execute the first mapped BB correctly and then suffer a misspeculation). In the

reconfigurable array, an infinite number of functional units per level is assumed, and also that there is an infinite number of slots in the configuration cache (so configurations are only removed after they have misspeculated two times, as explained in section 4.2.2). Additionally, the commit width was left unconstrained, to allow for a correct measurement of the uIPC. The issue width for operations in the load/store queue was also modified to one micro-op per cycle. As for the RA, the design taken for this analysis was the design 4 (as shown in Table 5.1). All of these assumptions and design configurations are shown in Table 5.4.

Table 5.4 - System configuration for the detailed performance analysis.

Configuration load latency	1 cycle
Register load/store latency	
Execution start (RA)	When synchronization micro-op reaches dispatch stage; does not wait for input context resolution
Branch predictor	Superscalar branch pred. always predicts next BB correctly; misspeculation still occurs if configuration loaded is incorrect trace.
Memory accesses	1 cycle latency
Issue width (instr. queue)	4 micro-ops per cycle
Issue width (load/store queue)	1 micro-op per cycle
Commit width	Infinite micro-ops per cycle
Configuration cache	Infinite slots
Other supescalar param.	According to Table 5.3
Other RA param	According to design 4 in Table 5.1

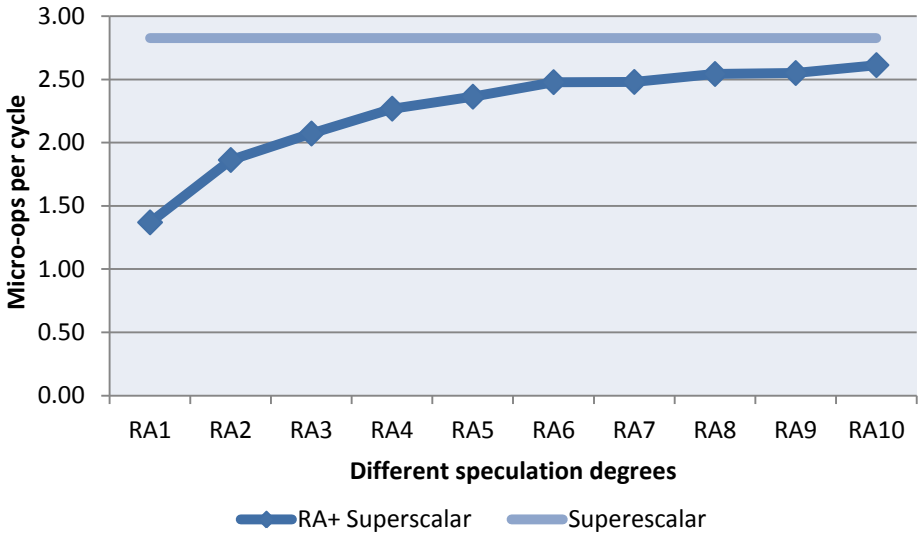
Source: the author.

Just like before, multiple simulations were run considering different values for the degree of speculation performed. The number of basic blocks allowed per configuration was varied from 1 up to 10. The interest still lies within the number of uIPC; however, this experiment also aims to analyze potential performance bottlenecks in the system. In this sense, other metrics were collected as well, such as the number of micro-ops and configurations executed on the RA and also relative amount of micro-ops which were executed on the RA.

The average uIPC for all the applications executing on Multi2Sim, considering the different degrees of speculation, is shown in Figure 5.6. The dark curve represents the uIPC of the system composed of the superscalar and the RA, and the light curve is the uIPC of the superscalar alone. Each numbered label on the horizontal axis indicates the number of BBs

speculated, i.e. the number of BB allowed per configuration. The curve presents a steep rise when initially increasing this degree, but approaches a flat limit from RA6 onwards. One possible cause is that, eventually, the speculation scheme becomes so aggressive that all BBs additionally executed are discarded and do not commit.

Figure 5.6 - Average uIPC of the system proposed and the superscalar.



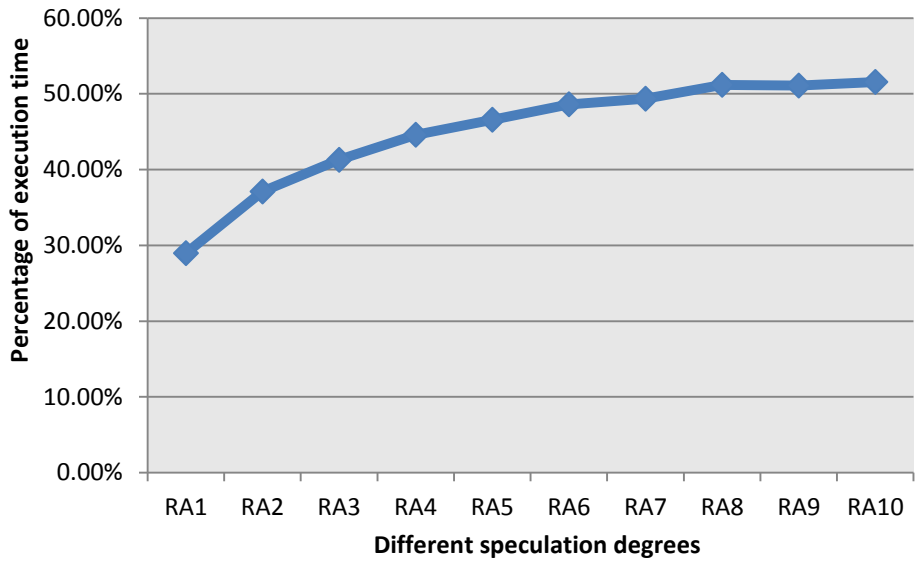
Source: the author.

It is important to note that the proposed system does not perform better than the superscalar, in this analysis. This lead to a deeper investigation on the Multi2Sim simulator, during which it was learned that the memory system was not behaving as it was thought to be. What happens is that the premise that memory accesses in Multi2Sim take exactly one cycle, as shown in Table 5.3, did not hold. In fact, every time a memory operation executed on the RA, it had to wait until that memory operation completes before executing the next level. If accesses took exactly one cycle, the RA would never have to wait, but this seemed not to be the case.

An experiment was performed to identify whether this assumption held and, whether it did not, how big was the impact on the RA performance. A counter was incremented every cycle in which the array could presumably execute a level, but had to wait for the memory operations in the previous level to complete. Afterwards this number of cycles was compared against the total cycles which execution took to determine the percentage of execution time which the array spent waiting. These results are shown in Figure 5.7. As can be seen in the figure, the array spends on average at least 30% of total execution time executing nothing,

when speculative execution is turned off. When it is turned on, this number quickly rises, reaching about 50% when using the most aggressive speculation degree.

Figure 5.7 - Percentage of total execution time in which the RA had to wait for memory operations to complete.



Source: the author.

Up until now, no explanation has been found as to why the simulator behaves this way. Analysis shows, however, that this behavior of memory accesses taking longer than one cycle is native to the Multi2Sim simulator, not to its extended version developed in this work. One of the reasons may be that the simulator limits the number of ports in memory without telling the user.

The results presented so far may be suggestive of the array behavior when considering memory access latency, but this was not the initial intention when specifying the experiment. With the results obtained, however, it is possible to estimate the system performance in case memory operations in the RA presented one cycle access latency. First, it is required to estimate the uIPC of the RA alone, when included in the system. Next, the uIPC of the RA considering no memory latencies is estimated. Finally, the uIPC of the entire system, given this condition, can be estimated. The methodology can be better explained with the following formulas.

$$IPC_{sys} = \%_{exec,RA} * IPC_{RA} + (1 - \%_{exec,RA}) * IPC_{ss} \quad (1)$$

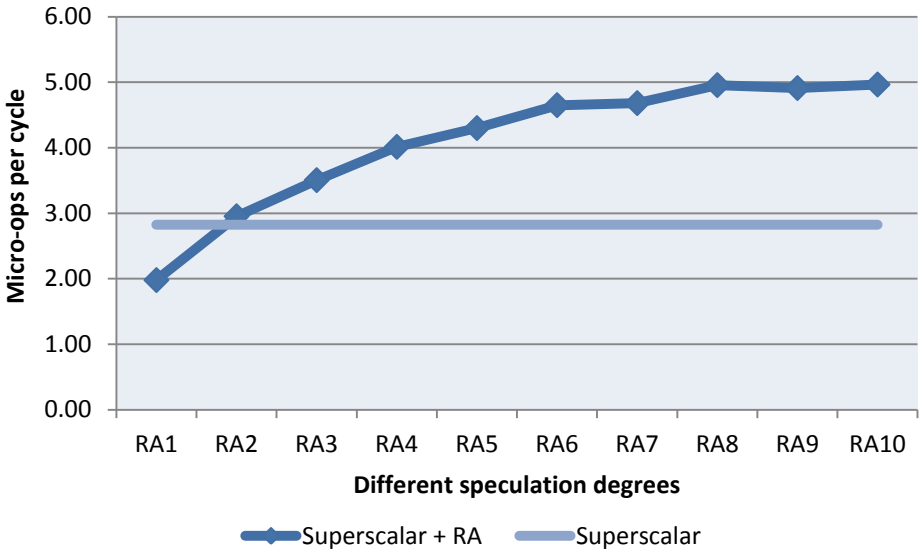
$$IPC_{RA} = (1 - \%_{time,RA\_no\_exec}) * IPC_{RA,no\_mem} \quad (2)$$

$$IPC_{sys\_no\_mem} = \%_{exec,RA} * IPC_{RA,no\_mem} + (1 - \%_{exec,RA}) * IPC_{ss} \quad (3)$$

Equation 1 is used to estimate the IPC of the entire system (IPC_{sys}), given the percentage of committed instructions which executed on the RA ($\%_{exec,RA}$) and on the superscalar ($1 - \%_{exec,RA}$), and the IPC of the RA (IPC_{RA}) and of the superscalar (IPC_{ss}). This equation provides the value of IPC_{RA}, which is the only unknown.

Equation 2 uses the same idea as Equation 1, but applied to the RA alone. The IPC observed in the RA (IPC_{RA}), which was obtained in Equation 1, is the fraction of time in which the array executed instructions and was not awaiting for memory operations ($1 - \%_{time,RA\_no\_exec}$) times the IPC of the RA in case of one-cycle access latency (IPC_{RA,no_mem}). The values for $\%_{time,RA\_no\_exec}$ are the ones depicted in Figure 5.7. Notice this equation is symmetric with Equation 1, but a term $0 * \%_{time,RA\_no\_exec}$ is not shown (when the RA has to wait for a memory access to complete, it executes no operation). This equation provides the value of IPC_{RA,no_mem}, which is the only unknown.

Figure 5.8 - Average uIPC of the system proposed and the superscalar, given one cycle memory access latency on the RA.



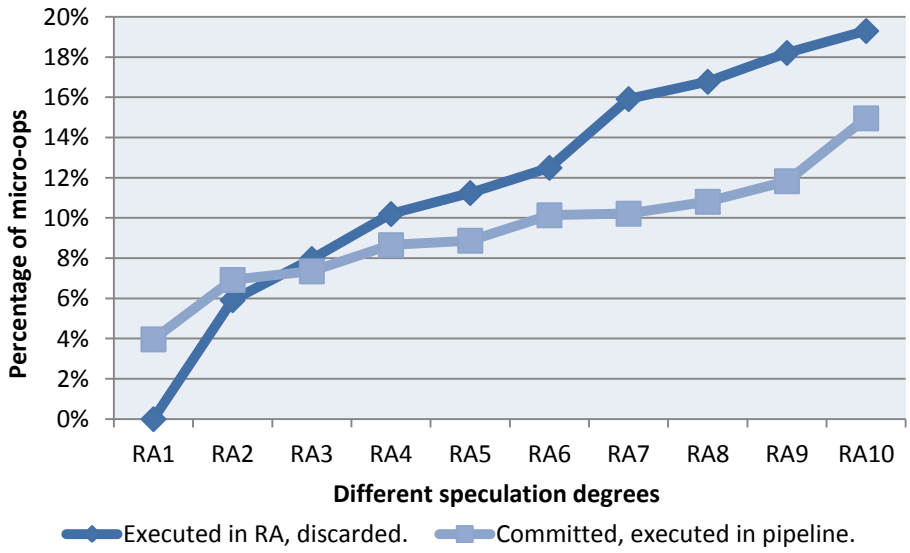
Source: the author.

Given IPC_{RA,no_mem}, we use the same kind of equation to estimate system performance in case of one-cycle access latency. This is given by Equation 3, where $IPC_{sys\_no\_mem}$ is the desired value. This provides an estimation for the IPC of the entire system in case memory accesses in the RA take exactly one cycle. These values are depicted

in Figure 5.8. Now, as can be seen, the performance of the proposed system excels that of the superscalar, when using a degree of speculation of two or higher (RA2-). When speculating with 10 BBs (RA10), the performance gains can be of up to 75%. This performance improvement seems to saturate when speculating with more than 8 BBs; this happens due to the high level of instructions which suffer misspeculation.

The dark curve in Figure 5.9 shows the percentage of micro-ops which were discarded, among all micro-ops which executed on the RA. As can be seen, this ratio can be quite high, reaching about 20% for RA10. Although it is not clear how these discarded micro-ops affect performance (because there is no drop in the curve in Figure 5.8, only saturation), it clearly affects energy consumption negatively, because these micro-ops which were already executed are discarded afterwards. The light curve in the same figure shows the percentage of micro-ops which were executed in the superscalar pipeline, among all committed micro-ops (notice the two percentages are over different sample spaces). The number rises as the amount of speculation is increased, because configurations spanning multiple BBs have a higher chance of misspeculating; still, even for the most aggressive scheme more than 80% of the committed instructions execute on the RA.

Figure 5.9 - Amount of discarded micro-ops in the RA execution, and amount of micro-ops which were executed in the processor pipeline.

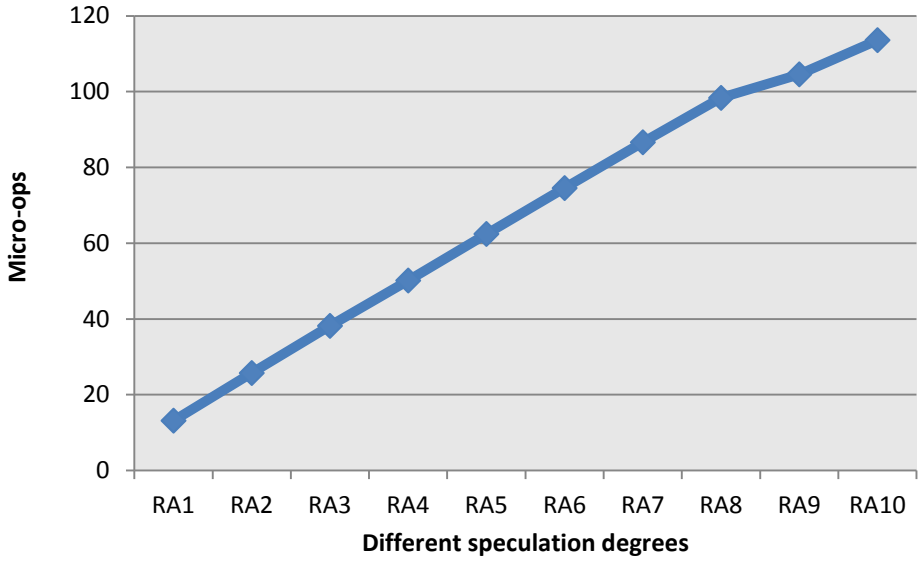


Source: the author.

One could also be interested in the average size of the configurations, in order to evaluate potential area overheads which the use of the RA would imply. This is shown in

Figure 5.10. It is clear that the size of the configurations grows linearly as the amount of speculation performed is increased. A larger configuration requires more functional units in the RA, as well as more space in the configuration cache. Given an area constraint on the configuration cache, employing a higher degree of speculation would imply that fewer configurations may coexist in the cache, and thus the chance of a basic block executing in the RA would also decrease.

Figure 5.10 - Average size of the RA configurations, in micro-ops.



Source: the author.

6 CONCLUSIONS

This work presented the first microarchitecture employing a reconfigurable array within a superscalar processor. The microarchitecture is composed of the RA tightly coupled to a superscalar pipeline, and exploits the fact that dynamic instruction traces appear repeatedly during program execution. Configurations for the RA store the dependency analysis for these code sequences, which is one of the major limiters of ILP exploitation is superscalar designs. By targeting the x86 architecture, one additional advantage of the proposed system arises, which is that of reducing the burden on continuously decoding CISC into RISC instructions. To translate micro-op sequences into configurations, a dynamic binary translation mechanism is employed, which allows for maintaining binary compatibility.

A high-level implementation of the BT mechanism was initially developed, which comprehended about 1500 lines of code. This implementation was used to perform a potential analysis on the system, by comparing execution on the RA (using trace-driven simulation) against execution on the superscalar (applying a set of benchmarks to the Multi2Sim simulator). The BT translation algorithm analyzed the trace of instructions which executed on the superscalar simulator and generated configurations from these instructions; the number of levels and micro-instructions in these configurations was counted and the number of uIPC was obtained. The results seem to support the fact that the RA performing speculative executing has a larger instruction window where to look for parallelism than the superscalar, which provides the former a higher potential to exploit ILP than the latter.

After the potential analysis, the BT mechanism was added to Multi2Sim, and a prototype of the RA was developed and also added (about 2000 lines were modified in the simulator). The same set of benchmarks was applied to Multi2Sim and the number of uIPC was compared in a situation where the RA was not used and a situation where it was used (with different degrees of speculative execution). This second analysis presented a few problems, mainly because of a difficulty in putting both systems on a same baseline for comparison. However, a few conclusions can be drawn from the results.

First, speculative execution is a fundamental requirement to allow the RA to improve performance over the superscalar. However, there seems to be an optimum point in the degree of speculative execution performed by the RA, after which the gains in performance are marginal (this point seems to be about 6 or 8 basic blocks speculated). This happens because

eventually the speculative scheme becomes so aggressive that the additional BBs executed will be discarded due to misspeculation.

Second, memory accesses may present a problem to the RA. In a superscalar processor, while memory accesses are performed the processor may keep reading micro-ops from the instruction queue and executing them, as long as they are independent of that memory access. In the RA, in case a memory access is performed at level N, level N+1 cannot execute until the memory access completes. All operations which are independent of the memory access are scheduled for execution on a cycle lower than N, so from this perspective there is no performance loss with respect to the superscalar. The problem arises with micro-ops that depend on another micro-op, which is also scheduled to level N, but which are independent of the memory access: these micro-ops have to wait before executing, even though their input dependencies have already been resolved.

One possible solution for the memory problem, which has yet to be investigated, is that of increasing the number of levels in the array which span a memory operation. By doing so, it is expected to alleviate the loss of performance caused by the RA waiting for memory operations to complete, because the execution of other independent instructions can be anticipated.

In case investigation of this microarchitecture continues, it is possible to suggest following works. An important next step would be to analyze system performance in a scenario where the desired assumptions hold (rather than estimating the performance in that case, as was done in this work). Next, the impact of memory operations should be analyzed, considering other RA and superscalar setups where the amount of simultaneous memory operations performed per cycle is reduced. Information on the impact of assuming a limited number of functional units within the RA is also important; for this, the binary translation mechanism implemented would also have to be modified. Finally, issues with a finite configuration cache could also be addressed.

The area and power requirements of the system should also be analyzed in detail. An initial estimation for the area of the system could be made by counting the number of bits required in the configuration cache and the number of functional units required in the RA. Because the RA execution is more efficient than superscalar execution, area for the regular instruction or trace caches could be reduced in favor of more area for the configuration cache. Likewise, structures in the pipeline could be reduced to allow for the area for the RA. As for the energy consumption, it could also be estimated by counting the amount of operations

executed on the RA, the amount of look-ups on the configuration cache, and the amount of execution traces translated.

REFERENCES

Altman, E. R., Kaeli, D., & Sheffer, Y. (2000). Welcome to the opportunities of binary translation. *Computer*, *33*(3), 40–45. doi:10.1109/2.825694

Barton, C., Chen, S., Chen, Z., Diop, T., Gong, X., Gurfinkel, S. et al. (2014). The Multi2Sim Simulation Framework - User Guide. Retrieved November 20, 2014, from https://www.multi2sim.org/files/multi2sim-v4.2-r357.pdf

Beck, A. C. S., & Carro, L. (2010). *Dynamic Reconfigurable Architectures and Transparent Optimization Techniques*. Springer. Retrieved from http://www.springer.com/engineering/circuits+&+systems/book/978-90-481-3912-5

Beck, A. C. S., Rutzig, M. B., Gaydadjiev, G., & Carro, L. (2008). Transparent reconfigurable acceleration for heterogeneous embedded applications. In *Proceedings of the conference on Design, automation and test in Europe - DATE '08* (p. 1208). New York, New York, USA: ACM Press. doi:10.1145/1403375.1403669

Binkert, N., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N. et al. (2011). The gem5 simulator. *ACM SIGARCH Computer Architecture News*, *39*(2), 1. doi:10.1145/2024716.2024718

Clark, N., Kudlur, M., Mahlke, S., & Flautner, K. (2004). Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. In *37th International Symposium on Microarchitecture (MICRO-37'04)* (pp. 30–40). IEEE. doi:10.1109/MICRO.2004.5

Compton, K., & Hauck, S. (2002). Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys*, *34*(2), 171–210. doi:10.1145/508352.508353

Patterson, D. A., & Henessy, J. L. (2006). *Computer Architecture: A Quantitative Approach* (4th ed.). Morgan Kaufmann.

Dixon, M., Hammarlund, P., Jourdan, S., & Singhal, R. (2010). The Next Generation Intel® Core™ Microarchitecture. *Intel Technology Journal*, *14*(3), 8–28.

Gonzalez, A., Tubella, J., & Molina, C. (1999). Trace-level reuse. In *Proceedings of the 1999 International Conference on Parallel Processing* (pp. 30–37). IEEE Comput. Soc. doi:10.1109/ICPP.1999.797385

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., & Brown, R. B. (2001a). 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)* (pp. 3–14). IEEE. doi:10.1109/WWC.2001.990739

Intel. (2005). Dual Core Era Begins, PC Makers Start Selling Intel-Based PCs. Retrieved November 06, 2014, from http://www.intel.com/pressroom/archive/releases/2005/20050418comp.htm

Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling* (1st ed.). Wiley.

Lipasti, M. H., & Shen, J. P. (1996). Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture* (pp. 226–237). IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=243846.243889

Lysecky, R., Stitt, G., & Vahid, F. (2006). Warp Processors. *ACM Transactions on Design Automation of Electronic Systems*, *11*(3), 659–681. doi:10.1145/1142980.1142986

Moore, G. E. (1965). Cramming More Components Onto Integrated Circuits. *Electronics*, 114–117. doi:10.1109/JPROC.1998.658762

Olukotun, K., & Hammond, L. (2005). The future of microprocessors. *Queue*, *3*(7), 26. doi:10.1145/1095408.1095418

Patel, A., Afram, F., Chen, S., & Ghose, K. (2011). MARSS: a full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference on - DAC '11* (pp. 1050–1055). New York, New York, USA: ACM Press. doi:10.1145/2024724.2024954

Perais, A., & Seznec, A. (2014). Practical data value speculation for future high-end processors. In *Proceedings - International Symposium on High-Performance Computer Architecture* (pp. 428–439). IEEE Computer Society.

Sodani, A., & Sohi, G. S. (1997). Dynamic instruction reuse. *ACM SIGARCH Computer Architecture News*, *25*(2), 194–205. doi:10.1145/384286.264200

Sodani, A., & Sohi, G. S. (1998). Understanding the differences between value prediction and instruction reuse, 205–215. Retrieved from http://dl.acm.org/citation.cfm?id=290940.290983

Stallings, W. (2010). *Computer Organization and Architecture: Designing for Performance* (8th ed.). Pearson Prentice Hall.

Ubal, R., Jang, B., Mistry, P., Schaa, D., & Kaeli, D. (2012). Multi2Sim: a simulation framework for CPU-GPU computing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT '12* (p. 335). New York, New York, USA: ACM Press. doi:10.1145/2370816.2370865

Wall, D. W. (1991). Limits of instruction-level parallelism. *ACM SIGPLAN Notices*, *26*(4), 176–188. doi:10.1145/106973.106991

Wendel, D., Kalla, R., Friedrich, J., Kahle, J., Leenstra, J., Lichtenau, C., … Zyuban, V. (2010). The power7TM processor SoC. In *2010 IEEE International Conference on Integrated Circuit Design and Technology* (pp. 71–73). IEEE. doi:10.1109/ICICDT.2010.5510286

Wilcox, K., & Manne, S. (1999). Alpha processors: A history of power issues and a look to the future. In *Proceedings of the Cool-Chips Tutorial, held in Conjunction with the International Symposium onMicroarchitecture*. New York: ACM/IEEE.

APPENDIX A - MULTI2SIM X86 SOURCE TREE

	Modified
m2s-src	
arch	
x86	
asm	
emu	
timing	
• branch-predictor	
• binary-translation	Added
• bt-queue	Added
• commit	
• core	
• cpu	Modified
• decode	Modified
• dispatch	Modified
• event-queue	
• fetch	Modified
• fetch-queue	
• functional-units	
• instr-queue	
• issue	
• load-store-queue	
• mem-config	
• reconfigurable-array	Added
• ra-config	Added
• ra-config-cache	Added
• recover	Modified
• reorder-buffer	
• reg-file	
• thread	
• scheduler	
• trace-cache	
• uop	Modified
• uop-queue	
• writeback	

APPENDIX B - GRADUATION PROJECT I

A new reconfigurable architecture for x86 processors

Marcelo Brandalero

Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91501-970 – Porto Alegre – RS – Brazil

mbrandalero@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 to match the performance requirements from applications. This work proposes investigation of a new architecture for x86 processors, based on a traditional superscalar design coupled to a reconfigurable array. A detailed model of the architecture shall be developed, from which performance and area estimations will be derived. These results will then be compared against a traditional superscalar architecture. The expected results are improved performance, energy savings and few area overhead.*

1. 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 a TDP limitation 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 (Flynn & Hung, 2005)(Olukotun & Hammond, 2005)(Borkar & Chien, 2011).

The key to achieving more performance is to efficiently exploit on chip the parallelism available from software. Applications present, by construction, parallelism at different levels: instructions, threads and data are some of them. Performance is constrained both by the inherent parallelism that the application presents and the hardware features that are implemented to explore it.

The first level of parallelism that was exploited over the years was instruction level parallelism (ILP). An application's degree of ILP reflects how often multiple instructions can be executed concurrently. Superscalar architectures were developed to exploit this form of parallelism, by including in the processor multiple pipelined functional units. However, every application presents data dependencies which are a natural part of any computation, and set an upper bound on the amount of ILP that can be exploited by hardware (Wall, 1991). This bound can hardly be reached, 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, single-threaded performance will increase very little in the following decades, due to the aforementioned discussion (Borkar & Chien, 2011). This has led to a switch in design focus to exploring both ILP, up to a certain level, but also thread-level parallelism (TLP).

An application's level of TLP reflects the degree in which multiple execution flows, or threads, can be performed in parallel. By executing multiple threads simultaneously on a processor, idle units that were not used by one thread may be used by another, enhancing the instruction throughput. Some applications, however, are inherently sequential, thus not presenting concurrent tasks, or presenting concurrent tasks with workload misbalances. These applications take little benefit from features that exploit TLP. It is, therefore, still of practical interest to find new means to increase the ILP, in addition to TLP, as doing so may provide performance gains for all applications.

In this sense, research has been done to develop new technologies that improve ILP. One of the developed approaches consists of finding repeating executed kernels within a set of applications and developing dedicated instructions to execute that specific kernel. This is known as instruction set customization (Clark, Zhong, & Mahlke, 2003). The downside of this approach is that performance gains can only be achieved for a specific set of applications, which is clearly not suitable for general-purpose processors. Besides, these applications must be recompiled towards the modified architecture to take benefit of the new instructions.

Another solution consists of implementing on die a reconfigurable circuit, such as an FPGA, that can be configured at runtime by the application. This way, flexibility is provided, as different workloads may use this feature and present performance gains. The application designer must determine during the application design phase which computation kernels are to be mapped into the circuit and which will be executed by the main processor - a process named hardware/software partitioning (Lysecky et al., 2006). To determine these kernels, usually application profiling is performed before deploying the final system. On the program binary, special instructions have to be inserted, which specify which code sequences are to be executed on the reconfigurable fabric. This process, however, being static, requires special compilers and tools, breaking the binary backward-compatibility.

Some techniques (Stitt & Vahid, 2002)(Clark et al., 2004)(Beck et al., 2008) have been developed to allow the use of reconfigurable computing with dynamic hardware/software partitioning, with a special technology responsible for doing the discovery of execution kernels at runtime. These techniques address three key issues: 1) Binary compatibility is maintained and transparency is provided to the programmer, because execution kernels are determined and optimized inside the processor at runtime; 2) Performance gains are provided for varying workloads, as the system is capable of reconfiguration at runtime; 3) Energy efficiency can be achieved, as repeating code kernels are mapped to and executed in combinatorial logic. However, to provide dynamic discovery, several changes have to be made into the processor microarchitecture. Many features that are already present may have to be replaced by the mechanism implementing the reconfigurable logic, due to area constraints. This hinders the development of new processors that make use of this technology.

Based on the discussion above, we present a new architecture that implements reconfigurable computing and also maintains binary compatibility with the x86 architecture. Because the architecture considers the underlying organization of x86 processors, the implementation is simplified and the system presents few area overhead. We use the micro-ops generated from the x86 instruction decoder as input to a binary translation mechanism which performs the mapping of instructions into a circuit configuration. This configuration is stored in a configuration cache, which replaces the traditional trace cache. By caching decoded instructions ahead on the pipeline, the need to fetch and decode instructions for the same code sequence is disabled, providing performance gains and energy savings. The reconfigurable circuit may then be used to execute hot spots that are dynamically detected.

In this work, we show the proposed architecture, as well as results on its potential performance. It proceeds as follows. On section 2, a review of other work regarding instruction level parallelism and reconfigurable computing is presented. On section 3, we present the proposed architecture for the system, discussing the x86 architecture and providing a brief overview of the reconfigurable system. Section 4 focuses on preliminary performance results, comparing these with the performance of a traditional superscalar architecture. Section 5 discusses the work to be done on further investigating the architecture and concludes this paper.

2. Related work

Early studies on instruction level parallelism have determined there are upper bounds on the amount of parallelism available from applications. Wall (Wall, 1991) presents a study on these limits for a given set of applications and an architecture. It considers five processor models, ranging from a very simple and inefficient one to a more sophisticated and powerful, considering architectural features such as branch prediction, register renaming, memory aliasing detection and out-of-order execution. It is shown that the limits of ILP could be as high as 50 instructions per cycle, when all hardware features are implemented and work ideally; this bound, however, is much stricter than this on a real processor. We conduct later on this paper an experiment similar to the one of Wall, but considering additional features, such as the possibility to execute multiple dependent ALU instructions in one cycle. This is only possible when replacing traditional, sequential code execution with combinatorial logic, thus allowing the ILP barrier to be crossed.

With respect to reconfigurable computing, vast literature has been produced. It is concept that fills a gap in computation: it is typically unfeasible to achieve high performance and simultaneously provide flexibility. Hardwired solutions, such as 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 are 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 a Field Programmable Gate Array, which the processor can program and use for execution. A survey on aspects of reconfigurable computing is presented by Compton and Hauck in (Compton & Hauck, 2002). Aspects such as system classification with respect to processor coupling, reconfiguration times and granularity of the execution units are discussed, but no experimentation is performed.

Most studies 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 (Altman et al., 2000), 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.

Many implementations of reconfigurable systems exist. Table 5 shows a comparison between some of the work discussed next. Stitt et al. (Lysecky et al., 2006) presents 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. (Clark et al., 2004) presents the use of a configurable compute accelerator (CCA) as a specialized function unit, to optimize 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; however, no discussion with respect to area overhead or reconfiguration times is provided.

Beck et al. (Beck et al., 2008) presents 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. (Fajardo, Rutzig, Carro, & Beck, 2013), 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. This approach still provides moderate gains with no additional energy consumption; however, the implementation of such a system requires great redesign as it does not take advantage of features already provided by processors implementing the x86 ISA.

Table 5. Comparison of systems proposed in previous work and system proposed in this work.

	Stitt et al. (Lysecky et al., 2006)	**Clark et al.** (Clark et al., 2004)	**Beck et al.** (Beck et al., 2008)	**This work**
Discovery	Dynamic	Static or dynamic	Dynamic	Dynamic
Reconfigurable Unit	Fine-grained (FPGA)	Coarse Grained Matrix of functional units	Coarse Grained Matrix of functional units	Coarse Grained Matrix of functional units
Configuration	CAD Tools	Subgraph replacement in code	Binary translation	Binary translation
Area Overhead	CAD processor, array	Not discussed	Reconfiguration cache, binary translation unit, array	Binary translation unit, array

In this work, we propose a new system implementing the x86 architecture which uses a coarse-grained reconfigurable array to optimize the execution of critical regions. We make use of the x86 instruction decoder to provide simpler instructions that can be easily mapped to the reconfigurable array, thus avoiding complicated microarchitectural changes. By making use of this feature, we expect to achieve better results than the ones presented on previous works.

3. Proposed architecture

We briefly describe the characteristic features of the x86 architecture, focusing on the ones that make it attractive for our approach; second we describe how the reconfigurable array works and finally we show how the array may be accommodated inside the x86 pipeline.

3.1. x86 architecture

The x86 architecture dominates on the general purpose market. It is a CISC architecture (Complex Instruction Set Computing), meaning 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 usually present variable length and each of them performs a different number of operations. To cope with it, x86 processors use a scheme in which CISC instructions are decoded into multiple RISC-like instructions, named micro-ops (Henessy & David A. Patterson, 2011). 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 x86 processors is the presence of a trace cache. A trace cache works similarly to an instruction cache, except it aims to explore much more of the temporal locality of data. Whenever an instruction has to be fetched from memory, a regular instruction cache works by fetching the requested instruction as well as all subsequent instructions that are mapped into the same cache block. This may lead to inefficiencies: a cache block may contain a branch instruction that is always taken. If that is the case, the instructions that follow the branch are needlessly taking up space in the instruction cache. A trace cache addresses this issue by caching entire sequences of basic blocks instead of instructions. 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 (Rotenberg, Bennett, & Smith, 1996).

On the latest editions of x86 processors, another characteristic feature has been added. The Loop Stream Detector is a mechanism that detects small, recurring loops in code. 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, providing energy savings (Dixon et al., 2010).

Our work uses an approach similar as the one provided by the Loop Stream Detector, combining the three features described above. A binary translation algorithm works on the micro-ops provided by the decode stage to map them into a configuration to be executed on the array. The trace cache is replaced by a configuration cache that stores the basic blocks that will be executed on the array. We detect recurring loops in code and execute them on the reconfigurable array, instead of on regular functional units.

3.2. Reconfigurable Array (RA)

A general overview of the array organization is presented in Figure 11. 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, and 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 choose as input to each functional unit any of the results on the line below. On the output context, multiplexers choose the correct values produced on the last level of the array to be written back to the register bank.

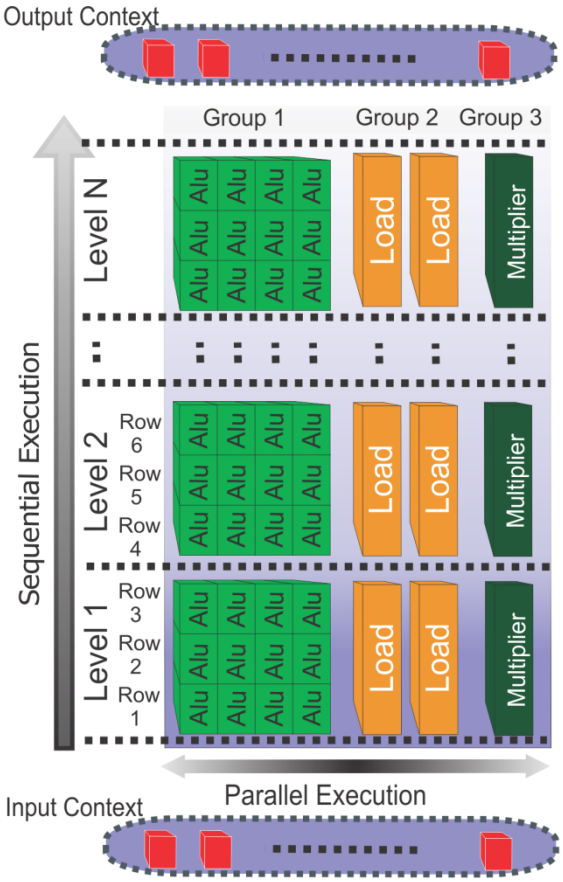


Figure 11. General overview of the reconfigurable array.

When a configuration is loaded into the array, the operation of each cell is determined and the multiplexers are configured to allow the proper data flow between the functional units. This eliminates every write-after-write and write-after-read dependency, because registers are only written-back after an entire block execution.

The array has potential to speed up applications, when compared with execution on a traditional superscalar architecture. Two are the reasons: first, the amount of functional units for each type of ALU operation is flexible, as their operation can be configured at runtime. Second, multiple dependent ALU operations can be performed on the same processor cycle, unlike traditional architectures. Because of the flexibility provided, the use of the RA on general purpose systems can increase the average performance for all applications, unlike the use of an ASIC.

Besides the potential performance gains, energy efficiency can also be provided. After configuration, execution on the RA resembles execution on a data-flow machine. By allowing multiple dependent ALU operations to be executed each cycle, sequential logic is replaced by combinatorial logic by eliminating intermediate flip-flops and registers. Besides, all complex logic required for dependency checking on multiple-issue designs is performed only once, thus allowing even more energy savings.

3.3. Coupling the RA to the x86 processor

Our proposed architecture is composed of the x86 processor, with the RA tightly coupled to the pipeline as another functional unit. This is shown in **Error! Reference source not found.**. This simple model for the x86 pipeline is composed of 6 stages: instruction fetch, decode, dispatch, issue, commit and write-back. 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. The configuration in the cache is indexed by the memory address of the first instruction in the basic block.

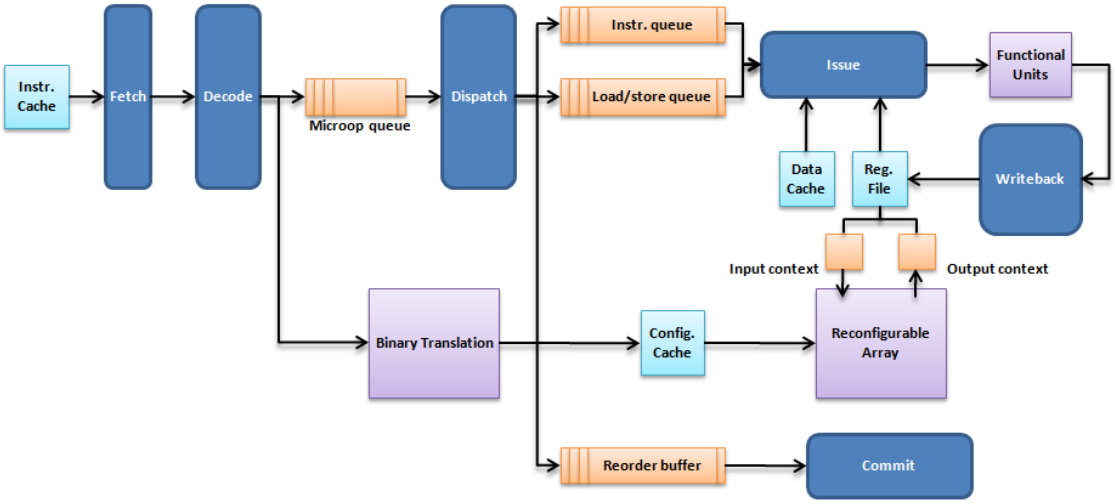


Figure 12. Overview of the proposed system.

Execution of the micro-ops generated continues 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 reorder buffer. Finally, on the commit stage, the operation results are written to the register file as soon as the operations 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 instruction cache, then the fetch, decode and dispatch stages are disabled, the configuration is loaded to the array and the basic block is executed using only the array. When a branch instruction is executed within that configuration, two situations are possible: if the target branch address is already mapped into a configuration which is in the cache, the first pipeline stages are disabled and the configuration is loaded to the array and executed; if not, then the instruction fetch engine resumes execution from the branch target address.

The array can also be extended to support speculative execution, in which a sequence of basic blocks can be mapped to a same configuration. This allows for the exploitation of parallelism across basic block boundaries. One possibility is to use an approach similar to the one proposed in rePLay (S. J. Patel & Lumetta, 2001). Branches that are biased towards a result are dynamically detected and converted into an assertion statement; next, the two basic blocks that execute together often are merged into a single circuit configuration. During the execution, the assertion statements are responsible for maintaining correct execution order. An investigation on how to properly address speculation shall be done, considering correct program execution, exception behavior and recoveries in case of misspeculation.

4. Preliminary results

We present a preliminary study on the performance of the proposed system. The final goal is to evaluate the performance of the microarchitecture proposed by us and the microarchitecture for a typical x86 processor. To achieve this goal, we choose a set of benchmarks and estimate the number of instructions executed per clock cycle (IPC) for each application in the suite for both architectures. Since our interest lies in the microarchitectural performance and the underlying ISA is the same, IPC is a good indicator for performance.

As our benchmark suite, we chose the applications in MiBench (Guthaus et al., 2001b) because it addresses a wide variety of applications, both control- and data-oriented, and because and all applications are single threaded (Guthaus et al., 2001b). This last reason fits well into our needs: because our solution is targeted towards improving parallelism at the instruction level, gains should be presented even for single-threaded applications. Each of the benchmarks were compiled on a Linux operating system using gcc v4.4 with -static, -O3 and -m32 flags. Some of the benchmarks presented compile errors and were left out of the tests. The benchmarks are to be executed on two simulators: one modeling the entire x86 pipeline and the second modeling our system.

There are three key requirements for the x86 simulator: it must support execution of all of our benchmarks; it must offer the modeling of the processor microarchitecture, including micro-ops, and it must provide legible code. The Multi2Sim simulator (Ubal et al., 2012) was chosen, as it addresses all of our requirements. Besides, it provides good documentation and reasonable simulation times.

Table 6. Different setups considered for the reconfigurable array.

Parameter	Setup 1	Setup 2	Setup 3	Setup 4	Setup 5	Setup 6
ALU operation latency	1 cycle	1/2 cycle	1/3 cycle	1/3 cycle	1/3 cycle	1/3 cycle
Max. memory operations per cycle	Unlimited	Unlimited	Unlimited	4	2	1

To estimate the potential performance of our architecture, we developed a simulator for the execution unit. This simulator implements an instruction scheduler for the reconfigurable array. An execution trace of micro-ops for each application in the benchmark is obtained from the Mutli2Sim simulator and is fed into the simulator. Our simulator then reads entire basic blocks from the trace and schedules them one-by-one for execution, based on true data dependencies only. We considered multiple setups, modifying parameters such as the latency of each ALU instructions and the number of memory operations allowed per cycle. These setups are presented in Table 6.

For each scheduled basic block, we count the amount of micro-ops executed and the number of cycles that the execution takes. The ratio of these values is used to provide an average IPC count. On Table 7, the average IPC observed for each benchmark executing on the reconfigurable array under the different setups is presented. 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 from allowing multiple 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.

Table 7. IPC measured for each benchmark executing on the reconfigurable array, considering different setups.

Benchmark	Setup 1	Setup 2	Setup 3	Setup 4	Setup 5	Setup 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

To compare our results with execution on a superscalar processor, we make a few assumptions in order to put both systems on the same baseline for comparison. First, since our simulator does not consider memory access latency, we configured Multi2Sim such that every memory access takes exactly one cycle. Second, Multi2Sim, as most superscalar processors, performs speculative execution. We must therefore consider additional setups in the RA on which multiple basic blocks may be executed simultaneously. Because we ignore reconfiguration times on our simulator 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, and the reconfigurable array setup taken as base for considering speculation was setup 4.

Figure 13 presents the execution results on the RA simulator, considering the aforementioned discussion. The average IPC values obtained were normalized with respect to the average IPC observed from execution on Multi2Sim (superscalar). Values higher than one, therefore, indicate performance gains over the superscalar model. As can be seen, for most applications no gain is provided when executing only one basic block at a time. This is expected, because the superscalar processor is executing multiple basic blocks simultaneously, as the branch predictor is configured to always hit. 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 executing two basic blocks at a time, 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 executed simultaneously from 3 to 5, the average normalized IPC values are of 1.32, 1.53 and 1.68. When executing 5 BBs simultaneously, all applications present performance gains, with

some applications, such as susan and jpeg decoder, performing twice as fast as the superscalar processor.

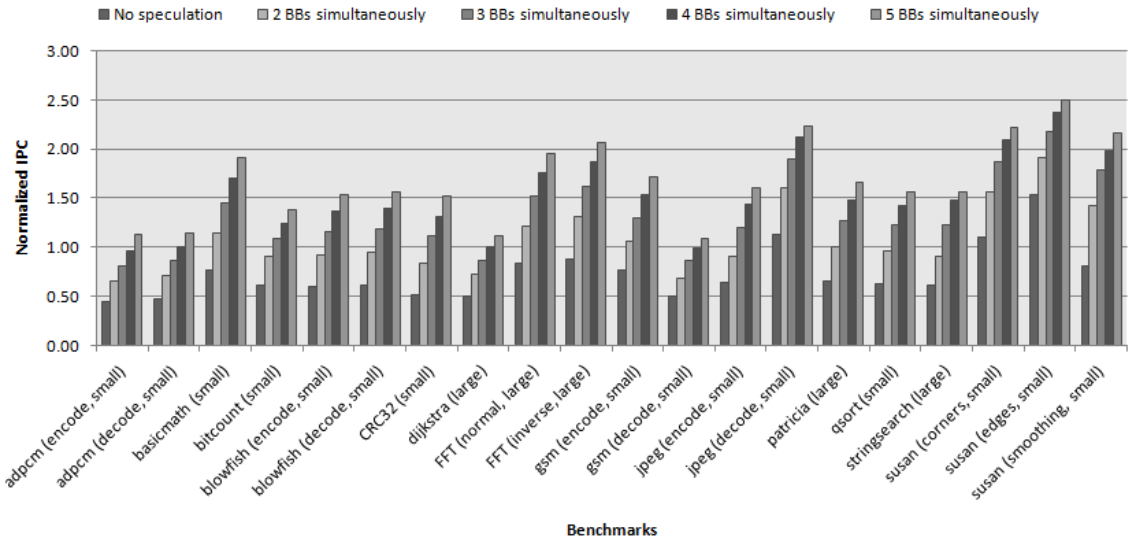


Figure 13. Relative IPC when executing the applications on the reconfigurable array.

5. Sequel

The next step of the project shall focus on how to implement the proposed architecture. A chronogram of the activities to be performed next is shown below on **Table 8**. We shall analyze the microarchitecture of existing x86 processors and properly define how to accommodate the array in the pipeline. Next, the communication between the different units will be specified, such as how the array accesses the register file, how a configuration is loaded, how control is switched to the array for execution.

With the microarchitecture properly defined, we will modify the existing x86 simulator, Multi2Sim, to model the new system. This simulator will then be used to gather more accurate performance results which will be compared to the execution on the traditional superscalar model.

The final step is to perform an area estimation of the new system. For this, a high-level model may be used or a partial implementation in hardware description language. As the expected result is to achieve minimal or no area overhead with respect to a traditional superscalar architecture, minor modifications in the microarchitecture are still possible. A final simulation to extract performance results will then be performed.

Table 8. Chronogram of the activities to be performed next.

	Jun	Jul	Aug	Sep	Oct	Nov
Detailed study of x86 superscalar organization	X					
Microarchitecture definition	X	X		X		
Integration into Multi2Sim		X	X			
Performance analysis			X	X		X
Area analysis				X	X	X

6. 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] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

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

[13] R. Bowles, S. Douglas, A. Binstock, M. Lacey, and D. L. Hill, “The Tick Tock beat of Microprocesssor development at Intel,” *Intel Technol. J.*, vol. 14, no. 3, pp. 1–164, 2010.

[14] S. J. Patel and S. S. Lumetta, “rePLay: A hardware framework for dynamic optimization,” *IEEE Trans. Comput.*, vol. 50, no. 6, pp. 590–608, Jun. 2001.

[15] 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.

[16] 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.