UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FELIPE TANUS

Exploring parallelism on pure functional languages with ACQuA

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Álvaro Moreira
Co-advisor: Prof. Gabriel Nazar

Porto Alegre
December 2017

CIP — CATALOGING-IN-PUBLICATION

Tanus, Felipe

Exploring parallelism on pure functional languages with ACQuA / Felipe Tanus. – Porto Alegre: PPGC da UFRGS, 2017.

58 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2017. Advisor: Álvaro Moreira; Co-advisor: Gabriel Nazar.

1. Architecture. 2. Parallelism. 3. Functional programming. 4. Accelerator. I. Moreira, Álvaro. II. Nazar, Gabriel. III. Título.

ABSTRACT

Moore's law reaching its physical limitations has pushed the industry to produce multicore processors. However, programming those processors with an imperative language is not easy since it requires developers to create and synchronize threads. A pure functional language is an adequate tool for this task both from the architectural point of view and from the developer's. We will show that an architecture can benefit from the implicit parallelism present on functional programs and from the lack of side effects making it easier to parallelize. The developer benefits from functional languages from the superior expressiveness of the language to avoid bugs. In this dissertation, we present the ACQuA architecture, a multicore accelerator created to explore parallelism available in function calls from a pure functional program. ACQuA uses hardware support and a specifically-tailored memory organization to minimize the overheads of scheduling, communication, and synchronization. Function calls are placed into a queue and are scheduled to different processing units. The processing units are interconnected and exchange results from function applications. In this work we defined a high level model of the accelerator and how to compile a functional program to it. We also simulated the accelerator and evaluated results, such as speedup, memory usage, and communication overhead of the proposed architecture. We defined the necessary traits of a program to achieve a good speedup on the architecture. On the ideal use case, we can increase the speed up at the same rate we increase the number of processing units in the architecture.

Keywords: Architecture. parallelism. functional programming. accelerator.

Explorando paralelismo em linguagens funcionais puras com ACQuA

RESUMO

A indústria está sendo pressionada a produzir processadores com múltiplos núcleos devido as limitações físicas da lei de Moore. Porém, programar esses processadores com linguagens imperativas não é uma tarefa fácil, já que o programador deve criar e sincronizar *threads*. Linguagens funcionais puras são ferramentas adequadas para essa tarefa tanto do ponto de vista da arquitetura e do programador. Do ponto de vista da arquitetura, essas linguagens apresentam paralelismo implícito, e são mais fáceis de paralelizar devido a falta de efeitos colaterais. Já o programador é beneficiado por não precisar definir explicitamente *threads* e a sincronização entre elas, além de aproveitar o alto nível de abstração provido por essas linguagens para evitar erros. Nessa dissertação, nós apresentaremos a arquitetura ACQuA, um acelerador de múltiplos núcleos criado para explorar o paralelismo implícito em aplicações de funções de linguagens funcionais. ACQuA têm suporte em hardware e uma organização de memória criada especificamente para minimizar a sobrecarga do agendamento, comunicação e sincronização dos núcleos. Chamadas de função são colocadas em uma fila e são agendadas para diferentes unidades de processamento. As unidades de processamento são interconectadas e trocam resultados das chamadas de função. Nesse trabalho, nós definimos um modelo para a arquitetura em alto nível, e como compilar um programa funcional para ela. Nós também simulamos o acelerador e avaliamos resultados como aceleração, uso de memória e sobrecarga de comunicação. Nós definimos as características necessárias de um programa para atingir uma boa aceleração na arquitetura. Nos casos de uso ideais, a aceleração aumenta na mesma taxa em que mais unidades de processamento são adicionadas na arquitetura.

Palavras-chave: arquitetura. paralelismo. linguagem funcional. acelerador.

LIST OF FIGURES

CONTENTS

1 INTRODUCTION

With future processors being expected to have hundreds of cores, it becomes impractical for a developer to manually divide and synchronize parts of a program to be executed in parallel, making automatic parallelization all the more important. Meanwhile, the absence of side effects in programs written in purely functional languages simplifies the task of identifying parts of a program that can be executed in parallel (HAMMOND, 2011).

To take advantage of that, we present the ACQuA (Active Call Queue Architecture) accelerator, an architecture that explores some of the inherent parallelism in purely functional languages by automatically parallelizing independent function applications, also known as function calls. With ACQuA's direct architectural support, a developer can write programs in a pure functional language and benefit from the parallelism in independent function applications transparently, without having to resort to language abstractions to explicitly create, synchronize or describe the communication among multiple threads. The architecture also features list support, since lists and functions to process them are present in any functional programing language and are widely used in this programming style. In particular ACQuA provides support for mapping a function to the elements of a list since this operation has the potential of spawning several independent function calls.

The main goals of this work are to introduce the ACQuA accelerator architecture, to evaluate its performance and scalability, and also to identify which are the properties that make programs more or less suitable to perform well on the proposed architecture.

The idea of having architectural support specific for executing functional languages is not new (SCHEEVEL, 1986; STOYE; CLARKE; NORMAN, 1984; RHO et al., 1994; VEGDAHL, 1984). This idea, however, lost its strength due to the fast rate of improvement of conventional processors which was a reality until recently. With the increasing difficulty of improving single-threaded performance and the rise of multicore processors, we believe this subject becomes again worth discussing.

Dedicated hardware modules and a specifically-tailored memory organization were devised in ACQuA to shorten the gap between the high-level abstraction of function applications and the architecture. Thereby, architectural support is provided to minimize the overheads of creating, scheduling and synchronizing tasks.

Pure functional languages are arguably easier to parallelize thanks to the lack of states and side effects. Thanks to this property, two function calls without explicit data

dependency from pure functional languages can always be evaluated in parallel, and a function called multiple times with the same argument always return the same value. On imperative languages we can not make such claims, since a function can modify a global state (JONES, 1989). There are other advantages on using functional languages that are also important to critical software modules. The usage of functional languages allows the developer to express complex programs with less lines of code than it does with imperative languages. The number of correct lines of code a programmer can write does not seems to depend on the language an developer is using (WASSERMAN; GUTZ, 1982), so using more expressive languages help to reduce the number of lines, and thus the number of bugs in a software. Having less bugs is specially useful for parallel programs, which are a source of high impact, hard to fix bugs (ASADOLLAH et al., 2016). Also, functional languages are easier to verify (BACKUS, 1978), which is an useful property for critical software modules.

The architecture is based on a queue of function calls and multiple processing units. It is currently proposed as an accelerator that operates when activated by a program running on a general-purpose host processor. With this accelerator-based approach, the programmer is responsible for identifying which parts of a program are critical regarding performance. These parts, if written in a pure functional language, can have their execution accelerated by ACQuA inside an impure software.

The accelerator is programmed with a small pure functional programming language, which is compiled to ACQuA_{IR}, a lower level language with specific commands, such as to enqueue and to synchronize non-independent function applications. Since ACQuA excels when multiple independent function calls execute in parallel, the architecture also provides native support for mapping a function to the elements of a list.

We have implemented a simulator for the architecture to evaluate its viability and performance. Experiments were conducted to collect data about speedup, memory usage, and communication overhead. Some experiments have outstanding results, for instance, the execution of the classical *longest common substring* algorithm achieved a speedup of 14 when running with 16 processing units.

The dissertation is structured as follows: In Chapter 2, we present the history of parallelism and how its exploration has advanced, be it with computer language features or architectural support. The accelerator architecture is explained at Chapter 3. In this chapter we will explain the ACQuA architecture and how a program run on it, explaining and justifying project decisions. In chapter 4 we present the accelerator intermediate rep-

resentation language ACQuA$_{IR}$, and explain how the core of a pure functional language can be compiled to this intermediate representation. In Chapter 5, we present the result of experiments done for this architecture on a simulator. Chapter 6 has our conclusions and a discussion about future work.

2 RELATED WORK

2.1 Parallelism on programming languages

On most programming languages, parallelism needs to be explicitly expressed on the code. For a long time the C language did not have built in support for parallelism, which could only be obtained by using system libraries to create and synchronize threads, like POSIX (STANDARD..., 2016) for unix based systems. C11 (ISO, 2011) has introduced a standard library to create and synchronize threads, making easier to share a parallel code through different operational systems, however the only way to achieve parallelization on C is still through threads.

Concurrency is the source of high impact, hard to fix bugs (ASADOLLAH et al., 2016), thus there is an effort to simplify the way we express concurrent and specially parallel programs. One interesting case is present in the Haskell programming language. While it has support for creating and synchronizing threads through Concurrent Haskell (JONES; GORDON; FINNE, 1996), it also has the `par` construct to explicitly evaluate two expressions in parallel (MARLOW; NEWTON; JONES, 2011). Having only one simple command to execute expressions in parallel makes it easier to understand the code since it abstracts threads synchronization, unlike traditional constructs such as POSIX threads and monitors which might execute code with complex synchronization.

More complex abstractions that remove the need of creating threads have been used by modern impure languages, for instance the actor model (HEWITT; BISHOP; STEIGER, 1973). In the actor model, each concurrent computation is an actor, and synchronization is achieved through messages. Each actor has a private state that can only be modified by the actor itself, thus avoiding the need for any lock. This model is implemented by Erlang (SALVANESCHI; GHEZZI; PRADELLA, 2012), Scala and Rust, for instance. While this model is less explicit to create parallel code than than Haskell's `par` construct, it can be used to implement complex synchronization. In comparison with traditional concurrency control methods like POSIX threads, it simplifies the synchronization by hiding hardware and operational system synchronization details like semaphores.

While there is an effort by modern programming languages to express the parallelism in a simpler way, there is still some hard work for the developer, for instance identifying the sources of parallelism of a program and making it explicit in the source code how the program should be parallelized. Recognizing a parallelism opportunity is

not always trivial, and parallelizing the wrong part of a source code might hurt the performance and/or introduce bugs.

2.2 Implicit parallelism exploration

One way to alleviate the burden on the developer is through implicit parallelism, that is, achieving parallelism on a program with no special commands, leaving the compiler to decide where and when to execute code in parallel.

One possible way to explore implicit parallelism is to perform a code transformation, including parallelism commands on a code that originally had none. It is hard to identify safe parallelization opportunities on imperative languages because of states. The simplest transformation that can be done in compile time for imperative languages is parallelizing loop iterations. For instance, Cetus (DAVE et al., 2009) is a C source-to-source compiler that explores implicit parallelism on loops. The ICU-PFC compiler (KIM et al., 2000) transforms Fortran code to include OpenMP directives.

There is also effort to explore the implicit parallelism of programs on the distributed memory model through code transformations. For instance, Jones (AUBREY-JONES; FISCHER, 2016) shows a technique to compile pure functional languages to MPI C++ code able to run on Linux clusters. By using pure functional languages it was possible to parallelize operations on arrays, maps and lists in a simpler fashion than on impure languages.

Exploring implicit parallelization does not need to be done at compiler level. It can be explored in the interpreter or virtual machine level. For instance, there is a version of the Java Virtual Machine (JVM) that explores the parallelism on method executions (CHEN; OLUKOTUN, 1998), which is tricky because of imperative programming side effects. Conversely, ACQuA explores the parallelism on function applications, which are similar to method executions from object-oriented languages, but are guaranteed to lack side effects. Also there is the JRPM (CHEN; OLUKOTUN, 2003), a version of the JVM that identifies which loops are worth being parallelized by thread-level speculation.

Instead of trying to extract the implicit parallelism on existing non-parallel languages, some programming languages are designed to be parallel. This is the case of Id (SHARP, 1992), a dataflow programming language (JOHNSTON; HANNA; MILLAR, 2004) which by definition evaluates some expressions in parallel, for instance function arguments or strict operator arguments. It does not specify a machine or interpreter for

the language, only the evaluation strategy, thus its performance is highly dependent on the system on which it is running. The only way to achieve parallelization on the current commercial processors and operational systems is through threads. Creating and synchronizing threads have a cost that limits the grain of effective parallelization, making it harder for lower grain parallelization efforts like the Id language to succeed.

2.3 Architectural support for functional programming languages

The idea of having a specific architectural support for executing functional languages is not new (VEGDAHL, 1984). Some efforts were made on the design of Lisp machines (SCHEEVEL, 1986) and processors based on the notion of combinators (STOYE; CLARKE; NORMAN, 1984). The DAVRID architecture (RHO et al., 1994) dates from that time and uses a functional language augmented with explicit constructs for loop unfolding to achieve some degree of parallelization. There is also a work that utilize multiple cores to speculatively execute possible program paths when reaching conditional or branch instructions (WATERLAND et al., 2014). While the work achieves good results, this approach is probably expensive in energy for programs with multiple conditions since it executes unnecessary code.

More recently, a special-purpose and FPGA-based processor called Reduceron (NAYLOR; RUNCIMAN, 2010) has been proposed to explore implicit low-level parallelism purely functional programs. It sees a pure functional program as a graph and executes the program by reducing the graph. It explores every tiny parallelism in the reduction by using a stack and parallel memories. ACQuA works on a coarser grain to avoid communication overhead. It also has a hardware implementation on an FPGA, while ACQuA has a simulator. Reduceron was implemented with parallel memories which are expensive in area, and those are fundamental to the results they got. ACQuA uses standard memory which escalates better, and because of that, it does not explore all parallelism in a functional program, focusing only on function applications. The architecture from Reduceron is totally new, while ACQuA uses standard cores to compute the program, being able to take advantage of advances in microarchitectural design, if desired.

Reduceron inspired some other works in the area, including the PilGRIM (BOEIJINK; HÖLZENSPIES; KUPER, 2010). This processor core also reduces a graph that represents the functional program, but unlike Reduceron it focuses on lazy evaluation. There is no physical implementation for PilGRIM, only a simulator, just like ACQuA.

Both Reduceron and PilGRIM present a small functional core to accelerate, not considering ample used data structures like lists, which ACQuA do.

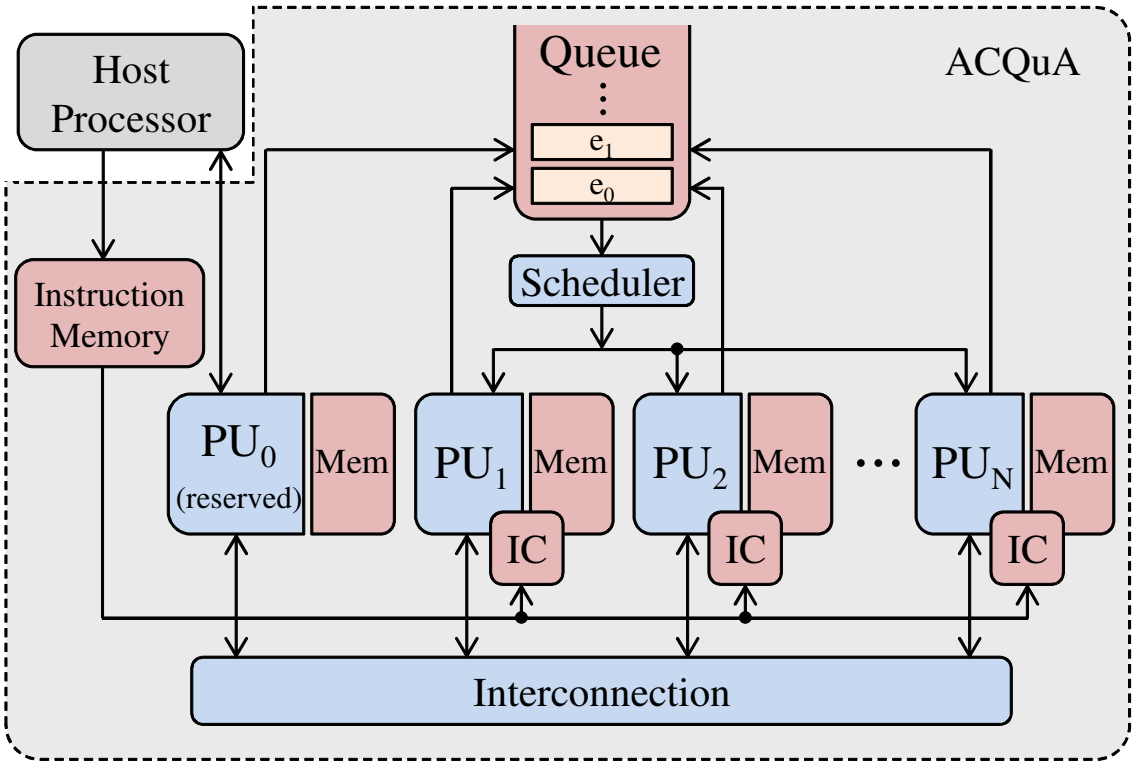
The most similar architecture with ACQuA we found was the Data Flow Multiprocessor (RUMBAUGH, 1977). This processor features many similar architectural components, including multiple memories, cores and a scheduler. Unlike ACQuA, it considers a functional program as a data flow, thus having nodes executing operations when data is available. It also uses a really small grain, parallelizing math operations. As far as we know, no simulator was written to test this processor. On our experiments, we will conclude that ACQuA does not perform well on such a small grain, and because the architecture is very similar, we believe that this processor would not perform well if implemented to parallelize the operations described in the paper.

As far as we know, ACQuA is the first architecture proposed specifically for supporting the automatic parallelization of function application in purely functional languages with native support to list processing.

3 ARCHITECTURE

The ACQuA architecture is presented in the following three sections. In Section 3.1 we explain the architecture functionality, hiding details and optimizations, with the objective of showing the basic execution flow. Afterwards, in Section 3.2, we lower the level and we present details of the architecture. In Section 3.3, we explain some optimizations and discuss project decisions and alternatives.

Figure 3.1: ACQuA accelerator architecture overview.



3.1 ACQuA architecture overview

The ACQuA accelerator aims to automatically explore parallelism existing in purely functional programs by executing independent function calls in parallel. The functions to be accelerated are written in a pure functional language and a successful compilation results in ACQuA_{IR} code.

Figure 3.1 shows the ACQuA architecture overview. ACQuA has five main components: a *Queue* of function calls, a set of *Processing Units* ($\texttt{PU}_1$, $\texttt{PU}_2$, ... $\texttt{PU}_N$), an *Interconnection* used by the processing units to exchange data, a *Scheduler* responsible for assigning function calls from the Queue to processing units, and an *Instruction*

Memory where the host processor loads the code of functions to be accelerated. Each processing unit has its own *Memory* (`Mem`) for data and an *Instruction Cache* (`IC`) that fetches code from the Instruction Memory. The architecture also has a *Reserved Processing Unit* ($\texttt{PU}_0$), that is a special processing unit responsible for the setup and for the end of execution of function calls initiated by the host processor.

The programmer invokes the accelerator by informing in the host program the name of a pre-compiled function with its arguments. The Reserved PU then places information about this function call in the Queue from where the Scheduler eventually assigns it to an available processing unit. Subsequent function calls, reached along the execution of the original function call, are also placed in the Queue before the Scheduler assigns them to processing units. Parallelism occurs when multiple processing units are executing the code of independent functions at the same time.

Whenever a processing unit finishes the execution of a function call that was assigned to it, the resulting value is returned, through the interconnection, to the processing unit where the function call was made. When the original function application finishes, its resulting value returns to the Reserved PU which sends it to the host processor.

3.2 ACQuA architecture details

During the life cycle of a function call in ACQuA, it is necessary to keep a record of a variety of information such as the entry point address of the function's code, the address of where the resulting value should be stored, etc. Since the initial setup of a function call and its actual execution might occur at different processing units, the overhead for moving all these information through the interconnection from one PU's memory to another can be too high. Hence, to reduce communication costs ACQuA uses different data structures to maintain all the information required by a function, the most important of which are *call records*, *queue entry*s, and *execution records*.

Below we describe these data structures in detail, and we explain the life cycle of function calls in ACQuA. We assume there is a single address space for all processing units. Hence, given an address it is possible to infer in which processing unit memory it is located.

Call Record. A *call record* is the first structure created on behalf of a function. A new call record is created every time the execution flow reaches a function. It is defined as a tuple

$$\langle fn\_addr, bindings, n\_available, n\_missing \rangle$$

where

- *fn_addr* is the address in the Instruction Memory of the entry point of the function associated to the call record,
- *bindings* holds the bindings for the names that appear in the function body which are declared in the function definition as its formal parameters or outside of the function in a bigger scope.
- *n_available* holds the number of entries of *bindings* which correspond to names with their values already defined.
- *n_missing* is the number of entries of *bindings* with names with values are yet to be defined

Note that the number of entries in *bindings* is always $n\_available + n\_missing$.

When a call record is created, regardless whether this call record is going to be called immediately, passed as a parameter or returned from a function, *n_missing* holds the number number of entries in *bindings* that still needd to be completely defined before the instructions in the body of the function can be executed. As values referencing the names are added to *bindings*, *n_missing* is decremented and *n_available* is incremented. Once

Call records can hold the information for higher-order functions, both to receive functions as arguments and to return them as results. If a function fn is passed as argument, the value for this function on *bindings* will be the memory address of a call record created for it the function fn. Similarly, functions that return functions return the memory address of a call record.

n_missing reaches zero, the function is ready for execution and a *queue entry* is placed in the queue.

Queue Entry. A *queue entry* is a tuple

$$\langle cr\_addr, ret\_addr, isMap, mapParam \rangle$$

where

- *cr_addr* is the memory address of its associated call record
- *ret_addr* is the memory address where the resulting value of the function call should be stored.
- *isMap* is a flag that is set if this queue entry refers to a function call that originated from the mapping of a function to a list, and
- *mapParam* is the element of the list to which the function was applied, used only when *isMap* is set

The *isMap* and *mapParam* components are used as an optimization for the map function which will be detailed in chapter 3.3.

Execution Record. Immediately before the function's body starts executing, a structure called *execution record* is allocated in the memory of the processing unit where the function call was assigned to. This structure is filled with information some of which come

from the function's call record and queue entry. An *execution record* is defined as a tuple

$$\langle ret\_addr, callCount, exeCtxt, env \rangle$$

where

- *ret_addr* is the function's return address. It is copied from the queue entry,
- *callCount* counts the number of other functions called by this function that did not return a value yet,
- *exeCtxt* holds the execution context of the processing unit, and
- *env* is the execution memory used for temporary data, and also for the *bindings* copied from the call record

Now that the main data structures were presented we explain the main components of the ACQuA's execution model.

Life cycle of a function call. The complete life cycle of a function call in ACQuA is composed of five main steps: *creation* of call records, *activation* of call queue, *scheduling* of queue entries, and function *execution* and *termination*. Below we explain each of these steps assuming a situation depicted in Figure 3.2 where the code for the application of a function f is reached at $\mathtt{PU}_1$, and the actual execution of f's code will be assigned to $\mathtt{PU}_2$:

STEP 1 - *call record creation.* When the execution flow reaches a function f a call record `cr` is created on its behalf in $\mathtt{PU}_1$'s memory. The function's code entry address at the Instruction Memory is inserted into *fn_addr*, *bindings* is set with the bindings the compiler can statically determine and space is reserved for the bindings it can't determine, *n_missing* is initialized with the number of missing bindings for f, and *n_available* is set to zero.

STEP 2 - *call queue activation.* As values are added to *bindings*, *n_available* is incremented and *n_missing* is decremented. Once *n_missing* counter of call record `cr` reaches zero, meaning that the function has all the values it requires, a queue entry `e`, pointing to `cr`, is placed in the Queue. Note that since it may take some time to compute the values for the missing bindings step 2 does not necessarily occurs immediately after the creation

of the call record in step 1.

STEP 3 - *queue entry scheduling*. The Scheduler continuously fetches entries from the queue and assigns them to idle processing units. In this example, ACQuA's scheduler identifies that $\texttt{PU}_2$ is idle and assigns the queue entry `e` to it, deallocating `e` from the queue. At this point (step 4), $\texttt{PU}_2$ can initiate the execution of the code of function f.

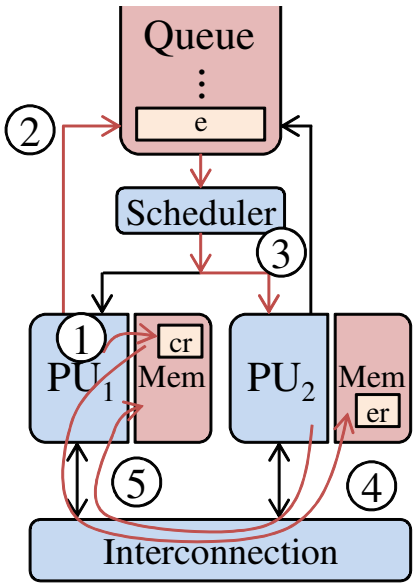
STEP 4 - *function execution.* This is the more involved of all five steps, and it consists of the following sub-steps:

(a) *call record copy* - When $\texttt{PU}_2$ receives queue entry `e`, it uses the *cr_addr* component of `e` to retrieve its call record `cr` which is copied from $\texttt{PU}_1$'s memory through the interconnection. At this moment the area occupied by `cr` is deallocated from $\texttt{PU}_1$'s memory.

(b) *execution record creation* - An *execution record* `er` is allocated at $\texttt{PU}_2$'s memory. The values contained in the *bindings* component of call record `cr` and the return address of the queue entry *e* are copied both to the *env* and to the *ret_addr* components of the execution record `er` respectively. The *callCount* counter is initialized to zero. Execution of the code of function f can finally initiate at $\texttt{PU}_2$ (sub-step (c) below). Additional auxiliary variables, such as partial values in the evaluation of an expression and return addresses for called functions, may be stored in *env* too during next sub-step.

(c) *execution and synchronization* - Whenever f calls another function which will eventually lead to a new queue entry, its *callCount* is incremented, and when a return value is received, *callCount* is decremented. Note that calling a function is a non-blocking operation. When a return value is required for the computation to proceed, a synchronization operation is necessary. These are automatically inserted by ACQuA's compiler, and cause the current function to be halted until *callCount* reaches zero, or are skipped if *callCount* already is zero. If the function is halted, the processing unit's context, including the program counter and general-purpose registers, are saved at *exeCtxt*. The processing unit then is free to execute other functions, and when it has nothing else to execute, it will first check whether it has unfinished functions that are ready to continue execution, and only then will it indicate to the scheduler that it is idle. In other words, continuing unfinished functions

Figure 3.2: Life cycle of a function call



has a higher priority then fetching new ones from the queue.

STEP 5 - *function termination*. Once *f*'s execution is completed at PU_2, its result is returned, through the interconnection, to the address in PU_1's memory indicated by the *ret_addr* component of the *execution record* `er`, concluding *f*'s call life cycle. The *execution record* `er` and the call record `cr` are deallocated from PU_2's memory.

Example 1. As a simple example to illustrate the life cycle of a function call and the role the data structures play on it, consider the functions defined in Figure 3.3 and suppose that the call `quad(10)` is placed at the host processor and the $ACQuA_{IR}$ code corresponding to the function `quad` is already present on the Instruction Memory.

Following the request of the host processor, the Reserved PU creates a call record `cr` for `quad` in its call record memory. The *fn_addr* component of `cr` is the address in the Instruction Memory of the entry point for the code of `quad`. Note that `double` is formally a free identifier from `quad`'s perspective. This identifier, however, can be statically resolved by the compiler and, therefore, it will not require an additional entry in `quad`'s call record *bindings*. Hence *n_available* component starts with 0 and *n_missing* starts with 1, which is the number of parameters `quad` needs to receive before it can be executed.

The argument of `quad` is readily available hence the value 10 is immediately added to the *bindings* of `cr`. At this moment, *n_available* is incremented to 1, and *n_missing* is decremented to zero, allowing the activation of the call queue with the in-

Figure 3.3: Example of function definitions.

```
1  fun double(x) = x * 2
2
3  fun quad(x) = double(double(x))
```

sertion of a queue entry `e` with its component *cr_addr* pointing to the call record and its component *ret_addr* pointing to a free position in the Reserved PU's memory. This free position will eventually receive the function application returning value. Since this is not a map application, *isMap* is set to zero, and *mapParam* can have any value, as it will not be used.

The scheduler then assigns the queue entry `e` to an idle processing unit. In the current state of this work, it will simply assign it to the first idle unit, $\texttt{PU}_1$, for instance, which will then fetch the call record `cr` from the Reserved PU's memory and access the Instruction Memory to retrieve `quad`'s code.

An *execution record* `er` is allocated at $\texttt{PU}_1$, with enough space to store in *env* all intermediate values, determined at compile time. The component *ret_addr* is copied from `e` and the binding $x \mapsto 10$ is copied from the call record `cr` to *env*. Both the call record `cr` and the queue entry `e` have been deallocated. The *execution record* `er` is the only descriptor left for the call `quad (10)`.

At this moment, the actual execution of the code of `quad` can begin. Its first action is to call `double`, forwarding `x`'s value to it. It will allocate a call record pointing to `double`'s code at its *fn_addr* component and with *n_available* = 0 and *n_missing* = 1. When `x` is added to the *bindings* of the new call record, *n_missing* becomes zero and a new queue entry can be placed in the queue, pointing to the new call record, and `quad`'s *callCount* is incremented to 1. The *ret_addr* in this entry will be a position in the *env* component of *execution record*. This second queue entry will be assigned to $\texttt{PU}_2$ while, in parallel, $\texttt{PU}_1$ realizes it has nothing else to do but to wait for the return value, saving its context at *exeCtxt*. $\texttt{PU}_2$ will then retrieve its call record, from where it gets the value of `x`, multiplies it by 2 and returns. With the returning value, *callCount* becomes zero and `quad` can resume its execution. It will repeat the creation of a new call record for `double`, this time placing the return value as the parameter, and enqueueing another entry. Once this second `double` call returns, `quad` can return its result to its *ret_addr*, which is at the Reserved PU. The Reserved PU then forwards the result to the host processor.

Figure 3.4: Example with higher-order functions.

```
1  fun f(x) = fn y => x + y
2  fun app_1(g) = g(1)
3  fun h(x) = app_1(f)(x)
```

Figure 3.5: Fibonacci sequence function with parallel function calls.

```
1  fun fib(x) = if x <= 2 then 1
2               else fib(x-1) + fib(x-2)
```

Example 2. Call records are also used by higher-order functions, as `app_1` and `f` in Figure 3.4. Let us assume the host issued a call for `h(2)`. The Reserved PU will allocate a call record for `h`, inserting the parameter 2 and issuing a queue entry, which will be assigned to a processing unit, e.g. $\texttt{PU}_1$. $\texttt{PU}_1$ will allocate a call record for `app_1`, with *n_missing* = 1. It will also allocate a call record for `f`, but in this case with *n_missing* = 2. The function `f` has only one parameter and can execute as soon as this parameter value is provided. The returning value would be simply an call record address for the function in the body of f, with the value of `x` included. In this case, `x` is a variable that occurs free in the function which is the body of f and that cannot be statically defined by the compiler. The compiler realizes this and it directly allocates a call record expecting two values, and with *fn_addr* pointing to code of the function in f's body. The address of this call record is added to the *bindings* of `app_1`'s call record, which has its *n_missing* decremented to zero. A queue entry for `app_1` is issued by `h`, which has now to wait for the return value.

The execution of `app_1`, which may take place at $\texttt{PU}_2$, for instance, starts by fetching `g`'s remote call record. For each time `g` is used in `app_1`'s code, this call record has to be copied, since it could potentially be applied multiple times to different parameters or forwarded to other functions. If the compiler can guarantee that `g` will be used only once, this copy could be avoided, although this optimization is considered future work. After the copy, the value 1 is added to the *bindings* of the call record, and its *n_missing* is decremented to 1, meaning it cannot be executed yet. The return value of `app_1` is, thus, the address of the produced call record. Upon receiving the return value, `h` resumes its execution by fetching the remote call record produced by `app_1`, inserting 2 to its *bindings* and decrementing *n_missing* to zero. A queue entry can now be issued, and the code of `f2` will add 1 and 2. The final return value is forwarded by `h` to the Reserved PU and then to the host processor.

Example 3. The previous two examples showed ACQuA's basic operation but lacked

parallelism. The interdependence among function calls causes ACQuA's compiler to insert synchronization operations that sequentialize operations. Figure 3.5, on the other hand, presents a function suitable for parallel execution. The naïve implementation of the Fibonacci sequence has two independent recursive function calls whenever `x > 2`. As these calls have no dependence among themselves, when the code of a `fib` call takes the else path, it will allocate a call record and issue a queue entry for the first call (with `x - 1`). As this is a non-blocking operation, it will proceed to make the second call (with `x - 2`). Only when both calls are enqueued, a synchronization operation is issued, as the compiler realizes that the return values are now necessary for the function to proceed. This kind of recursive parallelism can provide numerous parallel function calls for ACQuA to process, taking great advantage of an increased number of processing units, as will be seen in chapter 5.

3.3 Optimizations for map applications

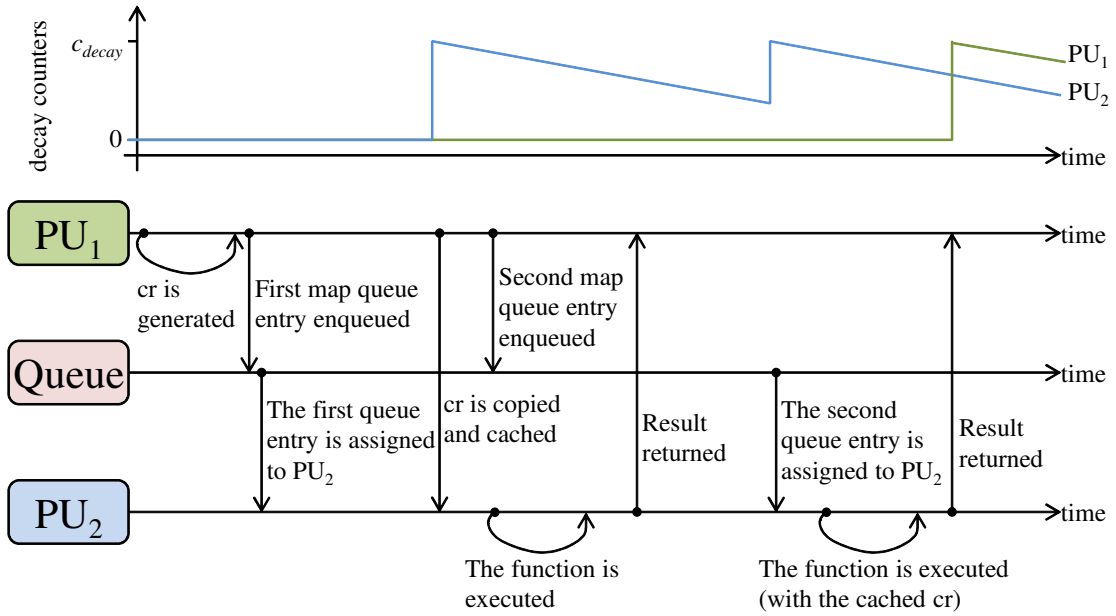
Every functional programming language comes equipped with a high order function called map. A *map* consists in the application of a function to each element of a list, returning a new list with the result of each application: $map\ f\ \left[v_1, v_2, ...v_n\right] = \left[f\left(v_1\right), f\left(v_2\right), ...f\left(v_n\right)\right]$. As such, the map function provides substantial parallelism for ACQuA's execution model, since these multiple calls are independent from each other. To take full advantage of this potential parallelism, however, some optimizations had to be devised and incorporated into ACQuA.

As described in Section 3.2, whenever a function is called, a call record and a queue entry have to be created. For a map application, this means one call record and one queue entry for each element of the list. Moreover, once the queue entry is assigned to a processing unit, the call record has to be transmitted through the interconnection, adding latency and increasing traffic. The optimizations proposed in this Section aim to mitigate these two possible bottlenecks for map applications: the costs of creating and fetching through the interconnection the new call records created for each list element; and the costs of creating a new queue entry for each list element.

Reuse of call records. The first optimization is based on the fact that, in the call records of function applications stemming from a map, only the function parameter changes. The other fields in the call record are identical. Therefore, we propose to reuse the same call record for all function calls, with the function parameter being hold on the queue entry instead of in the call record bindings. The *mapParam* field in the queue entry is used for this purpose, with the *isMap* bit indicating to the receiving processing unit that the *mapParam* field should be considered. Moreover, when *isMap* is set, the processing unit fetches the call record from the remote memory, as usual, but it also caches it for later reuse. A single cache entry is used in each processing unit. When receiving another queue entry with *isMap* on, the processing unit checks if the call record to be copied is at the same address of the call record in the cache, and if the call record is still valid. If it is at the same address and valid, the call record copy is skipped, using the cached call record. If the call record is invalid or resides on a different address, it is copied through the interconnection and cached.

Cache validity is required to avoid a potential incorrect execution when a memory address is reused to store a different call record. In this case, a processing unit may

Figure 3.6: Call record cache functionality



mistakenly use its old cached record. We use cache decay to avoid this problem. An unused cached call record only remains valid for a predetermined number *c_decay* of cycles. When a call record is cached, a counter is set to *c_decay* and starts to decrement every cycle. Each time a call record is reused, the counter is reset. If the counter reaches zero, the cache entry is deemed invalid. To avoid false positives, the processing unit which executed the map cannot allocate other call records on the same memory address until *c_decay* cycles after it receives the last returned value from all mapped functions. This approach ensures that any cached copy of that address will have already decayed. False negatives may happen, and in this case, the same call record is copied through the interconnection again. By setting a sufficiently large *c_decay*, we can make these repeated copies rare enough so that they are not a bottleneck to overall performance. Figure 3.6 shows the call record cache functionality during the execution of a map on a list of two elements.

Splitting calls among processing units. The second optimization aims at increasing the throughput with which the function calls are set up by a map. A single processing unit creating each queue entry can become a bottleneck, even if the call record is being cached and reused, especially if the function being mapped has low computational cost. In these scenarios, it might be possible that a single processing unit (or just a few of them) is processing the function calls at the same rate with which they are created, while the other processing units remain idle, reducing the attainable acceleration.

Thus, this optimization consists in dividing the task of creating queue entries among multiple processing units, increasing the throughput and allowing more units to process the map calls in parallel. When a map on a sufficiently long list is found, the list is sliced, and multiple parallel maps are initiated on the sublists. The results are then concatenated into a single list.

The optimization is only used when the list size is greater than eight times the number of processing units on the ACQuA accelerator being used. Also, it is only used when ACQuA has at least eight processing units. For shorter lists or fewer processing units, the costs of slicing and concatenating the lists are not amortized, providing no gains or even performance reductions. These thresholds are heuristically based on our test cases and can be adjusted for different functions, as they are embedded in the code.

4 ACQUA'S INTERMEDIATE LANGUAGE

The compiler developed for ACQuA accepts as input programs written in a small core for a strict and pure functional language with integers, (high order) functions, and lists. Figure 4.1 shows the language's abstract syntax. Examples of code given in the dissertation are written using usual syntactic sugar. Some examples of programs written on the language without any syntactic sugar can be found on Chapter B.

Figure 4.1: Abstract syntax for the source language.

$$
\begin{aligned}
e ::= {} & n \\
\mid {} & x \\
\mid {} & e_1 \ op \ e_2 \\
\mid {} & \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \\
\mid {} & \texttt{fn } x \Rightarrow e \\
\mid {} & e_1 \ e_2 \\
\mid {} & \texttt{let } x = e_1 \texttt{ in } e_2 \\
\mid {} & \texttt{let rec } x = \texttt{fn } x \Rightarrow e \texttt{ in } e' \\
\mid {} & e_1 :: e_2 \\
\mid {} & \texttt{nil} \\
\mid {} & \texttt{head } e \\
\mid {} & \texttt{tail } e \\
\mid {} & \texttt{map } e_1 \ e_2 \\
\mid {} & \texttt{last } e \\
\mid {} & \texttt{length } e \\
\mid {} & \texttt{concat } e_1 \ e_2 \\
\mid {} & \texttt{slice } e_1 \ e_2 \ e_3
\end{aligned}
$$

All processing units apart from the reserved processing unit execute the same instruction set, that is represented by the ACQuA_{IR}. There are no architecture specific variants on the compilation, so the code is not compiled for a specific number of proces-

sors, memory size or interconnection topology. This property allows moving to more or less powerful versions of the architecture without the need of recompilation, much like superscalar processors.

The compiler output is a program written in ACQuA's intermediate representation (ACQuA$_{IR}$). ACQuA$_{IR}$ is similar to a regular Instruction Set Architecture (ISA), but featuring commands for list processing and commands that are specific to ACQuA's execution model, such as commands for creating call records, and for updating/recovering call record components. ACQuA$_{IR}$ can be directly executed by specialized processing units or translated into any standard ISA. In this work, the former is considered.

Table 4.1: ACQuA$_{IR}$ Basic commands and terminators summary.

Command	Cycles	Description
x = `call` cr_{id}	1	Creates a queue entry which references the call record cr_{id}. The respective function's returned value will be stored on x.
`wait`	1	Holds this function execution until all functions called from it are finished.
x = `GetNPUS`	1	Assigns to x the number of processing units on ACQuA$_{IR}$. It does not count the reserved processing unit.
x = `innercopy` x'	x' size	Makes a copy of call record or list x' residing in this processing unit and assign it to x.
x = `outercopy` x'	varies	Makes a copy of call record or list x' residing in other processing unit and assigns it to x. This command requires data transfer through the interconnection, and might take an arbitrary number of cycles to finish depending on the architecture state.
`free` x	1	Free the memory referenced by x. x might reference a list or call record.
`return` x	1	Finishes the current function execution and returns x to the processing unit which inserted the queue entry that originated the call.

An ACQuA$_{IR}$ program is defined as a sequence of basic blocks. Each basic block starts with a label for its entry point, followed by a sequence of commands ending with a basic block terminating command (either a conditional/unconditional jump to another basic block, or a return command). As is common for machine-level code, some basic blocks are entry points for functions and can be referenced by function calls, while others are only accessible from within the same function.

In the context where ACQuA is used as an accelerator we assume the input for the compiler is a sequence of function definitions. The compiler will produce a sequence of

Table 4.2: ACQuA$_{IR}$ call record related commands.

Command	Cycles	Description
cr_{id} = `newCR` n	1	Creates a new call record of size n which can be referenced by cr_{id}.
`setCRentry` cr_{id} l	1	Sets the *fn_addr* component from the call record cr_{id} to label l.
`setCRmis` cr_{id} x	1	Sets the *n_missing* component from the call record cr_{id} to x.
`setCRcnt` cr_{id} x	1	Sets the *n_available* component from the call record cr_{id} to x.
`setCRpar` cr_{id} n x	1	Sets the nth identifier from *bindings* component present on the call record referenced by cr_{id} to x.
x = `getCRmis` cr_{id}	1	Assigns the *n_missing* value from the call record referenced by cr_{id} to x.
x = `getCRcnt` cr_{id}	1	Assigns the *n_available* value from the call record referenced by cr_{id} to x.
x = `getCRpar` cr_{id} n	1	Assigns the nth value from *bindings* component present on the call record referenced by cr_{id} to x.

basic blocks for each one of the functions $f_1 \ldots f_n$. When one of those functions, say f_i, is called from a program executing in the host processor, the cycle described in Section 3.2 initiates with the execution of ACQuA$_{IR}$ commands in the first basic block produced for f_i.

Commands and terminators are presented in three tables. Table 4.1 shows the basic commands and terminators. Table 4.2 shows call record-related commands. Table 4.3 shows list-related commands. These tables show the mnemonic representation and a brief description of each ACQuA$_{IR}$ command. They also include the cycle cost for each command, which is used during simulation, as will be detailed in Chapter 5. The compilation function that takes abstract syntax trees of expressions of the functional language and produces a sequence of ACQuA$_{IR}$ basic blocks is defined in the Apendix A.

The intermediate representation includes two data types: call records and lists. A call record is a structure that is used on function calls and was extensively discussed in Chapter 3. Lists are supported because they are frequently used in functional programs and also because the map function, which produces multiple independent functions applications, is of particular interest in this work. Lists have received native architectural support, which is reflected on ACQuA$_{IR}$ as well. In what follows, we explain some of the commands summarized in the tables.

Call records are created on ACQuA$_{IR}$ by using the command cr_{id} `= newCR n`,

where `n` is an integer that defines the memory size of the call record. The call record size varies depending on the number of parameters and non-local variables presented in the function body which the call record represents. This number is defined during compilation. The memory contents of a just-allocated call record memory are undefined. There are also a series of *set*/*get* commands to modify and to obtain the values of call record components which are shown on table 4.2.

The command `x = call` cr_{id} calls a function with call record pointed by cr_{id}. This instruction does not block execution and the identifier x remains with an undefined value until the function called returns and a value is assigned to it. This instruction also creates a new call queue entry pointing to cr_{id}. This call does not block execution, and x will have an undefined value until the function described by cr_{id} is finished and the returning value is sent through the interconnection. To determine how many return values a given function is currently waiting, the number of times that this command is executed needs to be tracked. For this purpose, the *callCount* component of the caller's *execution record* is incremented when calls are made and decremented when return values are received.

The only way to wait for the returning value after a call command is executed is by using the command `wait`. The command does not receive any parameters, and it does not return any value. This command's behavior is linked to the *callCount* component from the *execution record*. If the `wait` command is executed and *callCount* is zero, nothing happens. If *callCount* is greater than zero, the current execution stops, its context is saved in *exeCtxt*, and the processing unit is free to receive other queue entry from the scheduler.

The `return x` command is also specific to this architecture. Returning a value marks the end of a function execution. Thus the resulting value should be sent to the processing unit which created the queue entry. The processing unit also becomes available to be assigned other queue entries by the scheduler. The *execution record* component *ret_addr* contains the memory address in which the resulting value should be placed. The value of *ret_addr* is set when the queue entry is assigned to the processing unit. Executing the `return` command causes a message to be sent through the interconnection with the value of `x` to the processing unit which has the memory address *ret_addr*.

Upon receiving a message with the resulting value of a function execution triggered by `return`, the *callCount* of the function in which the resulting value is placed is decremented. If this value is zero, the receiving function is not expecting any other returning values and can resume execution. The function resumes its execution immediately

if the processing unit is idle. Otherwise, it is placed on an internal list of functions that can resume. This list is necessary because more than one waiting function might become able to resume execution while another function executes. The list of functions ready to be resumed has higher priority than new queue entries so a processing unit that becomes idle first checks its internal list and only if the list is empty does it inform the scheduler it is idle.

There are two optimizations related to the map command: the call record cache and the division of a map on a long list to multiple maps on shorter lists. Both optimizations are described in Section 3.3. The call record cache is handled directly by the processing units, and no extra ACQuA$_{IR}$ code is generated for it. However, code for map splitting is inserted by the compiler. This optimization is only applied when the list size is greater than eight times the number of processing units on the ACQuA accelerator being used, as discussed in Section 3.3.

To discover the number of processing units, the instruction n = `GetNPUS` is used. If there are more than eight elements on the list for each processing unit, the optimized code for map is executed. This optimized code starts by splitting a list $list_{id}'$ in smaller lists, using the $list_{id}$ = `slice` $list_{id}'$ n_1 n_2 command. The command $list_{id}$ = `smap` cr_{id} $list_{id}'$ n is then used to map the function represented by call record cr_{id} to each of the smaller lists produced. The command `smap` is similar to the `map` command, but allows its list argument to be on a different processing unit. After all partial resulting lists are done, they are concatenated to construct the resulting map of the original list.

Table 4.3: ACQuA$_{IR}$ list related commands.

Command	Cycles	Description
$list_{id}$ = `newList` n	1	Creates a list of size n and assigns its reference to $list_{id}$.
`setList` $list_{id}$ nx	1	Assigns x value to the nth element of the list referenced by $list_{id}$.
x = `getList` $list_{id}$ n	1	Assigns the value of the list referenced by $list_{id}$ nth element to x.
x = `length` $list_{id}$	1	Assigns the length of the list referenced by $list_{id}$ to x.
x = `last` $list_{id}$	1	Assigns the value of the last element from the list referenced by $list_{id}$ to x.
x = `head` $list_{id}$	1	Assigns the value of the first element from the list referenced by $list_{id}$ to x.
$list_{id}$ = `tail` $list_{id}'$	$list_{id}$' size	Creates a copy of the list referenced by $list_{id}$' without its first element. The copy can be referenced by $list_{id}$.
$list_{id}$ = `concat` $list_{id}'$ $list_{id}''$	$list_{id}$' size + $list_{id}$" size	Creates a new list referenced by $list_{id}$ with all elements from the list $list_{id}$' followed by the elements from list $list_{id}$".
$list_{id}$ = `map` cr_{id} $list_{id}'$	*bindings* size from cr_{id} + 1	Maps the function defined by cr_{id} to the list $list_{id}$'. The list with results is referenced by $list_{id}$.
$list_{id}$ = `smap` cr_{id} $list_{id}'$ n	*bindings* size from cr_{id} + 1	Maps the function defined by cr_{id} to the list $list_{id}$' that is at a processing unit. n is the list size of $list_{id}$'. The returning list can be referenced by $list_{id}$.
$list_{id}$ = `slice` $list_{id}'$ n_1 n_2	$list_{id}$ size	Creates a new list with elements from $list_{id}$', starting on $list_{id}$' n_1th element to its n_2th element. The new list is referenced by $list_{id}$.

5 EXPERIMENTAL RESULTS

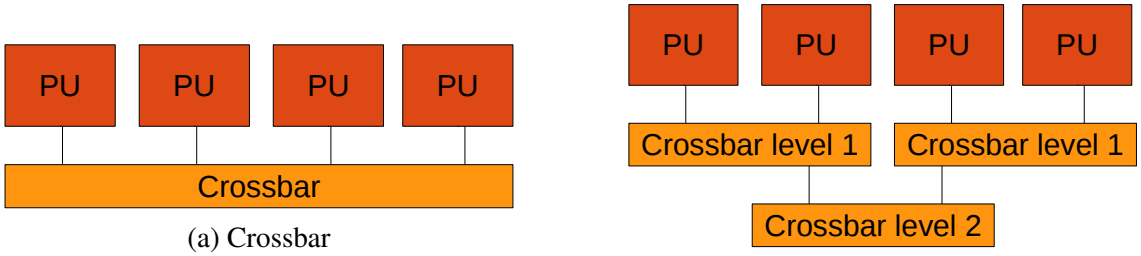
A simulator for ACQuA, written in Haskell, was developed to obtain the results presented in this section. The simulator itself, the resulting data, and the scripts used to conduct the experiments are all publicly available (TANUS,). The simulator operates at the instruction level, directly executing ACQuA_{IR} operations at each processing unit.

We consider one cycle per ACQuA_{IR} instruction except when handling lists. For instance, the `tail` instruction, described in table 4.3, takes n cycles where n is the size of the list. The simulator estimation of cycles for each instruction can be found on tables 4.1, 4.2 and 4.3. As the goal of this work is to evaluate the architecture's scalability and possible performance bottlenecks, instead of exactly counting execution cycles, the simulator simplifies details like processing unit pipelines and considers a fixed cycle cost per instruction. Optimizations such as instruction-level parallelism are possible mechanisms to improve the architecture's performance, but are outside the scope of this work. All processing units, the queue and the interconnection work with one synchronous clock.

The benchmarks are divided into two categories: list-based algorithms and scalar algorithms. We divided the benchmarks on those two categories because examples of the same category tend to have the same properties, simplifying the evaluation and explanation of the data gathered. Three scalar algorithms and four list algorithms with different properties are used. The scalar examples are naïve implementations of Fibonacci, factorial, and Newton-Raphson. The list-based algorithms are the longest common substring (LCS), dot product, mapping of the greatest common divisor (GCD), and quicksort. Those algorithms were picked to show how different program structures behave in the ACQuA architecture, aiming to identify properties of programs that make them good candidates for acceleration with ACQuA. As for the input to our algorithms, we calculate the 15th Fibonacci number, sort a list of 200 random numbers, find the longest common substring of 4-character string on a 100-character string, calculate the dot product of 2 vectors of 2000 numbers, find the greatest common divisor of a list of 200 pairs of numbers, and execute a total of 100 Newton-Raphson iterations. Strings and pairs are implemented using ACQuA's native lists. The inputs for each example were selected to be big enough in order to evaluate the architecture with 16 processing units, since increasing the example's input size will generate similar results. These benchmarks were compiled with both optimizations discussed on Section 3.3.

We explored two different interconnections: crossbar and hierarchical crossbar.

Figure 5.1: Difference between simulated crossbar and hierarchical crossbar interconections.



(a) Crossbar

(b) Hierarchical crossbar

The crossbar interconnection connects every processing unit with each other in a single hop 5.1 (a). The hierarchical crossbar is a slower interconnection model that escalates better with the number of processing units, in terms of area. Figure 5.1 (b) shows the interconnection architecture of a hierarchical crossbar, which is structured as a tree of degree-two crossbars. This structure has a linear area increase with the amount of processing units, instead of the quadratic increase observed for an n-degree crossbar. During the hierarchical crossbar simulations, we pessimistically consider that all processing units are as far apart as possible. Thus, messages have an additional latency of $2 * log_2 n$ cycles (to go up and down the entire tree), where n is the number of processing units.

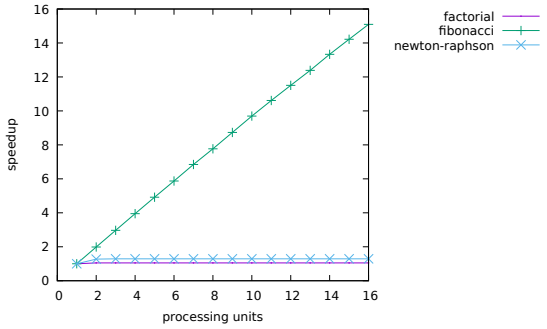
We start by evaluating the speedup and memory usage when increasing the number of processing units, using the crossbar interconnection. Then we evaluate the speedup for the hierarchical crossbar. We do not present the memory usage for hierarchical crossbar because both results and analysis are similar. Afterwards, we show the effectiveness of list and map the optimizations discussed on Section 3.3 by using two synthetic examples.

5.1 Speedup

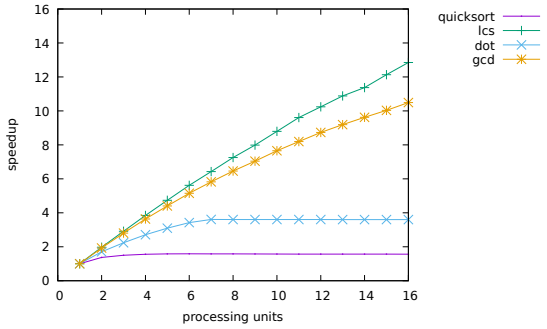
The analysis of speedup with a crossbar interconnection considers scalar and list-based benchmarks in separate. We start with the simulation results for the scalar algorithms.

Scalar benchmarks. Figure 5.2a shows the speedup of each scalar benchmark when the number of processing units is increased. We can see a significant speedup curve with Fibonacci. This speedup is expected because Fibonacci with input 15 is massively parallel and has two independent recursive function calls that easily occupy all the processing units. We can see an almost ideal linear speedup by increasing the number of process-

Figure 5.2: Speedup



(a) Speedup on scalar algorithms.



(b) Speedup on list-based algorithms.

ing units. The experiment with the Fibonacci function show us that ACQuA is capable of exploring the intrinsic parallelism of functions to a great extent without any specific language-related support.

For the factorial example, we expected no speedup at all, but there is an almost negligible speedup when adding a second processing unit in Figure 5.2a. However, adding more processing units does not make any difference. This happens because some small parallelism from the architecture is explored: a function call, `fat(5)` for instance, causes a new queue entry to be placed in the queue. The processing unit that was assigned to `fat(5)` carries on the execution until it finds a `wait` instruction. Only then we have a free processing unit to receive the new function call (`fat(4)`). With two processing units, these two function calls can alternate the execution, having some instructions of advantage. Because there is no function calls on factorial that can be executed in parallel, at most two processing units are being used on any given cycle.

The Newton-Raphson method divides a function by its derivative. The calculation of the function and its derivative can be done in parallel, and this is the only parallelism provided by this benchmark. For the next iteration, both function calls have to be finished. So naturally, as we can observe in Figure 5.2a, we have a small speedup when adding the second processor, but adding more processors shows no advantage at all, providing only slightly more speedup than the Factorial example.

List-based benchmarks. Figure 5.2b shows the speedup of each algorithm that operates with lists when the number of processing units is increased. LCS shows significant speedup, similar to the speedup curve of Fibonacci given in Figure 5.2a. We can identify a pattern that is present both in Fibonacci and in the LCS programs: the existence of independent recursive function calls. Algorithms that share this characteristic are the ideal

candidates for acceleration with ACQuA.

Observe that the speed up for Fibonacci is almost linear and it is higher than the speed up obtained for LCS. That happens because LCS has to perform longer sequential computations with the results returned after the (recursive) function calls. These computations involve list operations which take more cycles than the arithmetic operations of Fibonacci, reducing the overall speedup.

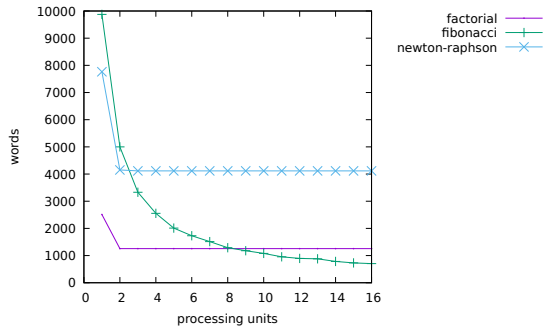
The dot product exhibits some speedup as we increase the number of processing units but quickly reaches a limit. Since the code of the function being applied is very small (it performs only a multiplication), the messages sent through the interconnection and the overhead of creating a new call record for just one multiplication operation becomes the bottleneck. To confirm this hypothesis, we can examine the benchmark that maps GCD (greatest common divisor) to a list. In this benchmark, a function that calculates the GCD is mapped to a list of tuples to define each tuple's greatest common divisor. The parallelism present is similar to the dot product, but since the greatest common divisor computation takes longer than a simple multiplication, a better speedup is observed. This shows that while ACQuA can achieve a small grain of parallelization in comparison with traditional parallelization techniques, some tiny parallelization is not successfully explored by ACQuA because of the overhead in managing the call records and scheduling the functions to be executed.

Quicksort also exhibits independent recursive function calls (like Fibonacci and LCS), and in Figure 5.2b we observe a small speedup by adding up to four processing units. The main difference of quicksort and LCS is that quicksort contains significant sequential parts when splitting the numbers of a list as lesser or greater than the pivot, reducing the achieved speedup. This limitation can be understood through Amdahl's law (AMDAHL, 1967), which states the sequential parts of any program impose limits on the maximum speedup. The first quicksort call, which operates on the entire list, has a much longer running time than its subsequent calls. It takes, in fact, roughly half of the entire execution time, assuming nearly-ideal pivots. Thus, the sequential nature of this first call greatly limits total speedup.

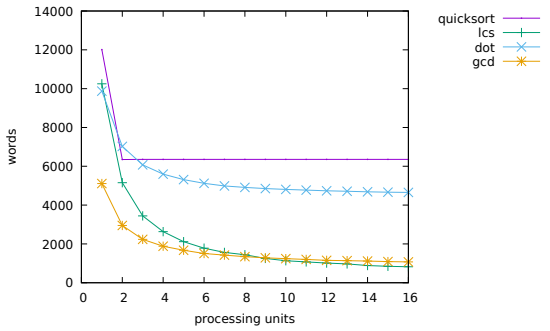
5.2 Memory usage

As we did for the analysis of speedup, we also consider memory usage first for scalar algorithms and then for list-based algorithms. In both cases, we assume a crossbar

Figure 5.3: Memory usage



(a) Memory usage in a processing unit for scalar algorithms.



(b) Memory usage in a processing unit for list-based algorithms.

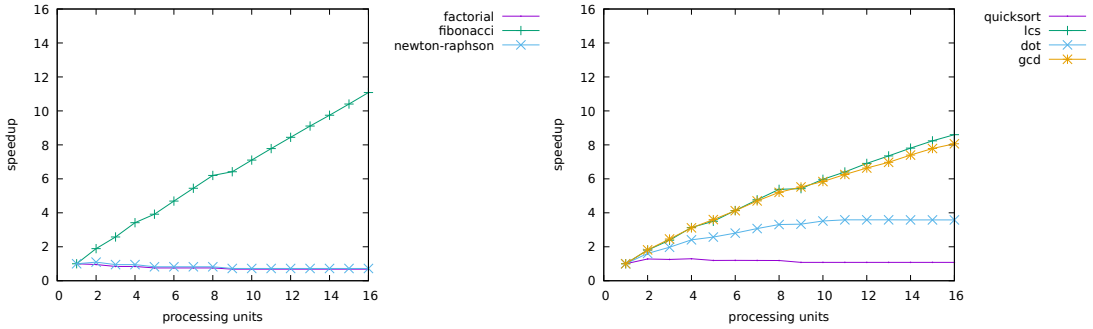
interconnection.

Scalar benchmarks. Figure 5.3a shows the memory usage of any single processing unit during the execution of each scalar benchmark when we increase the number of processing units. The memory measured is the area required, in memory words, to allocate call records and lists, which are the sources for memory dynamically allocated.

The goal of this analysis is to determine whether memory occupation is properly spread across multiple processing units when we increase the number of processing units. In other words, this analysis is important to investigate how the required memory per processing unit scales as more units are added. Requiring more memory per unit as more units are added would constitute a serious scalability problem.

The Fibonacci function shows a decrease in the memory usage in a single processing unit as more processing units are added, as expected. Memory usage per processor also decreases when a second processing unit is added to the factorial and Newton-Raphson examples. However, for these two benchmarks, the used memory does not decrease by adding more processing units because those extra processing units are not used during the execution. Part of this excessive memory usage is due to the simple scheduling algorithm used in the simulator. It takes the first processing unit available to execute a function, and on sequential code, it ends up always taking the same one or two processing units. This decision does not help to decrease the memory usage per processor since all other processing units are kept idle and with empty memories through the entire execution. Other scheduling algorithms are yet to be studied and can address this problem by spreading functions to different processors. This would decrease the memory usage per unit by occupying memory of different processing units.

Figure 5.4: Speedup using hierarchical crossbar



(a) Speedup with hierarchical crossbar for scalar algorithms.

(b) Speedup with hierarchical crossbar for list-based algorithms.

List-based benchmarks. Figure 5.3b shows the memory usage of any single processing unit during the execution of each list-based benchmark when we increase the number of processing units. The memory measured is the area required to allocate call records and lists.

The memory occupation for list programs follow the same trends observed for the scalar examples. As more processing units are occupied, memory occupation spreads throughout the architecture. Thus, the programs with higher speedup, LCS, and GCD, require progressively less memory per unit as more units are added. As observed for scalar benchmarks, modified scheduling approaches can be investigated to force the spread of memory occupation when required.

5.3 Speedup for hierarchical crossbar interconnection

The hierarchical crossbar is a slower interconnection model that escalates better with the number of processing units in terms of area. We evaluated only the speedup of the benchmarks for this model, to discover if this cheaper interconnection is enough to provide a performance gain. Figure 5.4a shows the speedup of scalar benchmarks when the interconnection is a hierarchical crossbar. On the Fibonacci example, we can see that adding more processing units still is quite effective, but the speedup line is not as straight as with a crossbar as interconnection. This is so because, in this scenario, adding more processing units means more time required to exchange messages. For very sequential algorithms (factorial and Newton-Raphson), adding more processing units slightly increases the execution time, due to the longer interconnection latency. Note, however, that the results on Figure 5.4a are pessimistic considering that PUs are as far apart from each

other as possible, not taking into account that neighbouring processing units have shorter interconnection latencies.

On Figure 5.4b we can see the speedups for list-based algorithms when using a hierarchical crossbar as interconnection. While this interconnection yields inferior performance, there is still a reasonable speedup for LCS and dot product, and especially for LCS. The quicksort is more severely penalized, similarly to Newton-Raphson and factorial. It is interesting that the dot product maintains its speedup even when adding more processing units. This is so because the current bottleneck of the algorithm is in sending data from one single processing unit to others. Therefore, performance is currently limited by the interconnection bandwidth and not latency, causing dot to yield similar results for both evaluated interconnection topologies. The mapping of GCD, on the other hand, shows a smaller speedup, because since it uses more cycles to execute the operations, the bandwidth is not fully used.

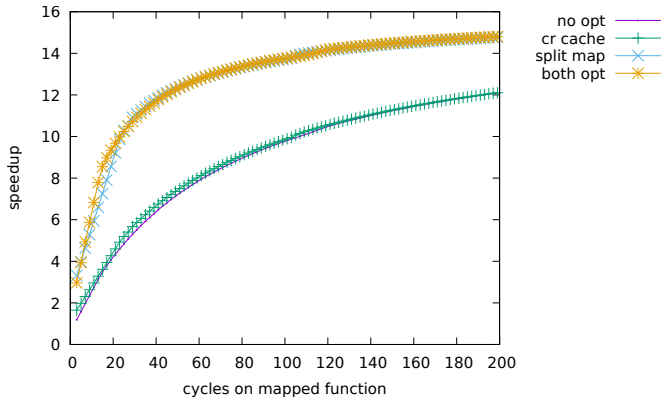
These results show that the interconnection topology plays a significant role on the attainable speedup, but that speedup can be achieved even with heavy interconnection penalties, provided the application presents sufficient parallelism.

5.4 Optimizations evaluation

We now discuss the impact of the two optimizations presented in Section 3.3: caching call records of map calls, and splitting map calls of big lists in multiple processing units. In this section, in order to have a finner control on the number of cycles, synthetic benchmarks are used to evaluate the effectiveness of each optimization, individually and combined, under different circumstances. The four possible combinations are considered: no optimizations; only splitting map among processing units (split map); only call record cache (cr cache); and both optimizations applied.

Figure 5.5 shows the speedup obtained using 16 processing units and varying the number of cycles of the function mapped to the elements of a list of size 500. The variation in the umber of cyles is obtained by varying the number of 1's added in the function mapped to the list (see code in figure 5.5b). First, as expected, the speedup increases along with the number of cycles in the mapped function and approaches the theoretical maximum value of 16. As was observed with dot and GCD, longer functions allow the amortizing of the costs, such as the allocation of call records and the issuing of queue entries. Maximum performance is always attained when both optimizations are enabled.

Figure 5.5: Synthetic example to evaluate the effectiveness of the split map optimization

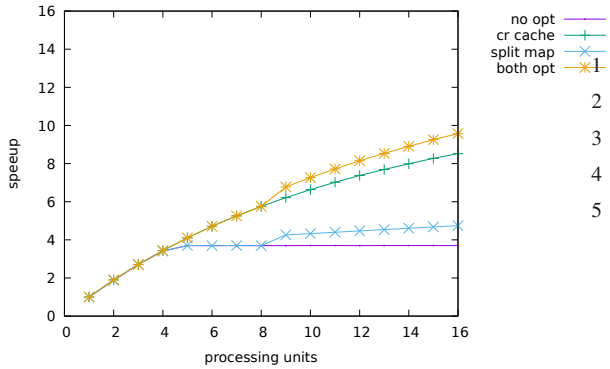


(a) Speedup with 16 PUs with optimizations and varying the number of cycles of the function mapped to a list.

```
map   fn x ⇒ x+1+1...+1   [n1...n500]
```

(b) The code above is executed multiple times varying the number of 1s added.

Figure 5.6: Synthetic example to highlight the effectiveness of the call record cache optimization



(a) Speedup for different number of processing units.

```
fun f x1 x2 x3 x4 x5 x6 ⇒
                x1+x2+x3+x4+x5+x6

let g = (((((f 1) 2) 3) 4) 5)
map g n1...n500
```

(b) The synthetic function f with a call record with 5 *bindings* mapped to a list with 500 elements.

Only using split map, however, provides very similar results in most cases. For very fast functions (around 20 cycles), the call record transfer time is relevant and, therefore, the cache of call records provides some improvements.

Figure 5.6 shows a different scenario. In Figure 5.6a, the code shown in Figure 5.6b is used. The call records created have five bindings. Then, the different values for the map are taken from the list, and passed through the *mapParam* field of each queue entry. The larger the call record, the more expensive to transfer it from one processing unit to another, increasing the relevance of the cache.

From Figures 5.5a and 5.6a, it can be seen that both optimizations combined always provide the best results and that their individual relevance depends on the properties

of the function being executed, and there are cases that one works better than the another.

6 CONCLUSION AND FUTURE WORK

We have presented the ACQuA accelerator architecture, capable of exploring intrinsic parallelism of functional programs automatically, without the need of language support. We have identified a class of programs that the accelerator excels with: programs that make multiple independent function calls with non-negligible processing done by each function. We have shown in the experiments that the accelerator can achieve outstanding results for programs that share these properties, such as the longest common substring and Fibonacci.

There is much to be studied about the ACQuA accelerator yet. The best interconnection between the processing units is yet to be defined. A crossbar has lower message passing cost, but the required area grows exponentially with the number of processing units. The performance penalty of slower interconnection models might give a faster overall accelerator because the extra space can be used to add more processing units. Studying this trade-off can be very rewarding to the architecture.

We have not defined the best algorithm to be used by the scheduler. This algorithm has a major impact on performance with interconnections other than the crossbar. In a hierarchical crossbar, a message might take a different number of cycles between two different processing units. Minimizing the distance between two processing units that will exchange a significant amount of data can improve the accelerator performance. The scheduler algorithm should be studied along with different interconnection models.

The compiler used to generate the ACQuA_{IR} on our simulator is quite basic. The only optimization done is the removal of unnecessary variable assignments resulting from the compilation. Not even known optimizations are applied to the resulting ACQuA_{IR}. Since ACQuA_{IR} has novel instructions not present in other popular intermediate languages, new optimizations might have to be developed and existing optimizations might have to be adapted to work with ACQuA_{IR} instruction set.

We have shown with the dot product example that parallelism on tiny processing does not achieve good results. The multiplications can be done in parallel, but the memory bandwidth does not allow the architecture to achieve great speedups. If the architecture could explore even the tiniest processing loads, it would increase significantly the class of programs worth accelerating, and it would avoid the necessity of complex compiler analysis to decide when to parallelize. It might be possible to solve this architectural problem using 3D stacked memory.

Currently, the evaluation of the ACQuA accelerator is done using a simulator. While the simulator is a good tool for a first study of viability and scalability, we cannot compare the performance of the accelerator with existing processors. It would be interesting to implement the architecture physically and see how ACQuA accelerator compares to mainstream processors.

It would be interesting to compile a feature-rich language to ACQuA_{IR}. It is possible to find more parallelism opportunities in the language features, other than function applications. Also, compiling existing software from a feature-rich language can show the speedup on real-world programs, and define areas which are more important to optimize.

REFERENCES

AMDAHL, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In: **Proceedings of the April 18-20, 1967, Spring Joint Computer Conference**. New York, NY, USA: ACM, 1967. (AFIPS '67 (Spring)), p. 483–485.

ASADOLLAH, S. A. et al. A study of concurrency bugs in an open source software. In: SPRINGER. **IFIP International Conference on Open Source Systems**. [S.l.], 2016. p. 16–31.

AUBREY-JONES, T.; FISCHER, B. Synthesizing MPI implementations from functional data-parallel programs. **International Journal of Parallel Programming**, v. 44, n. 3, p. 552–573, 2016. ISSN 1573-7640.

BACKUS, J. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. **Commun. ACM**, ACM, New York, NY, USA, v. 21, n. 8, p. 613–641, ago. 1978. ISSN 0001-0782.

BOEIJINK, A.; HÖLZENSPIES, P. K. F.; KUPER, J. Introducing the pilgrim: A processor for executing lazy functional languages. In: **IFL**. [S.l.: s.n.], 2010.

CHEN, M. K.; OLUKOTUN, K. Exploiting method-level parallelism in single-threaded java programs. In: **Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)**. [S.l.: s.n.], 1998. p. 176–184. ISSN 1089-795X.

CHEN, M. K.; OLUKOTUN, K. The jrpm system for dynamically parallelizing java programs. In: **30th Annual International Symposium on Computer Architecture, 2003. Proceedings.** [S.l.: s.n.], 2003. p. 434–445. ISSN 1063-6897.

DAVE, C. et al. Cetus: A source-to-source compiler infrastructure for multicores. **Computer**, Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, 17 th Fl New York NY 10016-5997 USA, v. 42, n. 12, p. 36–42, 2009.

HAMMOND, K. Why parallel functional programming matters: Panel statement. In: **Reliable Software Technologies-Ada-Europe 2011**. [S.l.]: Springer, 2011. p. 201–205.

HEWITT, C.; BISHOP, P.; STEIGER, R. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In: STANFORD RESEARCH INSTITUTE. **Advance Papers of the Conference**. [S.l.], 1973. v. 3, p. 235.

ISO. **ISO/IEC 9899:2011 Information technology — Programming languages — C**. Geneva, Switzerland: International Organization for Standardization, 2011. 683 (est.) p.

JOHNSTON, W. M.; HANNA, J.; MILLAR, R. J. Advances in dataflow programming languages. **ACM Computing Surveys (CSUR)**, ACM, v. 36, n. 1, p. 1–34, 2004.

JONES, S. L. P. Parallel implementations of functional programming languages. **The Computer Journal**, v. 32, n. 2, p. 175–186, 1989.

JONES, S. P.; GORDON, A.; FINNE, S. Concurrent haskell. In: **POPL**. [S.l.: s.n.], 1996. v. 96, p. 295–308.

KIM, H.-S. et al. Icu-pfc: an automatic parallelizing compiler. In: **Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region**. [S.l.: s.n.], 2000. v. 1, p. 243–246 vol.1.

MARLOW, S.; NEWTON, R.; JONES, S. P. A monad for deterministic parallelism. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 46, n. 12, p. 71–82, set. 2011. ISSN 0362-1340.

NAYLOR, M.; RUNCIMAN, C. The reduceron reconfigured. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 45, n. 9, p. 75–86, set. 2010. ISSN 0362-1340.

RHO, E. H. et al. Compilation of a functional language for the multithreaded architecture: Davrid. In: **Parallel Processing, 1994. ICPP 1994 Volume 2. International Conference on**. [S.l.: s.n.], 1994. v. 2, p. 239–242.

RUMBAUGH, J. A data flow multiprocessor. **IEEE Transactions on Computers**, C-26, n. 2, p. 138–146, Feb 1977. ISSN 0018-9340.

SALVANESCHI, G.; GHEZZI, C.; PRADELLA, M. Contexterlang: Introducing context-oriented programming in the actor model. In: **Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development**. New York, NY, USA: ACM, 2012. (AOSD '12), p. 191–202. ISBN 978-1-4503-1092-5.

SCHEEVEL, M. Norma: a graph reduction processor. In: ACM. **Proceedings of the 1986 ACM conference on LISP and functional programming**. [S.l.], 1986. p. 212–219.

SHARP, J. A. **Data flow computing: theory and practice**. [S.l.]: Intellect Books, 1992.

STANDARD for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. **IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016)**, p. 1–3957, Sept 2016.

STOYE, W. R.; CLARKE, T. J.; NORMAN, A. C. Some practical methods for rapid combinator reduction. In: ACM. **Proceedings of the 1984 ACM Symposium on LISP and functional programming**. [S.l.], 1984. p. 159–166.

TANUS, F. **Acqua simulator source code**. <https://github.com/fotanus/acqua>. Accessed: 2016-08-3.

VEGDAHL, S. R. A survey of proposed architectures for the execution of functional languages. **IEEE Transactions on Computers**, C-33, n. 12, p. 1050–1071, Dec 1984. ISSN 0018-9340.

WASSERMAN, A. I.; GUTZ, S. The future of programming. **Communications of the ACM**, ACM, v. 25, n. 3, p. 196–206, 1982.

WATERLAND, A. et al. Asc: Automatically scalable computation. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 42, n. 1, p. 575–590, fev. 2014. ISSN 0163-5964.

APPENDIX A — COMPILING FROM L1 TO ACQUA$_{IR}$

In this appendix we introduce intuitively a compilation function that takes as arguments an functional language abstract syntax tree as described in Chapter 4 plus a table `ST` with information about identifiers and returns a list of ACQuA$_{IR}$ instructions. The function C is defined by induction on the structure of the abstract syntax tree. Resulting values for the evaluation of each subexpression of the tree are saved in a special variable named `ans`, which is used to The value produced by executing the ACQuA$_{IR}$ instruction resulting from $C\left(e, ST\right)$ is assigned with identifier `ans`.

The table ST starts empty, and during the compilation process it can be extended or consulted. ST tables names to code. To consult the ST, we use the notation $ST\left(name\right)$, and to expand it, we use $ST\left[name \rightarrow code\right]$. Some parts are defined intuitively as pseudo code. This pseudo code features the function `Issue` $\left(code\right)$, which issues the code contained in x, `Typeof` $\left(x\right)$, which defines the type of a variable x, and `FreeVars` $\left(e, ST\right)$,, which finds names that are not bound to a value on e taking in by testing against the variable names defined on ST.

$$
\begin{aligned}
\mathrm{C}(\mathrm{n}, \mathsf{ST}) &= \mathsf{ans} = n \\
\mathrm{C}(\mathrm{x}, \mathsf{ST}) &= \mathsf{ST}(\mathrm{x}) \\
\mathrm{C}(e_1 + e_2, \mathsf{ST}) &= \mathrm{C}(e_1, \mathsf{ST}) \\
&\quad aux_1 = \mathsf{ans} \\
&\quad \mathrm{C}(e_2, \mathsf{ST}) \\
&\quad aux_2 = \mathsf{ans} \\
&\quad \mathsf{ans} = aux_1 + aux_2 \\
\mathrm{C}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \mathsf{ST}) &= \mathrm{C}(e_1, \mathsf{ST}) \\
&\quad \texttt{if}\ \mathsf{ans}\ \texttt{goto}\ l_2 \\
&\quad l_4 : \\
&\qquad \mathrm{C}(e_3, \mathsf{ST}) \\
&\qquad \texttt{goto}\ l \\
&\quad l_2 : \\
&\qquad \mathrm{C}(e_2, \mathsf{ST}) \\
&\qquad \texttt{goto}\ l \\
&\quad l : \\
\mathrm{C}(\texttt{fn}\ x_1 \Rightarrow e_1, \mathsf{ST}) &= \texttt{goto continue} \\
&\quad l : \\
&\qquad endFnCode \leftarrow [\,] \\
&\qquad x_2 \ldots x_n \leftarrow \texttt{FreeVars}\,(e_1, ST) \\
&\qquad for\ x_i\ in\ x_1 \ldots x_n \\
&\qquad\quad if\ \texttt{Typeof}\,(x) \in \{\mathsf{CR}, \mathsf{list}\} \\
&\qquad\qquad x_i = \texttt{remoteCpy}\ x_i \\
&\qquad\qquad endFnCode \leftarrow endFnCode\ .\ \texttt{free}\ c_i \\
&\qquad C\,(e_1, \mathsf{ST}) \\
&\qquad \texttt{Issue}\,(endFnCode) \\
&\quad continue : \\
&\qquad ans = \texttt{newCR}\ n
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{C}(e_1\ e_2, \mathsf{ST}) = {} & \mathrm{C}(e_1, \mathsf{ST}) \\
& \mathit{crid} = \mathtt{ans} \\
& \mathrm{C}(e_2, \mathsf{ST}) \\
& \texttt{setCRpar}\ \mathit{crid}\ \mathtt{ans} \\
& \mathtt{missing} = \texttt{getCRmis}\ \mathit{crid} \\
& \mathtt{missing} = \mathtt{missing} - 1 \\
& \texttt{setCRmis}\ \mathit{crid}\ \mathtt{missing} \\
& \mathtt{available} = \texttt{getCRcnt}\ \mathit{crid} \\
& \mathtt{available} = \mathtt{available} + 1 \\
& \texttt{setCRcnt}\ \mathit{crid}\ \mathtt{available} \\
& \mathtt{ans} = \mathtt{missing}\ < 1 \\
& \texttt{if}\ \mathtt{ans}\ \texttt{goto}\ l \\
& \mathtt{ans} = \mathit{crid} \\
& \texttt{goto}\ l_2 \\
& l: \\
& \quad \mathtt{ans} = \texttt{call}\ \mathit{crid} \\
& \quad \texttt{goto}\ l_2 \\
& l_2: \\
\mathrm{C}(\texttt{let}\ x = n\ \texttt{in}\ e, \mathsf{ST}) = {} & \mathrm{C}(\mathrm{e}, \mathsf{ST}[\mathrm{x} \mapsto \mathtt{ans} = n])
\end{aligned}
$$

$$
\mathrm{C}(\texttt{let}\ x = \left[v_1, \ldots v_n\right]\ \texttt{in}\ e, \mathsf{ST}) = \mathrm{C}(e, \mathsf{ST}[\mathrm{x} \mapsto C\left(\left[v_1, \ldots v_n\right], \mathsf{ST}\right)])
$$

$$
\begin{aligned}
\mathrm{C}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, \mathsf{ST}) = {} & \mathrm{C}(e_1, \mathrm{ST}) \\
& x = \mathtt{ans} \\
& \mathrm{C}(e_2, \mathsf{ST})
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{C}(\texttt{let } f = \texttt{fn } x_1 \Rightarrow \ldots x_n \Rightarrow e_1 \texttt{ in } e_2, \mathsf{ST}) &= \texttt{goto } \mathit{continue} \\
&\quad\; l : \\
&\quad\; \mathit{endFnCode} \leftarrow [\,] \\
&\quad\; \mathsf{ST'} \leftarrow \mathsf{ST} \\
&\quad\; \mathit{for}\ x_i\ \mathit{in}\ x_1 \ldots x_n \\
&\qquad x_i = \texttt{getCRpar } \mathit{callRecord}\ i \\
&\qquad \mathit{if}\ \texttt{Typeof}\,(x_i) \in \{\mathsf{CR}, \mathsf{list}\} \\
&\qquad\quad \mathsf{ST'} \leftarrow \mathsf{ST'}\,[x_i \mapsto \mathit{ans} = \texttt{remoteCpy } x_i] \\
&\qquad\quad \mathit{endFnCode} \leftarrow \mathit{code}\ .\ \texttt{free } c_i \\
&\qquad \mathit{else} \\
&\qquad\quad \mathsf{ST'} \leftarrow \mathsf{ST'}\,[x_i \mapsto \mathit{ans} = x_i] \\
&\quad\; \mathrm{C}(e_1, \mathsf{ST'}) \\
&\quad\; \texttt{Issue}\,(\mathit{endFnCode}) \\
&\quad\; \mathit{continue} : \\
&\qquad \mathrm{C}(e_2, \mathsf{ST}[\mathrm{f} \mapsto \mathrm{ans} = \texttt{newCR n}]) \\
C\,([e_1, \ldots e_n], \mathsf{ST}) &= \mathit{lstid} = \texttt{newList } n \quad *n \geq 0 \\
&\quad\; C\,(e_1, \mathsf{ST}) \\
&\quad\; \texttt{setList } \mathit{lstid}\ 1\ \mathrm{ans} \\
&\qquad \ldots \\
&\quad\; C\,(e_n, \mathsf{ST}) \\
&\quad\; \texttt{setList } \mathit{lstid}\ n\ \mathrm{ans} \\
&\quad\; \mathrm{ans} = \mathit{lstid} \\
\mathrm{C}(\texttt{head } e, \mathsf{ST}) &= \mathrm{C}(e, \mathsf{ST}) \\
&\quad\; \mathrm{ans} = \texttt{head ans} \\
\mathrm{C}(\texttt{tail } e, \mathsf{ST}) &= \mathrm{C}(e, \mathsf{ST}) \\
&\quad\; \mathrm{ans} = \texttt{tail ans}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{C}(\texttt{last}\ e, \mathsf{ST}) &= \mathrm{C}(e, \mathsf{ST}) \\
&\quad \mathsf{ans} = \texttt{last}\ \mathsf{ans} \\
\mathrm{C}(\texttt{length}\ e, \mathsf{ST}) &= \mathrm{C}(e, \mathsf{ST}) \\
&\quad \mathsf{ans} = \texttt{length}\ \mathsf{ans} \\
\mathrm{C}(\texttt{concat}\ e_1\ e_2, \mathsf{ST}) &= \mathrm{C}(e_1, \mathsf{ST}) \\
&\quad aux_1 = \mathsf{ans} \\
&\quad \mathrm{C}(e_2, \mathsf{ST}) \\
&\quad aux_2 = \mathsf{ans} \\
&\quad \mathsf{ans} = \texttt{concat}\ aux_1\ aux_2 \\
\mathrm{C}(\texttt{slice}\ e_1\ e_2\ e_3, \mathsf{ST}) &= \mathrm{C}(e_1, \mathsf{ST}) \\
&\quad aux_1 = \mathsf{ans} \\
&\quad \mathrm{C}(e_2, \mathsf{ST}) \\
&\quad aux_2 = \mathsf{ans} \\
&\quad \mathrm{C}(e_3, \mathsf{ST}) \\
&\quad aux_3 = \mathsf{ans} \\
&\quad \mathsf{ans} = \texttt{slice}\ aux_1\ aux_2\ aux_3 \\
\mathrm{C}(\texttt{filter}\ e_1\ e_2, \mathsf{ST}) &= \mathrm{C}(e_1, \mathsf{ST}) \\
&\quad aux_1 = \mathsf{ans} \\
&\quad \mathrm{C}(e_2, \mathsf{ST}) \\
&\quad aux_2 = \mathsf{ans} \\
&\quad \mathsf{ans} = \texttt{filter}\ aux_1\ aux_2 \\
\mathrm{C}(\texttt{map}\ e_1\ e_2, \mathsf{ST}) &= \mathrm{C}(e_1, \mathsf{ST}) \quad *non-optimized\ version \\
&\quad crid = \mathsf{ans} \\
&\quad \mathrm{C}(e_2, \mathsf{ST}) \\
&\quad lstid = \mathsf{ans} \\
&\quad \mathsf{ans} = \texttt{map} \quad crid\ lstid
\end{aligned}
$$

C(map e_1 e_2, ST) = C(e_1, ST) $*optimized\ version$

$crid$ = ans

C(e_2, ST)

$lstid$ = ans

$lstsize$ = length $lstid$

pus = GetNPUS

ans = $pus * 8$

b_1 = $lstsize >$ ans

b_2 = $pus > 8$

ans = b_1 and b_2

if ans goto $lopt$

$ldummy$:

ans = map $crid\ lstid$

wait

goto $continue$

$lopt$:

$divisor$ = pus div 2

$slicesize$ = $lstsize$ div $divisor$

$nresults$ = $lstsize$ div $slicesize$

$partialResulLst$ = newListN $nresul$

setCRmis $crid$ 0 ts

start = 0

idx = 0

goto $loop$

```
loop :
  lst = slice   lstid start slicesize
  thisSliceSz = length  lst
  crid = newCR 3
  setCRentry crid splitMap
  setCRmis crid 0
  setCRcnt crid 3
  setCRpar crid 0 lstid
  setCRpar crid 1 lst
  setCRpar crid 2 thisSliceSz
  callL idx
continue :
```

APPENDIX B — EXAMPLE OF FUNCTIONAL PROGRAMS USED AS INPUT TO THE SIMULATOR

Fibonacci 10

```
letrec fibo =
  fn x =>
    if x < 2
      then 1
      else (fibo (x - 1)) + (fibo (x - 2))
in
  fibo 10
end
```

Factorial 10

```
letrec fat =
  fn x => if x > 1
          then x * (fat (x-1))
          else 1
in
  fat 10
end
```

Newton-Raphson starting on 40, 10 iterations

```
let fun = fn x => (x*x) + (2*x) - 1848 in
let dfun = fn x => 2*x + 2 in
let x0 = 40 in
let newtonrapson = fn x => x - ((fun x)/(dfun x)) in
letrec apply_n = fn func => fn arg =>
    fn n => if n <= 0
                then arg
                else (((apply_n func) (func arg)) (n-1))
        in
(((apply_n newtonrapson) x0) 10)
end
end
end
```

```
end
end
```

LCS, looking for [2,3,4] on [1,2,3,2,3,4]

```
letrec lcs = fn list1 => fn list2 =>
    if (length list1) = 0
    then 0
    else if (length list2) = 0
         then 0
         else let x = head list1 in
              let y = head list2 in
                if x = y
                then
                  let match = (lcs (tail list1)) (tail list2) in
                    1 + match
                  end
                else
                  let dropx = lcs((tail list1), list2) in
                  let dropy = lcs(list1, (tail list2)) in
                    if dropx > dropy
                    then dropx
                    else dropy
                  end
                  end
              end
              end
in
  (lcs [2,3,4]) [1,2,3,2,3,4]
end
```

GCD of [10,20,30]

```
letrec gcd = fn elms =>
                let a = (head elms) in
                let b = (head (tail elms)) in
                let a = if a > 0 then a else a * -1 in
                let b = if b > 0 then b else b * -1 in
                  if b == 0
```

```
                      then a
                      else let c = a mod b in
                         gcd [b,c]
                      end
                end
                end
                end
                end
in
  map(gcd, [[10,20,30]])
end
```

Quicksort of [10,20,30]

```
letrec quicksort =
  fn list =>
    if (length list) > 1
      then
        let x  = head list in
        let xs = tail list in
        let smallerSorted = quicksort (filter(fn y => y < x, xs)) in
        let biggerSorted = quicksort (filter(fn y => y >= x, xs)) in
          concat3(smallerSorted, [x], biggerSorted)
        end
        end
        end
        end
      else
        if (length list) > 0
          then
            let x = head list in
              [x]
             end
          else []
in
  quicksort [10,20,30]
end
```

APPENDIX C — COMPILED FIBONACCI CODE

```
main:
    goto continue1
_fn_0:
    x = GetCallRecordParam callRecord 0
    var2 = x
    var3 = 2
    resp = var2 < var3
    if resp goto then2
dummy2:
    var0 = NewCallRecord 1
    SetCallRecordFn var0 _fn_0
    SetCallRecordMissingI var0 1
    SetCallRecordCountI var0 0
    var7 = var0
    var4 = x
    var5 = 1
    param = var4 - var5
    one = 1
    var7 = var7
    missing = GetCallRecordMissing var7
    count = GetCallRecordCount var7
    new_missing = missing - one
    new_count = count + one
    SetCallRecordCount var7 new_count
    SetCallRecordMissing var7 new_missing
    SetCallRecordParam var7 count param
    resp = new_missing < one
    var12 = var7
    if resp goto then0
dummy0:
    resp = var7
    goto back0
then0:
    var6 = var7
```

```
35      var12 = Call var6
36      goto back0
37  back0:
38      var0 = NewCallRecord 1
39      SetCallRecordFn var0 _fn_0
40      SetCallRecordMissingI var0 1
41      SetCallRecordCountI var0 0
42      var11 = var0
43      var8 = x
44      var9 = 2
45      param = var8 - var9
46      one = 1
47      var11 = var11
48      missing = GetCallRecordMissing var11
49      count = GetCallRecordCount var11
50      new_missing = missing - one
51      new_count = count + one
52      SetCallRecordCount var11 new_count
53      SetCallRecordMissing var11 new_missing
54      SetCallRecordParam var11 count param
55      resp = new_missing < one
56      var13 = var11
57      if resp goto then1
58  dummy1:
59      resp = var11
60      goto back1
61  then1:
62      var10 = var11
63      var13 = Call var10
64      goto back1
65  back1:
66      Wait
67      resp = var12 + var13
68      goto back2
69  back2:
70      goto continue0
```

```
then2:
    resp = 1
    goto back2
continue0:
    return resp
continue1:
    var0 = NewCallRecord 1
    SetCallRecordFn var0 _fn_0
    SetCallRecordMissingI var0 1
    SetCallRecordCountI var0 0
    var15 = var0
    param = x
    one = 1
    var15 = var15
    missing = GetCallRecordMissing var15
    count = GetCallRecordCount var15
    new_missing = missing - one
    new_count = count + one
    SetCallRecordCount var15 new_count
    SetCallRecordMissing var15 new_missing
    SetCallRecordParam var15 count param
    resp = new_missing < one
    if resp goto then3
dummy3:
    resp = var15
    goto back3
then3:
    var14 = var15
    resp = Call var14
    goto back3
back3:
    Wait
    return resp
```