TECHNISCHE
UNIVERSITÄT
DARMSTADT

Optimization of the Memory Subsystem of a Coarse Grained Reconfigurable Hardware Accelerator

vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Darmstadt

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

Dissertation

von

Lukas Johannes Jung

Erstgutachter:	Prof. Dr.-Ing Christian Hochberger
Zweitgutachterin:	Prof. Dr.-Ing. Diana Göhringer

Darmstadt 2019

Lukas Johannes Jung:
Optimization of the Memory Subsystem of a Coarse Grained Reconfigurable Hardware Accelerator
Darmstadt, Technische Universität Darmstadt
Jahr der Veröffentlichung der Dissertation auf TUprints: 2019
Tag der mündlichen Prüfung: 27.03.2019

Erklärung laut Promotionsordung

§ 8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

§ 8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

§ 9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

§ 9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

L. J. Jung

Kurzfassung

Schnelle und energieeffiziente Datenverarbeitung ist seit jeher eine wichtige Anforderung an Prozessorentwicklung. Aktuelle Entwicklungen in Bereichen wie zum Beispiel Bildverarbeitung verstärken diese Anforderungen. Die Nutzung von vielen mobilen Endgeräten steigert den Bedarf an energieeffizienten Lösungen. Viele Anwendungen wie zum Beispiel Fahrerassistenzsysteme setzen immer mehr auf Algorithmen aus dem Bereich Maschinelles Lernen. Hierbei müssen unter harten Echtzeitbedingungen viele Daten in kürzester Zeit verarbeiten werden. Bis zu den 1990er Jahren wurden Leistungssteigerungen in Prozessoren meist dadurch erreicht, dass neue und bessere Fertigungstechnologien verwendet wurden. Dadurch wurde es möglich, Prozessoren mit einer höheren Taktfrequenz zu betreiben, während die eigentliche Prozessorarchitektur weitestgehend unverändert blieb. Seit Beginn des einundzwanzigsten Jahrhunderts jedoch stagniert diese Entwicklung. Neuere Fertigungstechnologien ermöglichen es zwar mehr Prozessorkerne auf der gleichen Chipfläche zu fertigen, jedoch wurden kaum noch Steigerungen in der Taktfrequenz erreicht. Dies erforderte ein Umdenken in sowohl dem Entwurf von Prozessorarchitekturen als auch im Software-Entwurf. Anstatt die Leistung eines einzelnen Prozessors zu verbessern, muss nun ein zu berechnendes Problem so formuliert werden, dass es in kleinere Teile aufgeteilt wird welche auf mehreren Recheneinheiten parallel und dadurch schneller berechnet werden können.

Ein oft genutzter Ansatz ist der Einsatz von Mehrkernprozessoren oder GPUs (Graphic Processing Units), in dem jeder Prozessorkern einen Teil des Problems unabhängig von den restlichen Kernen berechnet. Dies erfordert jedoch neuartige Programmiertechniken und bestehende Software muss umformuliert werden. Ein anderer Ansatz sind Hardware-Beschleuniger, die mit einem Prozessor verbunden werden. Hier wird für ein bestimmtes Problem eine spezielle Schaltung entworfen, die dieses Problem effizient und schnell lösen kann. Die Berechnung dieses Problems findet dann nicht mehr auf dem Prozessor statt, sondern auf dem Hardware-Beschleuniger. Der Nachteil dieser Lösung ist jedoch, dass für jedes Problem eine eigene Schaltung in Hardware entwickelt werden muss. Dies bedeutet einen hohen Entwicklungsaufwand und die Schaltung kann im allgemeinen nicht im Nachhinein geändert werden.

Diese Arbeit beschäftigt sich mit der Nutzung von rekonfigurierbaren Hardware-Beschleunigern. Diese werden während der Laufzeit umkonfiguriert, um mehrere Probleme mithilfe der gleichen Hardware beschleunigen zu können. Wenn während der Laufzeit rechenintensive Software-Abschnitte erkannt werden, so startet der Prozessor selbstständig einen Prozess, der eine Konfiguration für den Hardware-Beschleuniger berechnet. Anschließend kann diese Konfiguration geladen werden und das Problem wird effizienter und schneller auf dem Beschleuniger ausgeführt. Es wurde eine grobkörnig rekonfigurierbare Architektur gewählt, da die Komplexität eine Konfiguration zu berechnen sehr viel geringer ist als in feinkörnig rekonfigurierbaren Architekturen wie zum Beispiel FPGAs (Field Programmable Gate Array). Außerdem sind durch den vergleichsweise geringeren Mehraufwand für die Rekonfigurierbarkeit höhere Taktfrequenzen möglich als bei FPGAs. Ein Vorteil dieses Verfahrens ist, dass ein Programmierer oder eine Programmiererin keinerlei Kenntnis über die Hardware besitzen muss, da die Beschleunigung automatisch während der Laufzeit geschieht. Außerdem können bereits vorhandene Programme (bei denen möglicherweise kein Programmcode mehr vorliegt) ohne weiteren Aufwand beschleunigt werden.

Ein Problem, das für alle Rechnerarchitekturen relevant ist, ist die effiziente und schnelle Datenübertragung zwischen Recheneinheit und Hauptspeicher. Diese Arbeit konzentriert sich daher auf die Optimierung der Speicheranbindung eines grobkörnig rekonfigurierbaren Hardware-Beschleunigers. Zu diesem Zweck wurde während dieser Arbeit ein Simulator für einen Java-Prozessor entworfen, in dem ein grobkörnig rekonfigurierbarer Hardware-Beschleuniger eingebunden ist. Es wurden mehrere Verfahren entwickelt, die die Speicheranbindung des Hardware-Beschleunigers verbessern. Dies umfasst sowohl Lösungen auf Hardware-Ebene als auch Lösungen auf Software-Ebene, die bei der Generierung der Konfiguration für den Beschleuniger versuchen die Nutzung der Speicherschnittelle zu optimieren. Der entwickelte Simulator wurde genutzt, um den Entwurfsraum nach der besten Implementierung abzusuchen. Durch diese Optimierung des Speichersystems wurde eine Leistungssteigerung von 22,6 % erreicht.

Außerdem wurde während dieser Arbeit ein erster Prototyp eines solchen Beschleunigers in Hardware entworfen und auf einem FPGA implementiert, um die korrekte Funktionalität des Verfahrens und des Simulators zu zeigen.

Abstract

Fast and energy efficient processing of data has always been a key requirement in processor design. The latest developments in technology emphasize these requirements even further. The widespread usage of mobile devices increases the demand of energy efficient solutions. Many new applications like advanced driver assistance systems focus more and more on machine learning algorithms and have to process large data sets in hard real time. Up to the 1990s the increase in processor performance was mainly achieved by new and better manufacturing technologies for processors. That way, processors could operate at higher clock frequencies, while the processor microarchitecture was mainly the same. At the beginning of the 21st century this development stopped. New manufacturing technologies made it possible to integrate more processor cores onto one chip, but almost no improvements were achieved anymore in terms of clock frequencies. This required new approaches in both processor microarchitecture and software design. Instead of improving the performance of a single processor, the current problem has to be divided into several subtasks that can be executed in parallel on different processing elements which speeds up the application.

One common approach is to use multi-core processors or GPUs (Graphic Processing Units) in which each processing element calculates one subtask of the problem. This approach requires new programming techniques and legacy software has to be reformulated. Another approach is the usage of hardware accelerators which are coupled to a general purpose processor. For each problem a dedicated circuit is designed which can solve the problem fast and efficiently. The actual computation is then executed on the accelerator and not on the general purpose processor. The disadvantage of this approach is that a new circuit has to be designed for each problem. This results in an increased design effort and typically the circuit can not be adapted once it is deployed.

This work covers reconfigurable hardware accelerators. They can be reconfigured during runtime so that the same hardware is used to accelerate different problems. During runtime, time consuming code fragments can be identified and the processor itself starts a process that creates a configuration for the hardware accelerator. This configuration can now be loaded and the code will then be executed on the accelerator faster and more efficient. A coarse grained reconfigurable architecture was chosen because creating a configuration for it is much less complex than creating a configuration for a fine grained reconfigurable architecture like an FPGA (Field Programmable Gate Array). Additionally, the smaller overhead for the reconfigurability results in higher clock frequencies. One advantage of this approach is that programmers don't need any knowledge about the underlying hardware, because the acceleration is done automatically during runtime. It is also possible to accelerate legacy code without user interaction (even when no source code is available anymore).

One challenge that is relevant for all approaches, is the efficient and fast data exchange between processing elements and main memory. Therefore, this work concentrates on the optimization of the memory interface between the coarse grained reconfigurable hardware accelerator and the main memory. To achieve this, a simulator for a Java processor coupled with a coarse grained reconfigurable hardware accelerator was developed during this work. Several strategies were developed to improve the performance of the memory interface. The solutions range from different hardware designs to software solutions that try to optimize the usage of the memory interface during the creation of the configuration of the accelerator. The simulator was used to search the design space for the best implementation. With this optimization of the memory interface a performance improvement of 22.6 % was achieved.

Apart from that, a first prototype of this kind of accelerator was designed and implemented on an FPGA to show the correct functionality of the whole approach and the simulator.

Contents

Part I.

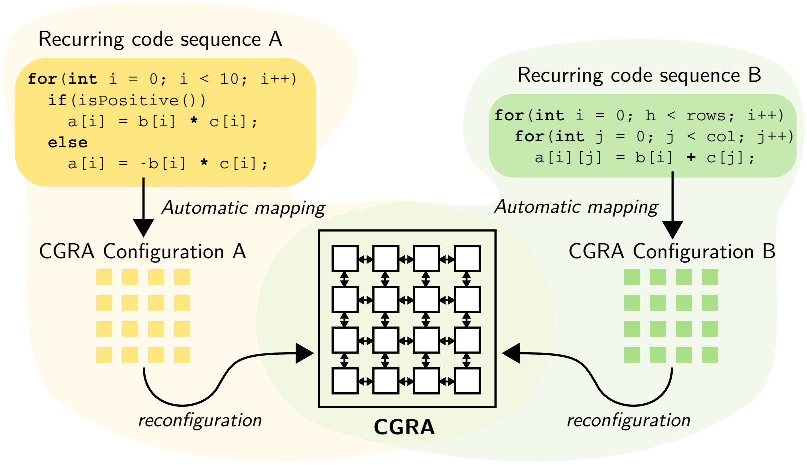
Introduction

1. Introduction

Writing working program code is easy. Writing good program code takes some effort. Writing program code that perfectly exploits all features of the machine on which it is running is hard. Especially, if the programmer doesn't have any information about the underlying hardware. Normally, this leads to two mutually exclusive design goals: The software can either be highly performant or it is cheap. In the last millennium this problem could simply be solved by waiting. The processor speeds kept increasing due to better manufacturing technologies so that after some time the written software would execute significantly faster on a new processor. Due to higher power consumption this trend stopped at around 2003. The improvements in single thread performance stagnated. To improve the performance further new concepts like multi-core processors were introduced and are being researched to exploit parallelism. Yet, in most cases this means that existing software has to be adapted so that it can benefit from these new concepts.

1.1. Motivation

In this work a new concept is used, that tries to exploit parallelism without user interaction. The idea is that the processor "knows" its own architecture so the processor itself will parallelize the code. A Coarse Grained Reconfigurable Array (CGRA) will be coupled to a host processor and will be used as an accelerator for recurring code sequences (called *kernel* in the remainder of this work) as shown in Figure 1.1. The code sequences will be mapped to the CGRA automatically and totally transparent for the user during runtime. The CGRA can be reconfigured quickly so that the same



```
for(int i = 0; i < 10; i++)
  if(isPositive())
    a[i] = b[i] * c[i];
  else
    a[i] = -b[i] * c[i];
```

```
for(int i = 0; h < rows; i++)
  for(int j = 0; j < col; j++)
    a[i][j] = b[i] + c[j];
```

Figure 1.1.: Accelerator concept used in this work

hardware can be used to accelerate different code sequences. Mapping the sequences to the CGRA during runtime gives the possibility to react to changes in both software execution or the hardware when for example a PE produces erroneous results.

This approach leads to the following benefits:

1. The programmer can write simple working code and the processor will automatically accelerate it on the CGRA so that it will be executed efficiently. This will result in performant software execution at low development cost.
2. Legacy code will be parallelized and thus accelerated automatically without user interaction or recompilation.
3. Hardware costs are low due to reconfiguration and reuse.
4. Applications can be accelerated with CGRAs of different sizes without the need to adapt the code.

The memory subsystem is a bottleneck of modern computers due to the memory wall[77][33]. The performance of the compute engine doesn't matter if the memory subsystem is not able to provide the data that has to be processed. Thus, this work concentrates on the design and optimization of the memory subsystem of the reconfigurable accelerator. The memory system is tightly coupled with the accelerator, so both parts cannot be designed independently. For that reason this work also covers significant parts of the whole CGRA based accelerator framework and not only the memory subsystem.

It is obvious that the performance of this approach can not compete with hand optimized code running on commercial high performance computers (HPC) which require a high programming effort as shown in Figure 1.2. In contrast to that, single threaded software can be written easily but results in a low performance. Multi-threaded applications exploit task level parallelism and range between HPC and single threaded applications both in performance and programming effort. The aim of this work is to increase the performance of arbitrary single threaded applications by exploiting both instruction level parallelism and loop level parallelism. At the same time the programming effort should be kept low as shown in Figure 1.2.

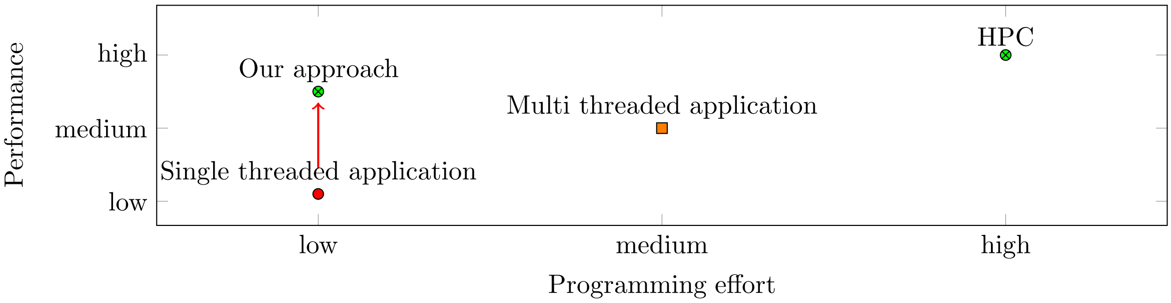


Figure 1.2.: Qualitative illustration of programming effort vs performance

Parallel patterns are often well studied and efficient implementations for different architectures are provided for common tasks like matrix multiplication or FFT. Thus,

Table 1.1.: Goals and resulting requirements

	Requirements on		
Goal	CGRA	Memory subsystem	Mapping Algorithm
Support arbitrary applications	Flexible interface to host processor	Flexible memory interface	
Reconfiguration during runtime	Quick reconfiguration times		Low complexity to minimize overhead
High performance	Support control flow on CGRA so that larger code regions can be mapped to the CGRA	Parallel memory accesses with low latency	
Low programming effort			Support high level language

in this work the focus lies on applications with irregular and possibly data dependent program flow and memory access patterns. This leads to several requirements for the CGRA based accelerator, the memory subsystem and the mapping algorithm which are listed in Table 1.1.

1.2. Contribution of this Work

The result of this work will not be to find *the* memory subsystem design which outperforms all other designs in all cases. The performance of the memory subsystem depends too much on the accelerator it is coupled to, the supported programming language and the technology that is used to implement the hardware or the application itself. Some general statements can be made but the optimal memory subsystem does not exist.

Instead this work aims to describe a way to find the best configuration for a certain setup exemplarily. A simulator of the accelerator framework will be implemented in work, which makes it possible to do huge design space explorations. This simulator should be able to run tests with more than thousand different configurations in a matter of hours with high accuracy. In order to be able to produce profound results, a prototype should be implemented in hardware, which is used to tune the simulator.

The high accuracy can only be guaranteed when the whole system and not just the memory subsystem is implemented. Otherwise assumptions are made which lead to inaccuracies. Previous publications ([20][19][15][12][13][16][18]) already described the whole system to some extend but only on a higher level of abstraction and no hardware implementation was available. Thus, the system will be (re-)implemented and significant parts of this work concern the design of the whole accelerator system as mentioned before.

This leads to the following goals that this work aims to achieve:

1. Implement a prototype of the CGRA based accelerator coupled with a processor based on the previous work mentioned above. The prototype should hold the requirements given in Table 1.1.
2. Extend the existing simulator of the CGRA based accelerator system or implement a new one. The insights of the prototype implementation should be used to achieve high accuracy while maintaining high simulation speed so that the simulator can be used to do design space explorations with profound results.
3. Use the simulator to find the optimal memory subsystem in the given setup. Which parameters are optimal and which factors influence the choice of parameters? Are there general findings?

As mentioned above, this work extends an existing framework so the the Parts I and II describe both existing and new work while Parts III and IV contain only new work.

In Appendix E a detailed description is given which parts of the existing system were just reused without change, which were adapted and which are completely new.

2. Technical Background

This chapter provides technical background which is helpful to understand the remainder of this work.

2.1. Reconfigurable Hardware

This section gives a short overview of the most common reconfigurable hardware architectures and discusses why CGRAs are used in this work. Typically, reconfigurable hardware consists of logic blocks that are connected by reconfigurable interconnect.

2.1.1. Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGA) are the most common reconfigurable hardware. As the name states, they are reconfigurable in the field which means that the hardware design can be changed even after the product has been deployed. The logic blocks in an FPGA are typically called CLB (Configurable Logic Block) and consist of look up tables (LUT), flip flops and multiplexers which can be configured and connected via switches at bit level. Thus, any application can be mapped to an FPGA. The downside is that the high hardware overhead that realizes the reconfigurability leads to relatively low clock frequencies at around 100-200MHz in the final design. Additionally, mapping the hardware design to the FPGA is a very complex task. Typically, a design is given in a Hardware Description Language (HDL) like Verilog or VHDL. Vendor tools map this design to the FPGA and create a bitfile that contains the configuration of all logic blocks and their interconnect. Creating this can take up to hours. Mapping the design that is described in this work on to a middle-class Artix 7 FPGA takes 29 minutes on an Intel i7 6700 with 3.4 MHz. The bitfile has a size of 9.7 MB and it takes milliseconds to load it onto the FPGA.

Commercial FPGAs usually also contain hard macros that are frequently used in typical FPGA designs. These hard macros range from RAM blocks (BRAM) over clock managers to DSP blocks with integrated integer adders and multipliers.

FPGAs can be reconfigured partially which can be used implement different hardware accelerators like in [28].

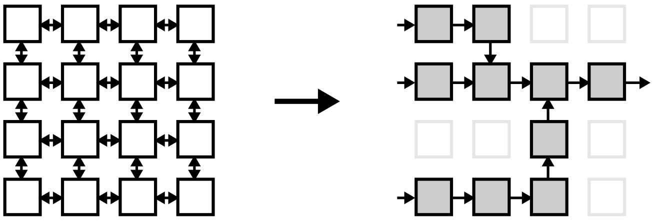


Figure 2.1.: Static kernel mapped onto a CGRA

2.1.2. Coarse Grained Reconfigurable Arrays

In contrast to FPGAs Coarse Grained Reconfigurable Arrays (CGRA) are not reconfigured at bit level but on word level. The logic blocks which are also called Processing Element (PE[1]) in this context, contain complete ALUs with one ore more registers or even a register file. PEs are connected with parallel busses that transfer one word in one cycle. The more coarse granularity limits the ability to map arbitrary code to the reconfigurable hardware (especially for operations on the bit level) but at the same time the complexity of the mapping process is reduced. Mapping a design to the CGRA can be done in less than one second and the configuration contains only around 1 kBit depending on the number of PEs and their interconnect. One configuration of a CGRA is also called *context*. FPGAs can only store one configuration in the fabric. Due to the small context size, it is possible to include a context memory on the CGRA that stores up to thousands of contexts. Once the contexts are loaded into that memory they can be switched in one clock cycle. Also, the coarse granularity results in higher frequencies. Preliminary tests have shown that frequencies of up to 1GHz are possible on 45nm TSMC technology.

CGRAs can be used in different fashions. In the **static** case a *hardware description* is mapped to the CGRA (e.g.[37] or [22]). Only a single context is used to create a specialized data path. Data is typically fed in to the CGRA on one side of the array and the results will be returned on the other side as shown in Figure 2.1. This way, efficient hardware pipelines can be created for regular and compute intense applications like image processing. Yet, only basic control flow can be included with the help of multiplexers. Also, the size of the datapaths that can be realized is limited to the number of PEs. This approach is often used in CGRAs that are included in the datapath of a CPU so that a set of instructions can be executed efficiently on that specialized datapath (e.g. [22] or [76]).

In the **dynamic** case an *application* is mapped to the CGRA. A dynamic datapath is created from several contexts. The next context that will be executed can be dependent on previous results. It is possible to realize complex control flow and larger datapaths. Hardware pipelines like in the static case can not be created. Thus, techniques like

[1] In some literature the term Functional Unit (FU) is used. In the CGRA that is used in this work, the term PE will be used, as parts of our AMIDAR processor are also called FUs as described in Chapter 6

software pipelining have to be applied when an application is mapped to the CGRA to increase parallelism.

In this work a CGRA will be used with dynamic kernels for the following reasons:

1. CGRAs can operate at higher frequencies than FPGAs
2. CGRA configurations can be computed much faster which is important for runtime reconfiguration
3. CGRAs can store multiple contexts
4. Dynamic Kernels allow the acceleration of kernels with complex control flow

Often CGRAs are created as an overlay design for FPGAs so that an existing design can be adapted quickly. In this work the proposed CGRA design will also be mapped on to an FPGA but only to test the prototype. Creating an ASIC is the goal so that higher frequencies can be used.

A detailed description of the used CGRA is given in Chapter 7.

2.2. Caches

The performance of modern memories does not match the performance of modern processors. Either the memories are fast enough to provide the required data in time or they are large enough to hold all data of an application at one time. Both at the same time is not possible with current memory technologies.

Fortunately, memory accesses typically follow two principles: First, data that is used in the processor is likely to be used again in the near future (temporal locality). Second, data that lies in the same memory region is likely to be used as well (spatial locality). Thus, it is possible to connect the processor to a small but fast memory (called cache) that holds only the subset of the data that is currently needed. Caches are organized in lines that contain data words from a contiguous memory region. When the required data is in the cache (*hit*) it can be sent to the processor directly. Only when the required data is not in the cache, the larger and slower main memory has to be accessed. In this case not only the desired data is loaded, but a whole set of data from the same region (exploiting spatial locality) is loaded into one cache line (shown for example in light blue in Figure 2.2). The main memory is traditionally realized in DRAM which has a high overhead when a memory cell is accessed but data from the same memory row can then be read with low cost. When the cache memory is full, data that has been used least recently will to be replaced (exploiting temporal locality).

In order to know which data resides in one cache line, not only the data but also meta information has to be stored. This information is called tag and contains (parts of) the original address of the data (shown in dark blue in Figure 2.2).

In fully associative caches any cache line can contain any data. This means that the tag of every cache line has to be checked in order to find out whether the desired data is currently in the cache. This is very costly and not used in practice. Instead n-way

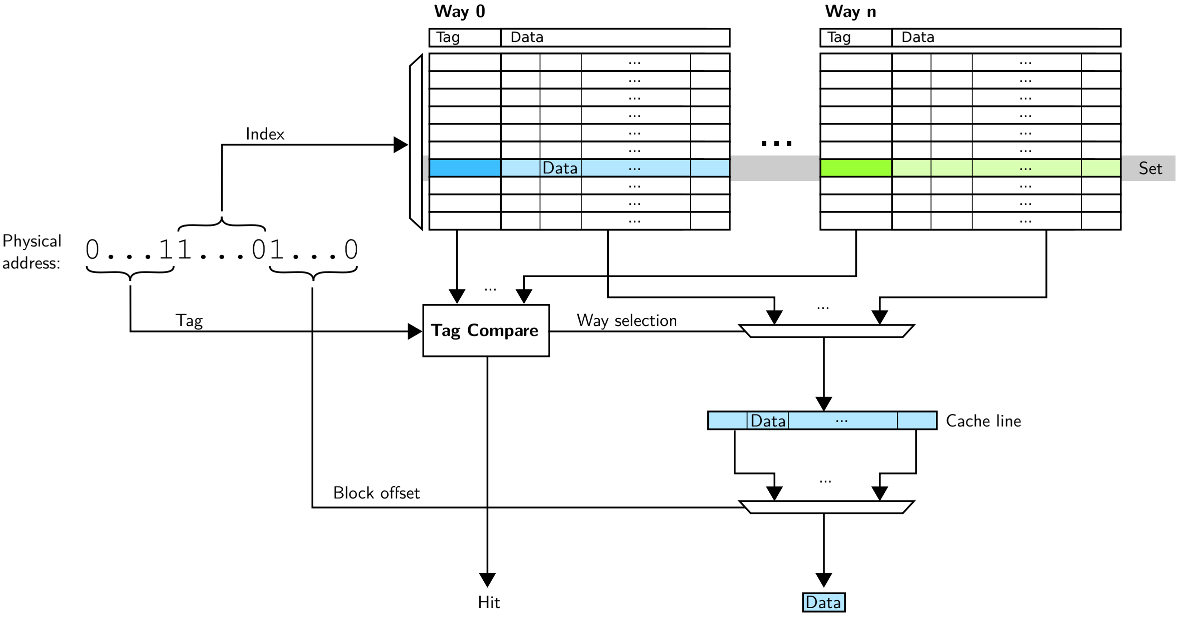


Figure 2.2.: Illustation of working principle of caches

associative caches are used. A certain data word can only be cached in one of definite n cache lines (called set - shown in Figure 2.2) which are identified by an index i which is calculated from the physical address of the data word. Thus, only the tag of n cache lines has to be checked. The correct data word can then be read from the cache line with the block offset b which is also calculated from the physical address as shown in Figure 2.2.

The average access time of a cache can then be calculated with $t = t_{cache} + p_{miss} \cdot t_{DRAM}$ where p_{miss} is the probability that the desired data is not in the cache. In a good cache this probability converges against zero. Thus, from the processor point of view the memory is approximately as fast as the cache and has the size of the DRAM.

When data is written to the cache from the processor, the data can either be written to the main memory directly (*write-through*) or only when the cache line is replaced by another cache line (*write-back*). The latter will be used in this work as it reduces the number of accesses to the main memory.

The parameters of such a cache are replacement strategy, write back strategy, number of ways, data words per cache line, number of ways and the cache size. The number of sets can then be calculated from the parameters.

Typically, more than one cache level is used. The cache size increases with each cache level while the speed decreases. Each cache accesses the next cache level only in case of a cache miss. The cache that is connected directly to the processor is called L1 cache and other levels are named with increasing number. Thus, the cache that is directly connected to the main memory has the highest number.

2.2.1. Cache Coherency

Note: Parts of this section have already been published in [61]. The marking of self-citation is omitted in order to improve the reading flow.

When the *write-back* strategy is used, modified cache lines are only written to the next cache level when that line is replaced. For example a L2 cache can contain outdated cache lines when the data has not yet been written back by a L1 cache. Those cache lines are called *dirty*. In single core systems this is not a problem, but in multi-core processors each core has a separate L1 cache which all access the main memory via a L2 cache. If two processors access the same cache line, it has to be ensured, that both caches use the same (coherent) data. Thus, the caches need to communicate which data they access.

In a **snoop based** cache system all caches on the same level listen to all transfers on the bus that connects them to the lower cache level in order to ensure coherency. In a **directory based** cache system a directory is maintained that contains which caches share which cache lines. With this information it is possible to send coherence messages only to relevant caches. This directory introduces overhead in timing and hardware costs but can reduce the communication between caches if cache lines are shared among a small amount of caches [29]. The system only benefits from this approach, if there are multiple communication channels between the caches on one level[2]. If there is only one bus, all other caches can not communicate while the coherence messages are exchanged and there is no benefit compared to a snoop based system [67].

Coherence Protocols To ensure coherence, the MSI (*Modified Shared Invalid*) protocol was developed. Here a cache is allowed to hold dirty lines only as long as no other cache needs this cache line. When another cache accesses the cache line, the new value has to be written back to the next lower cache level. The drawback of this protocol is, that a cache does not know whether the cache line resides in another cache as well. Thus, for every write access to a cache line every other cache has to be notified that it has to invalidate the cache line in case it also holds that cache line.

In order to overcome this issue, the *Exclusive* state was added in the MESI protocol. If a cache line is in the Exclusive state, no notification has to be sent to the other caches when the cache line is modified. Still if another cache wants to read a modified cache line it has to be written back to the next cache level. That way, shared cache lines are never dirty. The MOESI protocol introduces the *Owned* state which allows to share dirty cache lines. The dirty value is only written back when the cache line is replaced. This leads to even less write backs. All the protocols described above have in common that upon a write access a notification is sent and all other caches holding that cache line will invalidate this line. If the data is needed later on, it has to be reloaded.

[2] Intels Xeon Phi X100 for example uses a distributed directory. All caches are connected via a bidirectional ring so that two transmissions between two caches in each case are possible.

The Dragon protocol [3] tries to avoid these reloads by not invalidating the cache line in all other caches upon a write access. Instead, the new value is directly provided. A more detailed description of MOESI and Dragon can be found in Chapter 11.

3. Related Work

Recently many coarse and fine grained reconfigurable architectures have been researched. In most of them the memory interface is not covered in depth. Many approaches focus on accelerating applications with known regular memory access patterns. The data can easily be streamed into the accelerator so that the required data is available in the accelerator just in time (for example [8],[38],[60], [4], [65], [39]). Thus, many approaches can not handle irregular memory accesses efficiently. In other cases, the research is in early stage so that the problem of efficient memory accesses is simply not yet covered.

In the following, selected reconfigurable accelerators are described that either address general applications or represent milestones in their field of research. Afterwards, the memory subsystem of a Nvidia GPU is briefly described. In the third part of this chapter software approaches are described to increase the performance of the memory interface.

3.1. Reconfigurable Accelerators

Reconfigurable Accelerators can be divided into three categories. The first category are tightly coupled accelerators which are included into the execution stage of a processor. The second category are tightly coupled co-processors that share some hardware like caches with the processor but can execute at least simple loops autonomously. The third category are loosely coupled co-processors that only communicate with the processor via shared memory. In the following, relevant examples are given for all categories roughly sorted according to the date of publication.

3.1.1. ADRES

In ADRES (*Architecture for Dynamically Reconfigurable Embedded Systems*) a VLIW processor is coupled with a CGRA consisting of Reconfigurable Cells (RC) and FUs [48]. The *Reconfigurable Matrix* shares the FUs and the global register file with the VLIW processor as shown in Figure 3.1. ADRES supports simple control flow using predication and it realizes dynamic kernels as described above. The contexts are stored in a configuration RAM and can be switched within one cycle. Its configurations are generated by DRESC (*Dynamically Reconfigurable Embedded System Compiler*) [47] before runtime. DRESC supports modulo scheduling to exploit loop level parallelism [46] but the runtimes are high. The authors state that "it takes minutes to schedule a loop of medium size" (about 50 to 70 operands). The CGRA relies on the VLIW processor to access the memory. Several FUs are provided with access to a multiport L1 cache which is very costly [72]. In [45] a AVC decoder was implemented with ADRES.

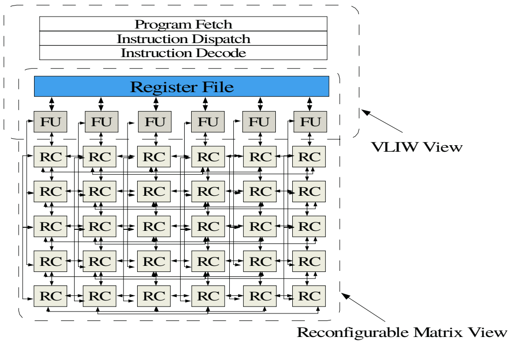


Figure 3.1.: ADRES architecture [48]

In this work the authors replaced the data cache with a software controlled data memory consisting of 4 SRAM banks that can be accessed in parallel. Using such a memory structure needs some improvements in the compiler and the ADRES architecture[44].

ADRES accelerates the VLIW processor by a factor 4.6[48]

Table 3.1.: ADRES summary

Requirement	**Realization**
Flexible memory interface	Access through a multiport L1 cache
Parallel accesses possible	Yes
Programmer transparency	CGRA mapping is done automatically by the compiler framework
Ease of use	Programmed in C
Accelerates control flow	Limited to simple constructs
Time of mapping process	Before Runtime with slow compiler

3.1.2. Warp Processor

Warp processors use a profiler to identify time consuming kernels during runtime. Those kernels are then mapped to an FPGA [42]. Mapping kernels to an FPGA is a complex task due to the fine granularity. Thus, Lysecky et al. implemented a custom FPGA for Warp (W-FPGA) which can efficiently be routed with Riverside On Chip CAD tools (ROCCAD) [43]. Still, a dedicated MicroBlaze processor is used to do the mapping on chip during runtime as shown in Figure 3.2. The W-FPGA can access the data cache (Data BRAM in Figure 3.2) with the help of an address generator in the W-FPGA interface. This generator only supports regular address patterns, like linear array accesses. Kernels with irregular accesses cannot be mapped. A loop controller is able to support loops with a specific number of iterations and also break conditions[41].

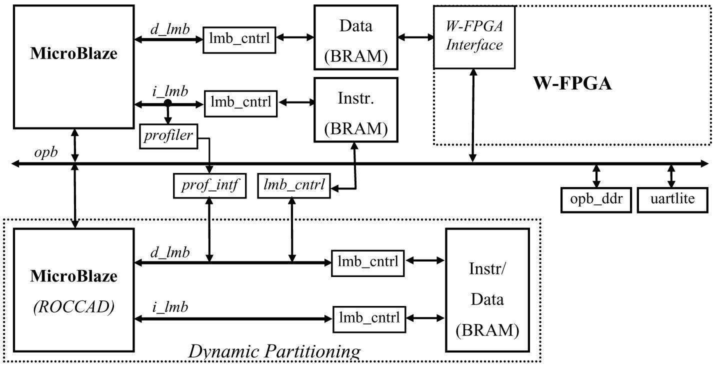


Figure 3.2.: Warp architecture [42]

Table 3.2.: Warp processor summary

Requirement	**Realization**
Flexible memory interface	Only regular patterns are supported
Parallel accesses possible	Only single port access to the cache
Programmer transparency	Yes
Ease of use	Programmed in c
Accelerates control flow	Limited
Time of mapping process	During runtime

3.1.3. DySER

In DySER (*Dynamically Specialized Datapaths*) a CGRA is used to create static datapaths that speed up recurring code sequences. Figure 3.3 shows the overview of the processor with the integrated DySER CGRA in the execution stage. The DySER compiler creates specialized datapaths and maps them to the DySer CGRA before runtime. Special DySER instructions to invoke computations on DySER are included in the program code. Before the execution can start, the FUs and the switches of DySER have to be configured by a `dyser_init` instruction[25]. The configuration bits are distributed between the FUs using the data flow network of the CGRA. The

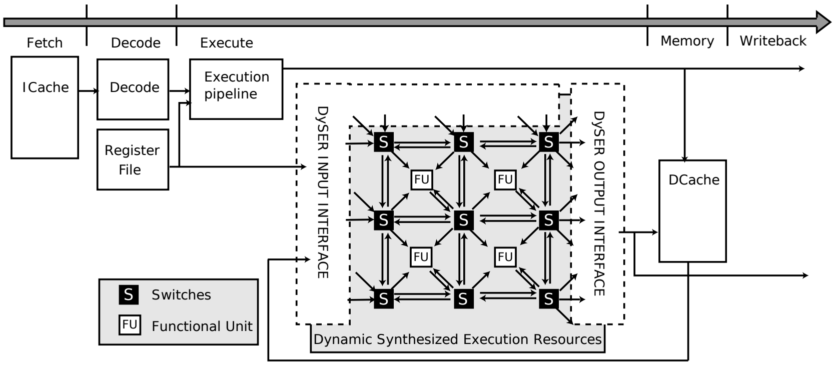


Figure 3.3.: DySer architecture [23]

instructions dyser_send and dyser_load are used to load values from the register file or the memory respectively and send it to the DySER Input Interface. When all required data is available, the execution on DySER begins. When the execution is finished, the instructions dyser_recv and dyser_store are used to write values from the DySER Output Interface back to the register file (in the writeback phase) or the memory respectively. The execution on DySER can be pipelined as shown in Figure 3.4

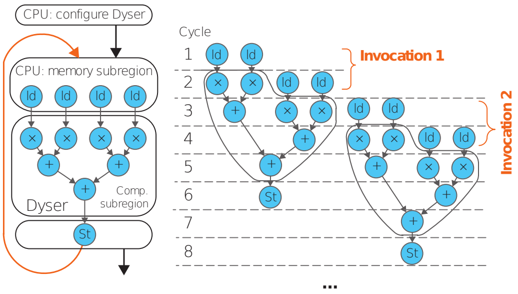


Figure 3.4.: Pipelining on DySER [22]

Simple control flow like an if/else construct can be realized on DySER with the help of predication signals that are distributed among the FUs.

The DySER compiler can identify critical code regions automatically which leads to a speedup of about 3 compared to the OpenSPARC processor. The programmer has the possibility to support the compiler with pragmas which increases the performance by a factor of 2[27].

The authors of [27] implemented a prototype of DySER on an FPGA coupled with an OpenSPARC processor. They found that speculative loads / stores and address aliasing from the "early-stage design [...] proved overly complex" so that they were omitted in the prototype. This leads to very limited speedup in the SPECINT benchmarks (and even slowdown in one case). The following reasons can be identified:

- Only limited control flow is supported. DySER does not support data dependent control flow [22] and the prototype doesn't even support conditional memory accesses[27]. Loop carried dependencies can not be mapped efficiently.

- DySER relies on the processor to provide data from the memory as shown in Figure 3.4. No parallel accesses are possible an thus the DySER array has to wait on the data from the memory and the ability to pipeline iterations can not be exploited [27].

- Irregular memory accesses can not be mapped to the DySER array [22] efficiently.

Table 3.3.: DySER summary

Requirement	**Realization**
Flexible memory interface	Irregular memory accesses cannot be mapped efficiently
Parallel accesses possible	CGRA relies on the host processor to load/store data
Programmer transparency	Pragmas can be used
Ease of use	Programmed in c
Accelerates control flow	Limited to simple constructs
Time of mapping process	Before runtime

3.1.4. Layers CGRA

In the Layers CGRA[58] a memory layer is responsible to access the main memory as shown in Figure 3.5. P memory banks can be accessed in parallel. They execute load and store patterns which are activated by a state machine (see *q_state_reg* in Figure 3.5). Deriving efficient patterns for a kernel is a complex task as shown in [59]. The N^2 PEs can access the memory banks via register clusters in the communication network. Each register is connected to one of the P hubs in the memory layer[57]. Each hub can access each memory bank via $P \times P$ crossbar. The state automaton also controls the communication and the computation layers but apparently no feedback from the computation layer to the state automaton exists. Thus, it is not possible to realize data dependent control flow on the CGRA.

Table 3.4.: Layers CGRA summary

Requirement	**Realization**
Flexible memory interface	Only patterns are supported
Parallel accesses possible	Memory banks allow parallel accesses (via an expensive $P \times P$ crossbar)
Programmer transparency	Application has to be designed specifically for the Layers CGRA
Ease of use	Efficient memory access patterns have to be derived
Accelerates control flow	Limited to simple constructs
Time of mapping process	Before runtime

3.1.5. Accelerating Megablocks

Paulino et al accelerate so called megablocks with a CGRA[53] called RPU (Reconfigurable Processing Unit) which is loosely coupled to a MircoBlaze processor. Megablocks are code sequences that are executed regularly. Typically, they contain several basic blocks from different branches. That means that it is possible to accelerate code with control flow if the control flow is regular and predictable, because the most common

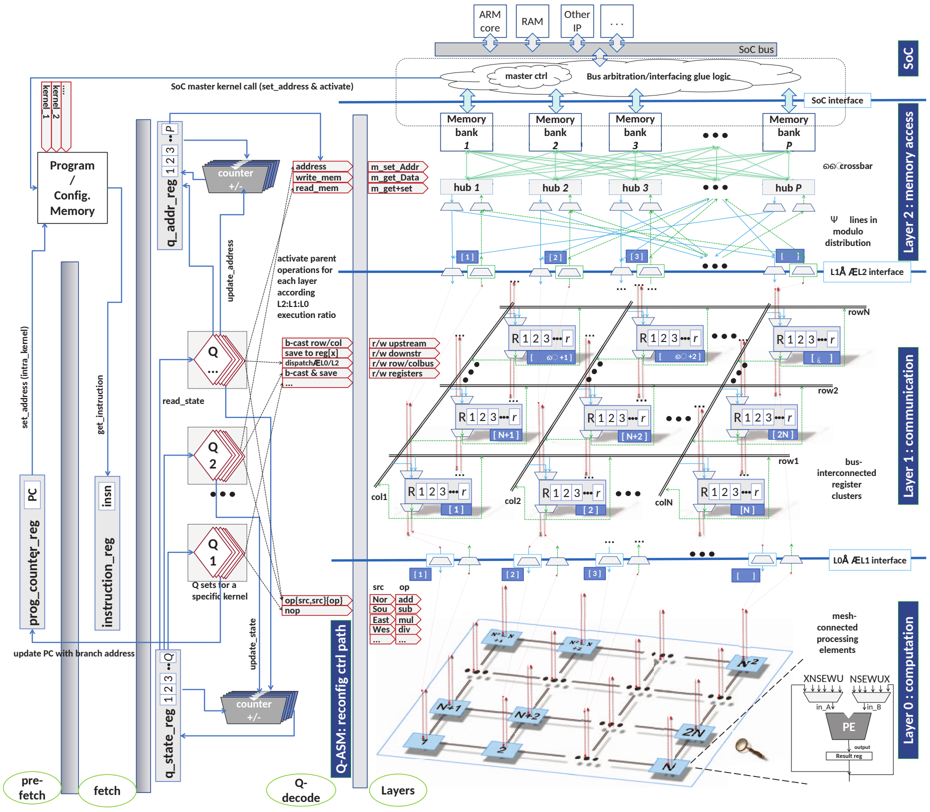


Figure 3.5.: Architecture of Layers CGRA [57]

branches are mapped to the RPU. If actually another branch is taken, the execution on the RPU has to be stopped. Before runtime an Instruction Set Simulator is used to extract megablocks from the execution traces. With this knowledge an application specific RPU and the corresponding configurations for all megablocks are generated [54]. The RPU accesses the main memory via a dual ported cache as shown in Figure 3.6. The cache is direct mapped and supports two parallel accesses. The accesses have

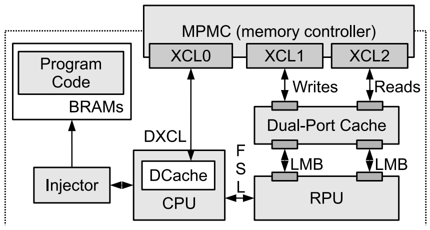


Figure 3.6.: Architecture of Paulino et al's approach [53]

a delay of four clock cycles but can be pipelined [53].

The RPU is loosely coupled with the CPU. This means that no coherence protocol can be implemented. Before the execution on the RPU starts, both the cache in the CPU and and the RPU cache are invalidated. In the RPU cache this can be done in one clock cycle whereas the CPU cache needs 192 clock cycles. When the execution on the RPU is long enough, this time introduces no delay, as it can be done in the background. For kernels with small execution times, this means a significant overhead. A write through strategy is used in the RPU cache to ensure that the correct data will be loaded into the CPU cache afterwards.

Table 3.5.: Accelerating megablocks summary

Requirement	**Realization**
Flexible memory interface	Yes, but with high delay and no coherency support. Thus, invalidation is needed
Parallel accesses possible	Two parallel accesses are possible
Programmer transparency	No programmer interaction needed
Ease of use	RPU and configurations have to generated before runtime with the help of traces
Accelerates control flow	Limited to the most common control flow path
Time of mapping process	Before runtime

3.1.6. Plasticine

Plasticine [56] is a relatively new approach which targets parallel patterns. A mesh constructed from Pattern Compute Units (PCU) and Pattern Memory Units (PMU) is connected via switches as shown in Figure 3.7. Each PCU consists of a small CGRA that computes inner loops and supports parallel data transfers in order to perform SIMD instructions. Within this CGRA different rows are used in parallel to exploit loop level parallelism and the columns represent pipeline stages.

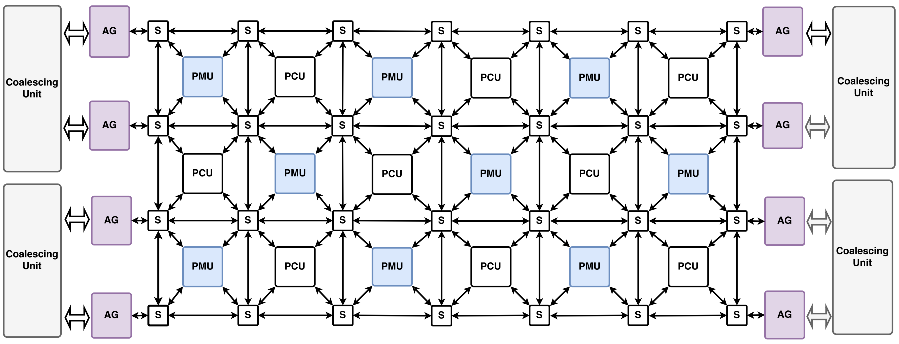


Figure 3.7.: Architecture of Plasticine [56]

PMUs contain scratchpad memories that have to be managed by the programmer to buffer data on chip. Ideally, those scratchpads are not used and the produced data is directly forwarded to the next PCU. The off chip DRAM memory can be accessed via four channels. Address generators are used to generate the read and write addresses according to the input and output memory access pattern of the accelerated parallel pattern. In case of sparse memory access patterns, a coalescing unit combines several requests in order to minimize the DRAM accesses. Otherwise, burst requests are issued.

As this approach focuses on parallel patterns, no sophistic control flow mechanisms are needed. Tokens are used to ensure correct execution when loop carried dependencies exist and a simple enable signal is used in streaming applications.

Plasticine is programmed using the parallel pattern based language DHDL (*Delite Hardware Description Language*). Supported patterns are for example *map* or *fold*. The pattern *map* takes a vector a_i of length n as inputs and calculates an output vector $b_i = f(a_i)$ where the function f is independent in each case. The pattern *fold* first calculates a *map* and then reduces the output vector b_i to a scalar value using the function $z = g(x, y)$ multiple times.

The authors compared the performance and energy consumption of Plasticine to a Stratix V FGPA. They report a 95x improvement in performance and a 77x improvement in performance per Watt[56]. The mapping process is only a matter of minutes on Plasticine while it can take up to hours to map a design on an FPGA.

While this approach is very promising to accelerate parallel patterns, it cannot unfold its potential to accelerate applications with irregular memory access patterns or data dependent control flow as it uses counters to control the loop iterations.

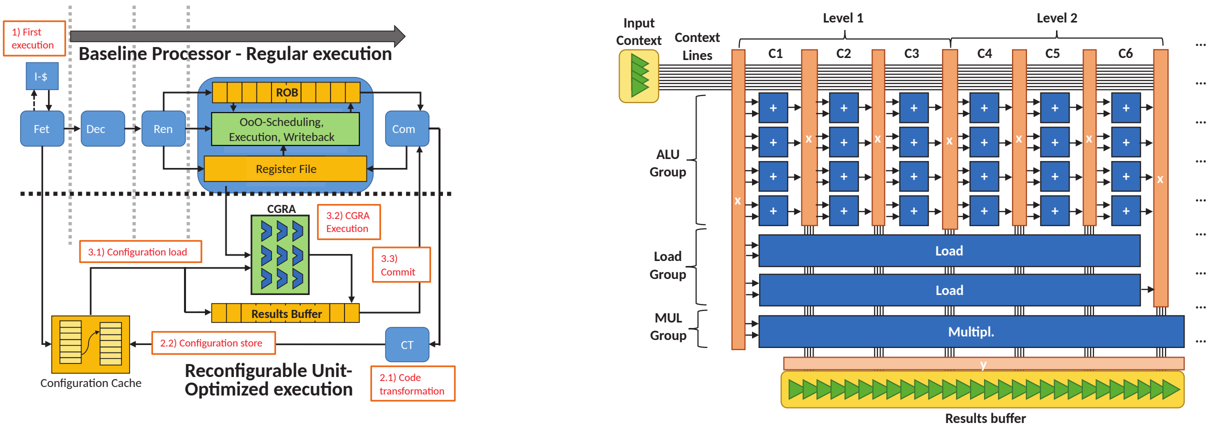
Table 3.6.: Plasticine summary

Requirement	**Realization**
Flexible memory interface	Coalescing Units combine sparse memory accesses
Parallel accesses possible	DRAM can be accessed via four channels
Programmer transparency	Programmer has to use parallel patterns
Ease of use	Programmed in DHDL
Accelerates control flow	Regular nested loops are supported
Time of mapping process	Before runtime

3.1.7. Accelerating x86 Instruction Streams

Brandalero and Beck proposed a CGRA that is tightly coupled to a superscalar processor with out of order execution as shown in Figure 3.8 (a). When the code is executed for the first time on the processor, it will be executed regularly on the superscalar processor. A Code Transformation (CT) module observes the execution and generates CGRA configurations that describe static data graphs for recurring kernels. The configurations are stored in a configuration cache. When the kernel is executed the next time, it is checked in the fetch stage whether there is a CGRA configuration in the configuration cache. If yes, the configuration and input values from the register file are loaded and the kernel is executed on the CGRA. When the execution is finished, the results are committed in the register file.

The structure of the CGRA is shown in Figure 3.8 (b). The CGRA is organized in rows and columns. The rows contain different PEs like adders or load units. The PEs of two



(a) System Overview

(b) CGRA in detail

Figure 3.8.: Architecture of Brandalero and Becks approach[7]

columns are connected by crossbar networks (marked with x). Data always flows from left to right. The delay of an adder is 1/3 of a clock cycle. So one level that consists of three columns corresponds of one clock cycle. If an operation like multiplication has a latency longer than the addition, it spans over several columns or even levels.

The CGRA only consists of combinatorial logic and has no registers. That means, that if a kernel is mapped onto n levels of the CGRA, the results can be read from the results buffer only after n cycles. The more levels the CGRA has, the bigger are the kernels that can be mapped. At the same time, the energy consumption increases. The authors found that a CGRA with 30 levels give the best trade-off. Two load units (4 cycles latency) and one store unit (1 cycle latency) access the 3-level cache hierarchy of the base processor which has a latency of 4 clock cycles. The load store units maintain a request queue to avoid pipeline stalls.

Control flow is realized with speculation. Several control flow paths are mapped onto the CGRA but only the results of valid paths are committed. Similar approaches are described in [68] and [5]

Table 3.7.: Acceleration of x86 instruction streams summary

Requirement	**Realization**
Flexible memory interface	Yes
Parallel accesses possible	Yes via two loads and one store unit
Programmer transparency	Transparent for the programmer
Ease of use	Programmed in C
Accelerates control flow	Simple control flow is supported
Time of mapping process	During runtime

3.2. Graphic Processing Units

As the name suggests, graphic processing units (GPU) are originally used to speed up image processing for example for computer games. Yet, GPUs are more and more used to accelerate code from other applications domains like machine learning. Those application domains have in common that huge amounts of data have to be processed quickly. Normally, the algorithms include a huge amount of parallelism on thread and loop level. GPUs are designed to exploit this parallelism. GPUs target other goals than this work but a huge challenge of the GPU design is to load the data into the processing elements quickly. Thus, it is helpful to have a quick look at the memory subsystem of a GPU.

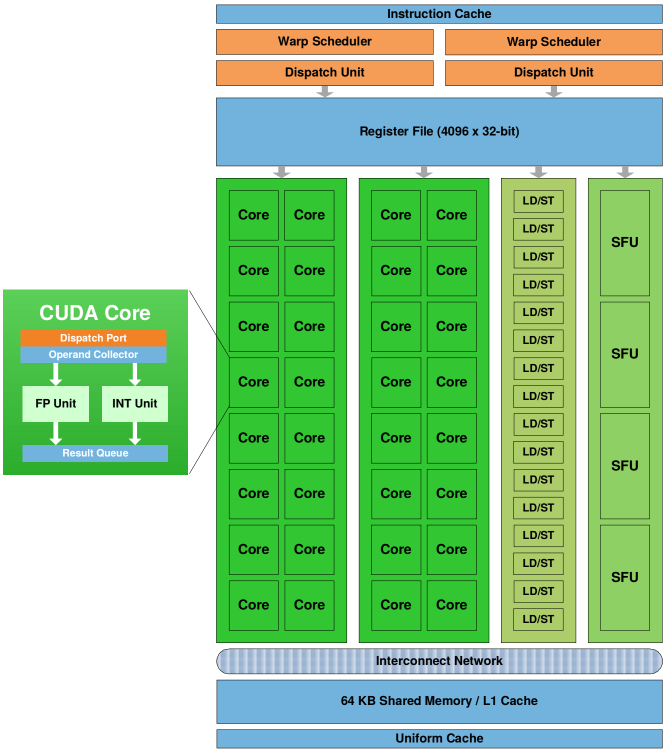


Figure 3.9.: Architecture of a Nvidia Fermi Streaming Multiprocessor [52]

Figure 3.9 shows the architectural overview of the Nvidia Fermi Streaming Multiprocessor (SM). Each SM consists of 32 Cuda cores (including integer and float ALUs), 16 Load/Store units and 4 Special Functional Units (SFU) to calculate for example trigonometric functions or roots. The Nvidia GPU consists of 16 SMs which share a L2 cache, which results in 512 Cuda Cores in one Fermi GPU. Newer Versions like the Volta GPU eva have up to 5120 Cuda Cores.

The SM contains a L1 cache which can also be used as a software managed shared memory [52]. Using shared memory is highly efficient but requires programming effort. Using the L1 cache instead trades performance against less programming effort and allows efficient accesses to addresses that are not known beforehand. The programmer can chose an approach depending on the software design goal. Nvidia states that "While shared memory remains the best choice for maximum performance, the new Volta L1 design enables programmers to get excellent performance quickly, with less programming effort"[51]. This suggests that using L1 caches in this work is a good choice to accelerate irregular applications with low programming effort.

It has to be noted that GPUs have a dedicated DRAM. Thus, every data that has to be processed has to be copied from the processor main memory to the GPU memory and results have to be copied from the GPU memory back to the main memory.

3.3. Compiler-based Approaches

Some research focuses on the smart mapping of memory instructions to PEs in order to improve the memory access times. Little research projects focus on this topic, as most CGRAs are used to accelerate streaming applications and assume the data is available just in time. In the following two examples are explained in more detail.

3.3.1. Alleviating the Memory Bandwidth Bottleneck

The authors of [10] assume a two dimensional CGRA in which all PEs are connected to a L1 cache via one data bus. Each PE has a local RAM to store intermediate values and operands. In this work the local RAMs will be used as L0 cache. If desired data lies in one of these RAMs, the mapping algorithm can either load it from the L1 cache or from a local RAM. If it is loaded from the RAM, routing delay might be introduced when the PE holding this data is not directly connected to the PE that requires that data. If it is loaded from the L1 cache, the data bus is blocked for other accesses. The mapping tool comes to a decision based on a cost function that evaluates routing and cache access delays.

This procedure decreases the load on the data bus but it only works if data is reused in the control and data flow graph that is mapped to the CGRA. The authors show that this procedure works well when loops are unrolled, because this results in more data reuse in the control and data flow graph. This mechanism to decrease the load on the L1 cache is also called load forwarding. This will also be used in this work as shown in Chapter 10.

3.3.2. Memory-Aware Application Mapping

The authors of [35] use the same technique to minimize the number of memory accesses. In contrast to the previous approach they assume a different CGRA structure. Here selected PEs are connected to different memory banks which are filled with double buffering as shown in Figure 3.10.

The compiler tries to classify the kernel that is mapped to the CGRA. If a kernel is compute intense, it is mapped in a way that routing paths on the CGRA are short even

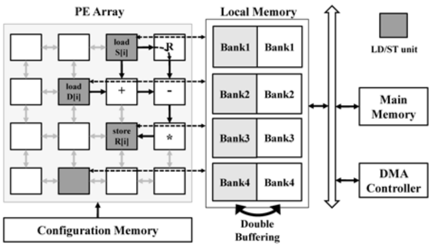


Figure 3.10.: Architecture of the CGRA used to do memory-aware application mapping [35]

if this results in a suboptimal memory bank usage. If the kernel has many memory operations, longer routing paths are allowed in order to optimize the memory bank usage as described above. This is done by a compiler that tries to minimize duplicates of arrays in the banks so save memory by applying smart mapping of memory instructions to the corresponding PEs.

3.4. Summary

It can be seen that none of the described approaches satisfies all requirements mentioned above. Some approaches focus solely on regular memory accesses while others rely on the single cache of the host processor. Only Brandalero and Becks approach fulfills all requirements but only limited control flow is supported.

Four key findings can be drawn from the study of related work:

1. Using caches is mandatory if irregular memory accesses are to be supported.
2. Using a single cache is not sufficient as only one word can be read in one clock cycle. Multi-port caches are too expensive (see ADRES)[72]. Instead multiple caches have to be used.
3. Coherence not only between the caches of the accelerator but also in the host processor has to be ensured (see the acceleration of Megablocks [53]).
4. The mapping algorithm can support the performance of the memory subsystem as shown in [35] and [10].

4. Description of Our Approach

In this work a CGRA based accelerator will be tightly coupled to a Java processor. During runtime a profiler will identify hotspots in the code that consume substantial runtime. These kernels will then be mapped automatically to the CGRA during runtime in order to accelerate the execution. The mapping algorithm will run as a separate task on the Java processor totally transparent for the programmer as shown in Figure 4.1. Changing program flow is detected by the profiler and new kernels can always be mapped to the CGRA.

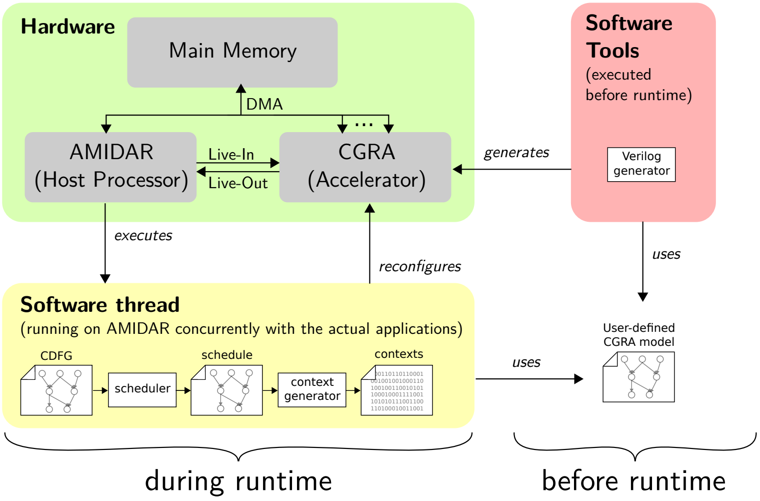


Figure 4.1.: Overview over the whole system

Coupling the CGRA based accelerator to a Java processor has two benefits: First, the programmer does not have to learn any specialized language like Cuda to make use of the accelerator. Second, analyzing Java byte code during runtime in the mapping process is less complex than analyzing for example native ARM machine code. Properties such as the independence of two memory operations can be proved much easier as shown in Section 10.2.

The used CGRA was specifically designed to be able to execute kernels with complex data dependent control flow autonomously.

As mentioned before, it is a goal to accelerate arbitrary code independent of the application or even application domain. Thus, at design time of the accelerator the code structure and the memory access patterns of the application are not known. This leads to several challenges that will be discussed in the following sections.

4.1. Problem Formulation

Designing the memory subsystem independently of the rest of the system is not possible as we will see in the following. This section describes the problems and challenges that have to be solved in order to be able to evaluate the whole system.

4.1.1. Full Stack Main Memory Interface Optimization

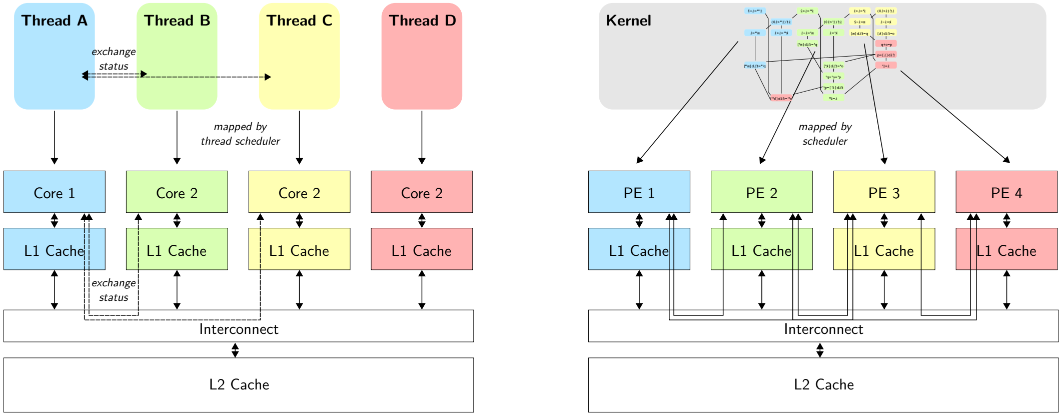
Accessing the main memory is the bottleneck in both performance and energy consumption for many applications [70].

Many accelerator techniques such as Plasticine [56] heavily rely on a priori knowledge of the access patterns. This knowledge is used to implement streaming engines that load the desired data and transfer it to the accelerator in time. Other approaches like Layers CGRA [59] even rely on the programmer to load the data and distribute it in different memory banks in the accelerator to enable efficient parallel memory accesses.

As mentioned above, in this work the memory access patterns are not known a priori and can be irregular. Thus, the CGRA will be equipped with direct access to the memory so that the CGRA can load the desired data on demand. Multiple caches will be used to allow parallel accesses to the memory with small average memory access times. Prefetching will be used to fill the caches efficiently.

Memory subsystems of multi-core systems have been studied thoroughly but they have different characteristics. Several software threads are designed so that they use as little common variables as possible to exchange for example status messages. They will mostly run independently and can be optimized independently. Those threads will be mapped by a thread scheduler to the different cores as shown in Figure 4.2 (a). As those threads are normally communicating only sparsely, the communication between the cores and the number of shared cache lines is low. Hence, the load on the interconnect between the L1 caches is low. The design of the memory subsystem, the thread scheduler and the threads itself are almost decoupled and each part can be optimized independently.

When a kernel is mapped to the CGRA, the nodes of a complex graph are mapped to the different PEs as shown in Figure 4.2 (b). Those nodes have strong dependencies and they may access similar memory regions which results in many shared caches lines. This increases the load on the interconnect between the L1 caches tremendously. The decisions that are made during the mapping process strongly influence the performance of the memory subsystem. For example, when the mapping algorithm is not able to prove that two memory accesses are independent, the capability of the memory subsystem to allow parallel memory accesses cannot be exploited. Also, it is possible to map two memory accesses to adverse PEs so that many cache misses occur as described later in Section 11.2. Thus, unlike in multi-core systems, the several parts can not be optimized independently. Instead the whole system consisting of kernel detection, graph generation, mapping process and memory subsystem has to be optimized jointly. This increases the design space tremendously. Implementing and debugging all options in



(a) Multi-core system

(b) CGRA

Figure 4.2.: Comparision of memory subsystem usage

hardware is a tedious task and mapping such a design to an FPGA takes tremendous amount of time. Thus, a fast and accurate simulation framework is needed for an efficient design space exploration.

4.1.2. Local Variables Interface

Code Example 1 shows the code to find the number of the first 42 array elements that are greater than 314. It can be seen that before the execution of the loop the potential accelerator has to know the values of the variables `array` and `cnt`. Those variables are called *Live-In* variables. Additionally, the values of the constants `1`, `42` and `314` have to be made known to the accelerator.

Code Example 1: Code Example showing *Live-In* Variables (green), *Live-Out* variables (blue) and Constants (yellow)

```
for (i =0; i< 42 ; i = i + 1 ; ) do
    if ( array [i] < 314 ) then
        cnt = cnt + 1 ;
```

When the execution is finished, the value of the variable `cnt` has to be transferred from the accelerator back to the host computer. Such variables are called *Live-Out* variables. Transferring *Live-In* and *Live-Out* variables and constants between accelerator and host computer results in an overhead that is negligible for loops with long runtimes but has a large impact on the performance of the accelerator for smaller loops. The *Live-In* and *Live-Out* variables are different for each kernel. Thus, it is important to implement an efficient and flexible interface between the *Local Variable Memory* of the host computer and the accelerator. A thorough discussion can be found in Chapter 8.

4.1.3. Context Memory Interface

The CGRA configuration for a kernel is stored in several contexts. Once, those contexts are loaded in the context memory of the CGRA, they can be switched in a single cycle. Still, loading the contexts in the memory generates some overhead. Thus, preloading the contexts parallel to the normal code execution is desirable to mask this overhead. This problem is independent of the design of the memory subsystem and it is not the focus of this thesis. Therefore, it is only covered briefly in this work.

4.2. Thesis Outline

This chapter concludes Part I of this thesis which contains the introduction.

In the next Part, the existing system is described with several enhancements that were implemented during this work. Chapter 5 explains the basic concepts behind the programming language Java. Chapters 6 and 7 describe the host processor AMIDAR and the CGRA which is used as accelerator. Afterwards, Chapter 8 describes the interface between both. This part is concluded with the description of the kernel mapping algorithm in Chapter 9.

Part III covers the optimization of the memory subsystem. Chapter 10 describes the high level compiler optimizations that are included in the kernel mapping algorithm in order to exploit the capabilities of the memory subsystem fully. Afterwards, Chapter 11 describes the actual memory subsystem including the cache architecture and coherence protocols. Chapter 12 describes *Lookahead Prefetching* which improves the performance of the memory subsystem further. This part is concluded in Chapter 13 with a description of the implemented memory subsystem.

Finally, the whole system is evaluated in Part IV with the AMIDAR simulator which is described in Chapter 14. Chapter 15 describes the benchmarks and the evaluated CGRA instances while Chapter 16 contains the actual design space exploration. The results are shown in Chapter 17. This work is then concluded in Chapter 18 with a summary and an outlook on future works.

Part II.

System Description

5. Java as Instruction Set Architecture

This chapter aims to explain the basic processes behind the programming language Java rather than explaining the programming language itself, as it is already well known. Further details can be found in [21].

First, the Java memory system will be explained because this has some impact on the design oft the memory subsystem of the whole system including the AMIDAR processor and the CGRA based accelerator. Afterwards, information about the Java bytecode and method calls is given. This is necessary to understand the process that maps Java bytecode to the accelerator.

5.1. Java Memory System

The Java memory system is divided in the parts Local Variable Memory, *Stack Memory* and the *Heap Memory*. As the name suggests, the *Local Variable Memory* stores local variables. In contrast to that, the *Heap Memory* contains all objects and arrays[1]. Operands of an operation can be loaded both from *Local Variable Memory* or *Heap Memory*. Values will be pushed to the *Stack Memory* prior to the execution of the operation. When the operation is executed, the operands are popped from the stack and the result is pushed. Afterwards, the value result is again popped from the *Stack Memory* and stored in the *Local Variable Memory* or *Heap Memory*, respectively. Both *Local Variable Memory* and *Stack Memory* are located inside of the processor and can be accessed directly. The heap is typically too big to be part of the processor. Thus, it is for example realized as DRAM.

The code example shown in Listing 5.1 is used to explain the memory accesses.

5.1.1. Heap Memory

In this work memory objects are identified by a unique handle. The *Heap Memory* is then addressed indirectly with the handle of the memory object and the offset. A dedicated memory called *Handle Table* stores the physical addresses of each memory object[2] as shown in the lower left corner of Figure 5.1. With *addr = physical address + offset* the desired object field or array element can be loaded. This addressing scheme will be called *virtual addressing* in the remainder of this work. The advantage of this virtual addressing scheme is that the Garbage Collection is eased as described in Section 5.1.4. Figure 5.1 shows on the right side how the object `c` (green) and the array `value` (blue) are stored in the heap after the constructor call in line 11 of Listing 5.1.

[1] In the remainder of this work the term *memory objects* is used to refer to both arrays and objects.

[2] Note that the *Handle Table* also contains additional information like object/array size or class type.

Listing 5.1: Memory access example

```
1  class Container{
2    int [] values;
3    int size;
4
5    public Container(int size){
6      this.size = size;
7      this.values = new int[size];
8    }
9
10   public static void main(String [] args){
11     Container c = new Container(42);
12
13     System.out.println(c.getValue(true, 2));
14   }
15
16   public int getValue(boolean addOffset, int position){
17     int offset = 18;
18     if(addOffset){
19       return values[position] + offset;
20     } else {
21       return values[position];
22     }
23   }
24 }
```

5.1.2. Local Variable Memory

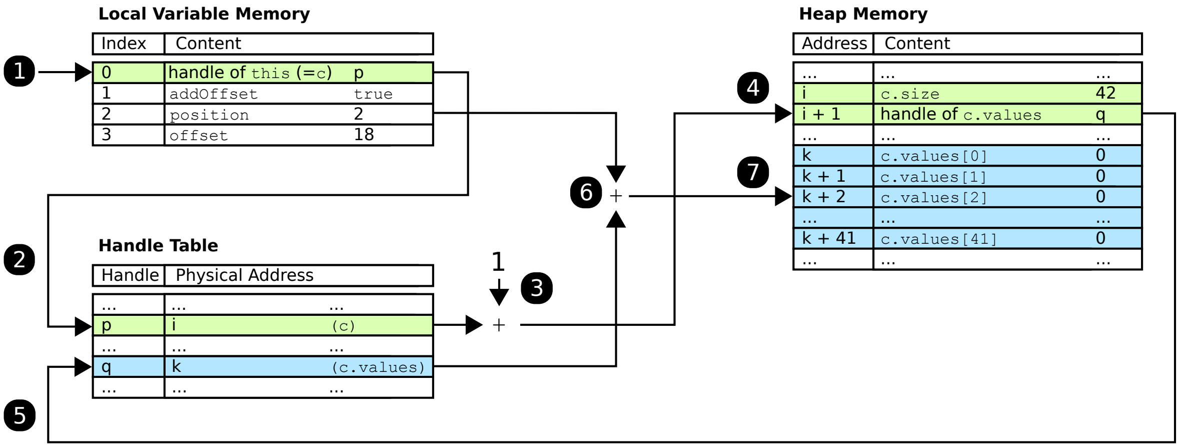
Local variables are variables that are only defined within one method. The main method in Listing 5.1 for example has the local variables `args` and `c`. The *Local Variable Memory* is simply addressed by an index. In object methods the handle of the `this` object is stored at index zero followed by the method parameters and all local variables that are declared in the method. Figure 5.1 shows all local variables of the method `getValue` in the upper left corner.

When a new method is called, the *Local Variable Memory* has to be saved so that the new *Local Variable Memory* can be initialized. When this method returns the old *Local Variable Memory* has to be restored. The same holds for the *Stack Memory*.

5.1.3. Heap Access Example

In order to obtain the value of the expression `values[position]` in line 19 in Listing 5.1 the following steps (also shown in Figure 5.1) have to be executed when `getValue` is called on object `c` in line 13:

1. Load handle of `c` from *Local Variable Memory* index = 0



Local Variable Memory

Index	Content	
0	handle of this (=c)	p
1	addOffset	true
2	position	2
3	offset	18

Handle Table

Handle	Physical Address	
...	...	...
p	i	(c)
...	...	...
q	k	(c.values)
...	...	...

Heap Memory

Address	Content	
...	...	...
i	c.size	42
i + 1	handle of c.values	q
...	...	...
k	c.values[0]	0
k + 1	c.values[1]	0
k + 2	c.values[2]	0
...	...	...
k + 41	c.values[41]	0
...	...	...

Figure 5.1.: *Heap Memory* access example without cache

2. Load physical address of c from *Handle Table* (= i)

3. Calculate i + 1 to get the address where the handle of `c.values` is stored

4. Load the handle of `c.values` from the *Heap Memory*

5. Load physical address of `c.values` from *Handle Table* (= k)

6. Calculate k + `position` to get the address of the `values[position]`

7. Load `values[position]` from the *Heap Memory*

From this example it is clear that heap accesses are expensive and it is beneficial the to use a cache for the *Heap Memory* that is virtually addressed with a combination of handle and offset.

5.1.4. Garbage Collection

In Java the programmer is relieved of the task to free memory manually. The Garbage Collector task is executed repeatedly and finds *dead objects* that can not be accessed by the running program because there are no references from the running program to the object any more. Those dead objects are deleted and the memory space can be reused. Over time this leads to a fragmented memory. Thus, the Garbage Collector moves living objects in the memory to defragment it. When objects are moved in the memory, all references to the original memory address have to be updated. Here the advantage of the virtual addressing scheme becomes obvious. In that case the only reference to the physical memory address is in the corresponding entry of the *Handle Table*. If the objects were addressed directly with physical addresses, every reference to that object has to be updated.

5.2. Java Bytecode

Unlike other programming languages, Java is not compiled to machine code but to Java bytecode which is typically run on a virtual stack machine. For each Java class a **.class* file containing the Java bytecode is created. For many bytecodes the naming follows a simple scheme: {data type, operation Name}. Data type can for example be `I` (for Integer) or `F` (for Float). Operation names are are for example `ADD` or `LOAD`. This results for example in the bytecodes `IADD`, `ILOAD`, `FADD` or `FLOAD`. Bytecodes are 8-bit wide and can have a variable number of 8-bit parameters, which are determined during compile time and stored in the instruction code directly behind the bytecode. The bytecodes can coarsely be categorized into four groups:

- **Method calls and jumps**
  Examples:
  - `INVOKEVIRTUAL` - Pops the method parameters from the *Stack Memory*, saves *Local Variable Memory* and *Stack Memory* and enters a new object method.
  - `IRETURN` - Ends the current method, restores *Local Variable Memory* and *Stack Memory* and returns an integer value by pushing it onto the restored *Stack Memory*.
  - `GOTO<param>` - Calculates the new address in the instruction memory from `param` and jumps to that address.
- **Stack Memory operations** (mostly arithmetic operations)
  Examples:
  - `IADD` - Pops two integer values from the stack, calculates the sum and pushes the result onto the *Stack Memory*.
  - `FDIV` - Pops two float values from the stack, calculates the quotient and pushes the result onto the *Stack Memory*.
- ***Local Variable Memory* operations**
  Examples:
  - `ILOAD<param>` - Loads an integer value from the *Local Variable Memory* at index = `param` and pushes it on the *Stack Memory*. Note that there are special bytecodes like `ILOAD_0` without parameter for local variables with lower indices as they are used more often (see Table 5.1).
  - `FSTORE<param>` - Pops a float value from the *Stack Memory* and stores it in the *Local Variable Memory* at index = `param`.
- **Heap Memory operations**
  Examples:
  - `IALOAD` - Pops a handle and an offset from the *Stack Memory*, loads the desired value from *Heap Memory* and pushes it on the *Stack Memory*.

Code Example 2: Minimal code example

```
a = b + c;
```

Table 5.1.: Bytecode of Code Example 2

Address	Bytecode
0	`ILOAD_1`
1	`ILOAD_2`
2	`IADD`
3	`ISTORE_0`

- `FASTORE` - Pops a float value, a handle and an offset from the *Stack Memory* and stores the value at the desired position in the *Heap Memory*.
- `GETFIELD<param>` - Pops a handle from the *Stack Memory*, calculates the offset from `param`, loads the desired value from *Heap Memory* and pushes it on the *Stack Memory*.
- `GETSTATIC<param>` - Calculates both handle and offset from `param`, loads the desired value from *Heap Memory* and pushes it on the *Stack Memory*.

Table 5.1 shows the bytecode of Code Example 2. The bytecodes 0 and 1 load the local variables `b` and `c` from the *Local Variable Memory* with the indices 1 and 2. Both values are pushed on the *Stack Memory*. Bytecode 2 pops both values from the stack, adds them and pushes the result on the stack. The last Bytecode again pops this value from the stack and stores it in the local variable `a` (index 0).

5.2.1. Heap Memory Operations

Heap Memory operations can also be divided into three groups as shown in Table 5.2.

Table 5.2.: *Heap Memory* operation types

Type		Handle	Offset
Array Accesses:	`IALOAD`, `IASTORE`, ...	Popped from *Stack Memory* during runtime	Popped from *Stack Memory* during runtime
Object field Accesses:	`GETFIELD`, `PUTFIELD`	Popped from *Stack Memory* during runtime	Read from the bytecode parameter
Static field Accesses:	`GETSTATIC`, `PUTSTATIC`	Read from the bytecode parameter	Read from the Bytecode parameter

Green cells show that the corresponding information can be obtained during compile time while yellow cells show that this information is only available during runtime. Thus, these three *Heap Memory* operations have to be handled differently when dependencies between *Heap Memory* accesses are calculated. More details are given in Section 10.2.

Also, it has to be noted that arrays, object fields and static fields are always stored in distinct memory regions.

Listing 5.2: Virtual method example

```
 1 class Parent{
 2         public int valueA(){
 3                 return 314;
 4         }
 5 
 6 
 7         public static void main(String[] args){
 8 
 9                 Parent p;
10                 if(args.length > 0)
11                         p = new Parent();
12                 else
13                         p = new Child();
14                 System.out.println(p.valueA());
15         }
16 
17 }
18 
19 class Child extends Parent{
20         public int valueA(){
21                 return 42;
22         }
23 }
```

5.3. Java Method calls

In Java methods can be virtual or non-virtual. Virtual methods are interface methods or public and protected object methods, which can be overwritten by subclasses. Listing 5.2 shows that the class of the object `p` in line 14 can be either `Parent` or `Child`. Thus, it is not known during compile time which version of the method `valueA()` is actually executed. Only during runtime when the class of `p` is known, the correct method can be determined.

Constructors, static methods and private methods are non-virtual methods. The exact method is known during compile time because those method calls are always linked to one specific class.

Before a method is called, all parameters of this method are pushed onto the stack. Then they are transferred to the *Local Variable Memory* of the called method. A new empty *Stack Memory* will be initialized. Both the *Local Variable Memory* and the *Stack Memory* of the calling method have to be saved. When the called method returns to the calling method both memories will be restored. The return value of the called method will now be on top of the restored stack.

6. AMIDAR Processor

AMIDAR stands for *Adaptive Microinstruction Driven Architecture* and describes a reconfigurable class of processors [20]. AMIDAR processors consist of a set of FUs *Functional Unit* that operate independently and potentially in parallel. FUs can for example be an ALU, *Thread Scheduler* or the *Object Heap* as shown in Figure 6.1. The Microinstructions for the FUs are called tokens. The core of an AMIDAR processor is the *Token Machine*, which is also an FU. The *Token Machine* loads the machine instructions from the instruction memory and translates each instruction into the tokens for all FUs. Tokens are sent via the Token Distribution Network (TDN) and data is exchanged via the data bus. Both data and tokens are annotated with tags to make sure the correct data is used for each instruction. Optimized token sets for kernels can be generated during runtime to increase the performance [26].

In this work the AMIDAR-Java Processor described in [40] will be used as a host processor for the CGRA based accelerator. In this AMIDAR version the *Stack Memory* and the *Local Variable Memory* are combined into a single FU called *Framestack*. The *Heap Memory* is realized by the FU *Object Heap* (further details are given in Section 6.2).

The following sections give a short overview of this processor.

6.1. Basic Principle

Java bytecodes will be translated into tokens for each FU by the *Token Machine* with the help of a lookup table[1].

Tokens can contain a destination FU and a destination port when the operation produces an output. Ports are needed to maintain the operand order. Additionally, each token and each data that is transferred on the bus is associated with a tag.

[1] Bytecodes and tokens have similar names in many cases. In order to be able to distinguish between both, Bytecodes will be written in typewriter (`ILOAD`) and tokens will be written in italics (*LOAD*)

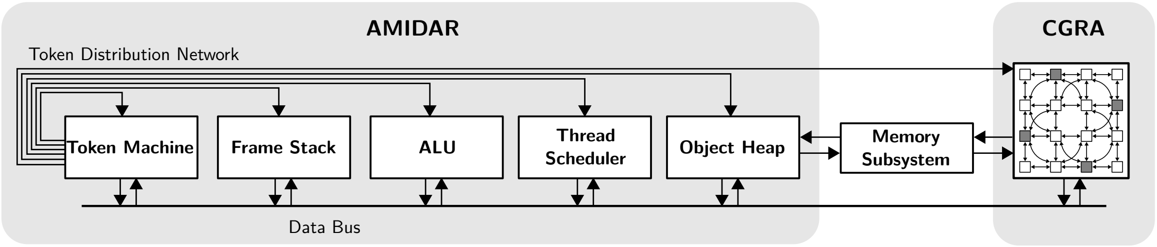


Figure 6.1.: Structure of an AMIDAR Processor

Table 6.1.: Translation from bytecodes to tokens

Bytecode	IALU Token	Destination FU / Port	Framestack Token	Destination FU / Port	*Stack Memory*
`ILOAD_1`	-		*LOAD32<1>*		`b`
`ILOAD_2`	-		*LOAD32<2>*		`b,c`
`IADD`			*POP32*	→ IALU / 1	`b`
			POP32	→ IALU / 0	*empty*
	ADD	→ Framestack / 0	*PUSH32*		`b+c`
`ISTORE_0`	-		*STORE32<0>*		*empty*

Each FU maintains a token queue and executes the tokens in the incoming order. When an FU starts executing a token, it has to wait for the operands to be sent over the data bus. It will only accept data packets with a matching tag. When all operands are available the actual execution is started and the result is sent to the destination port in the destination FU via the data bus.

The *Token Machine* will only stop decoding bytecodes and sending tokens to wait for a branch decision or when any token queue of an FU is full. The FUs synchronize only via the transferred data. Thus, FUs can easily be exchanged without having to adapt any other part of the processor. This makes the AMIDAR processor a good choice to be coupled with a CGRA-based accelerator.

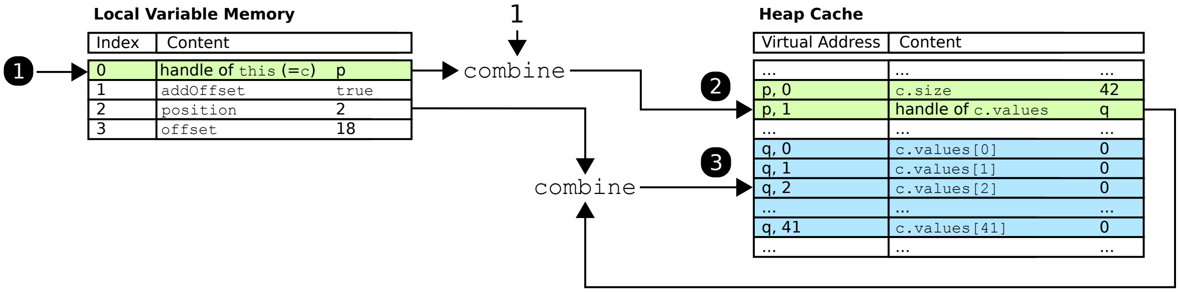
Table 6.1 shows the tokens that are generated for the bytecode sequence shown in Table 5.1. The token *LOAD32<param>* loads a 32 bit local variable from the *Local Variable Memory* and pushes it on the stack. *POP32* pops a 32 bit value from the stack and sends it to the desired port in the destination FU. *ADD* awaits two inputs and adds them. The result is also sent to the desired port in the destination FU. *STORE32<param>* pops a value from the stack and stores it to the *Local Variable Memory.*

6.2. Functional Units

This section will briefly discuss the FUs that are relevant for this work. The **Token Machine** contains the instruction memory cache and the token generation logic. This functionality was split into two FUs called Instruction Memory and Token Generator in early publications of the AMIDAR concept. The *Token Machine* also sends tokens to itself to execute jumps, branches and method invocations.

The **Object Heap** handles all operations concerning the *Heap Memory.* Those operations include for example memory object accesses, memory allocation when a constructor is called or to find out the class of a memory object. Accesses to the *Heap Memory* are realized via a virtually addressed cache as mentioned in Section 5.1.3. Figure 6.2 shows the steps that have to be executed (assuming that all necessary data resides already in the cache):

1. Load handle of `c` from *Local Variable Memory* index = 0



Local Variable Memory

Index	Content	
0	handle of this (=c)	p
1	addOffset	true
2	position	2
3	offset	18

Heap Cache

Virtual Address	Content	
...	...	...
p, 0	c.size	42
p, 1	handle of c.values	q
...	...	...
q, 0	c.values[0]	0
q, 1	c.values[1]	0
q, 2	c.values[2]	0
...	...	...
q, 41	c.values[41]	0
...	...	...

Figure 6.2.: *Heap Memory* access example with a virtually accessed cache

2. Load the handle of `c.values` from the cache with the virtual address {handle = p, offset = 1}
3. Load `values[position]` from the cache with the virtual address {handle = q and offset = position = 2}

When such a cache is used, the *Handle Table* and the *Heap Memory* only have to be accessed in case of a cache miss. Further details can be found in Chapter 11. Also, if the Garbage Collector moves objects in the heap, not all references to the object have to be updated but only the entry in the *Handle Table*.

Peripheral devices are connected to the *Object Heap* via a Wishbone Bus. Registers in the peripherals are mapped into the heap address space and can be accessed via software with the methods `AmidarSystem.readAddress(int address)` or `AmidarSystem.writeAddress(int address, int value)`. Alternatively, those registers can be accessed via object fields of the corresponding Peripheral object. Those objects are generated by the AMIDAR bootloader during startup and map the object field addresses to the peripheral addresses. It is clear that those registers can not be cached. Some FUs are also connected to the *Object Heap* as peripheral in order to be able to configure that FU or to read status registers.

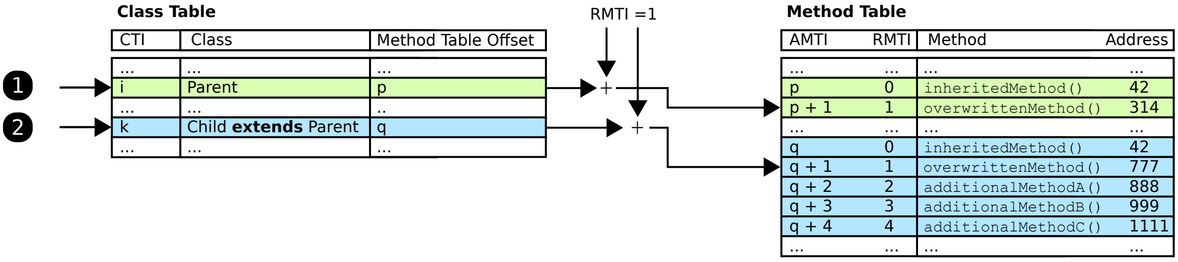
The **Framestack** combines both the *Local Variable Memory* and the *Stack Memory* in one memory. This brings benefits if many methods with many parameters are called. Instead of copying all parameters from the caller stack to the callee *Local Variable Memory*, only pointers have to be adjusted

The AMIDAR processor can contain arbitrary **ALU**s like for example Integer ALU, Floating Point ALU or just a Integer Division FU.

The **Thread Scheudler** manages the fair execution of multiple threads that are executed on AMIDAR in time multiplex. It ensures that interrupts are executed if needed. Multi-threading is an important feature of the AMIDAR processor because the algorithm to map kernels to the CGRA is executed as a separate thread on AMIDAR.

The **CGRA** based accelerator will be included in AMIDAR as an FU with direct access to the memory subsystem as shown in Figure 6.1. Details will be described in the next Chapter.

In this work the AMIDAR **prototype** described in [40] will be used and extended. It consists of the FUs *Token Machine*, *Framestack*, *Thread Scheduler*, *Object Heap*, FPU



Class Table

CTI	Class	Method Table Offset
...	...	...
i	Parent	p
...	...	..
k	Child **extends** Parent	q
...	...	...

Method Table

AMTI	RMTI	Method	Address
...	...	...	...
p	0	`inheritedMethod()`	42
p + 1	1	`overwrittenMethod()`	314
...	...	...	...
q	0	`inheritedMethod()`	42
q + 1	1	`overwrittenMethod()`	777
q + 2	2	`additionalMethodA()`	888
q + 3	3	`additionalMethodB()`	999
q + 4	4	`additionalMethodC()`	1111
...	...	...	...

Figure 6.3.: *Class Table* and *Method Table* in AXT format (simplified)

(floating point unit), long ALU, integer ALU, integer division, integer multiplication, floating point division and a debugger module. It uses a single L1 cache and a crossbar interconnect between the FUs instead of a bus.

6.3. AMIDAR Executable Format

The AMIDAR processor does not execute Java **.class* files directly. Instead they are converted to the AMIDAR Executable Format (AXT). In this format all class files that are needed for an application are combined in one file. Next to the bytecode the AXT file also contains several tables like *Method Table*, a *Class Table* or a common *Constant Pool*. The whole AXT file will be loaded into the *Heap Memory* before the execution is started on AMIDAR.

All dependencies are resolved during compile time if possible. One example are the parameters of non-virtual invoke bytecodes (e.g. `INVOKESTATIC`) in a class file. Each parameter contains a reference to the *Constant Pool*. This *Constant Pool* entry itself references the desired method. In AXT these references are resolved and the absolute index to the *Method Table* (called *Absolute Method table Index* = AMTI) is used as the parameter of that bytecode.

As mentioned before, virtual methods can not be identified exactly during compile time. Instead a relative *Method Table Index* (RMTI) is used as a parameter for virtual object methods. In order to find the correct method during runtime, the class of the object on which the method is called, has to be read from the *Handle Table* by the *Object Heap*. The class of an object is defined by an index to the *Class Table* (called *Class Table Index* = CTI). The corresponding entry of the *Class Table* contains a *Method Table Offset* from which the AMTI can be calculated: AMTI = *Method Table Offset* + RMTI. Figure 6.3 shows that when the method `overwrittenMethod()` is called with `INVOKEVIRTUAL<RMTI=1>` on an object of class `Parent` the method with the AMTI = p + 1. will be executed (Case 1). This method starts at address 314 in the memory. If the same method is called on an object of class `Child`, the *Method Table Offset* is different and the method with the AMTI = q + 1 is called (Case 2). This method starts at the memory address 777.

This procedure requires that the AXT Converter orders the methods in the *Method Table* that overwritten and inherited methods in child classes always have the same RMTI as

Listing 6.1: Loop structure with two backward jumps

```
1 while(a==b){
2         doSomeThing();
3         if(a==c){
4                 doSomeThingDifferent();
5         }
6 }
```

shown in Figure 6.3. Note that in both classes the method `inheritedMethod()` points to the same start address.

For interface methods similar mechanisms exist to find the correct method during runtime. Details can be found in [40]. The calculation of AMTI is done in the *Token Machine*.

6.4. Online Profiler

A profiler based on [18] is implemented in the *Token Machine* to identify all loops in the executed code. The most time consuming loops (kernels) will be mapped to the CGRA. Depending on the compiler the loops are structured differently. In the first variant the loop condition is checked in the beginning of the loop body (used by *javac*). If the condition is not met, the loop is left with a relative forward jump over the loop body. If it is met, the loop body is executed and in the end an unconditional backward jump restarts the execution. In the second variant the loop condition is checked in the end of the loop body (used by *Eclipse Compiler for Java*). If it is not met, the loop is left and no jump is performed an simply the next bytecode is executed. If it is met, a relative backward jump is performed to the beginning of the loop body. Unless a do-while loop is executed, this means the loop has to be entered by an unconditional forward jump over the loop body directly to the loop condition. In both cases loops can be identified by the backward jumps at the end of the loop body. In those simple cases the profiler can easily identify the borders of the loop body which are the target of the backward jump and the backward jump itself.

A more complicated case is shown in Listing 6.1 when compiler variant 1 is used. In this case two backward jumps exists. First, in line 3 a conditional backward jump is executed if the condition evaluates to false. Otherwise the basic block in line 4 is executed and in the end an unconditional backward jump to the beginning of the loop body is executed. In compiler variant 2 similar cases occur if there are combined loop condition like `while(a || b)`. If `a` evaluates to true, `b` doesn't have to be evaluated and a backward jump can be performed directly.

The profiler from [18] was implemented and extended to be able to handle those cases in both compiler variants during this work [73]. In [18] a content addressable memory (CAM) was proposed for each method to store the loop profiles. On each method call the old CAM would have to be saved and restored after the called method returned. In this work [73] loop profiles are stored in a global hash map which uses the start address of the loop as key. The keys are mapped to 512 buckets with 4 slots each. If more than four keys map to the same bucket, no correct profiles will be generated but correct program execution is guaranteed. During this work that case never occurred.

7. CGRA Architecture

As mentioned above, the CGRA based accelerator will be included into the AMIDAR processor as an FU. The structure of this FU is shown in Figure 7.1. The FU contains a CGRA core which executes the actual computation. The CGRA Frame realizes the communication with the host processor and has to be adapted if the CGRA is connected to another processor. The CGRA core is independent of the host processor.

This chapter only describes the core of the CGRA. In Chapter 8 the communication between the CGRA and the rest of the AMIDAR processor is explained.

Figure 7.2 shows an Overview over the CGRA core. The four main components are the Array of *Processing Elements*, *Context Control Unit*, *Condition Box* and the context memories (gray). The CGRA is parametrized and the number of PEs, the interconnect, the operations of each PE, etc can be defined by the user before runtime. The following sections describe all components in detail.

7.1. Processing Element Array

The Processing Element (PE) Array executes the actual computation on the CGRA. Figure 7.3 shows the basic structure of a single PE. The ALU can take inputs from the local register file directly or from neighboring register files via the inputs *in*. ALU results are always written back to the local register file in order to keep the critical path short. In the next clock cycle the value can be transferred to a neighboring PE via the signal *out*.

7.1.1. Local Variable Interface

In order to exchange *Live-In* and *Live-Out* Variables between the CGRA and the host processor, the basic structure of the PE has to be extended as shown in Figure 7.4. Thus, *Live-In* Variables can be written directly to the local register file. In contrast to that *Live-Out* Variables can be read both from the local and the neighboring register files.

Local variables will be transferred via the data bus shown in Figure 6.1. All *Live-In* connections can be connected directly to the bus and need no further management.The current context controls whether a PE stores the incoming value. *Live-Out* connections have to be handled differently as only one PE may drive the bus at one time. Thus, at least one multiplexer is necessary.

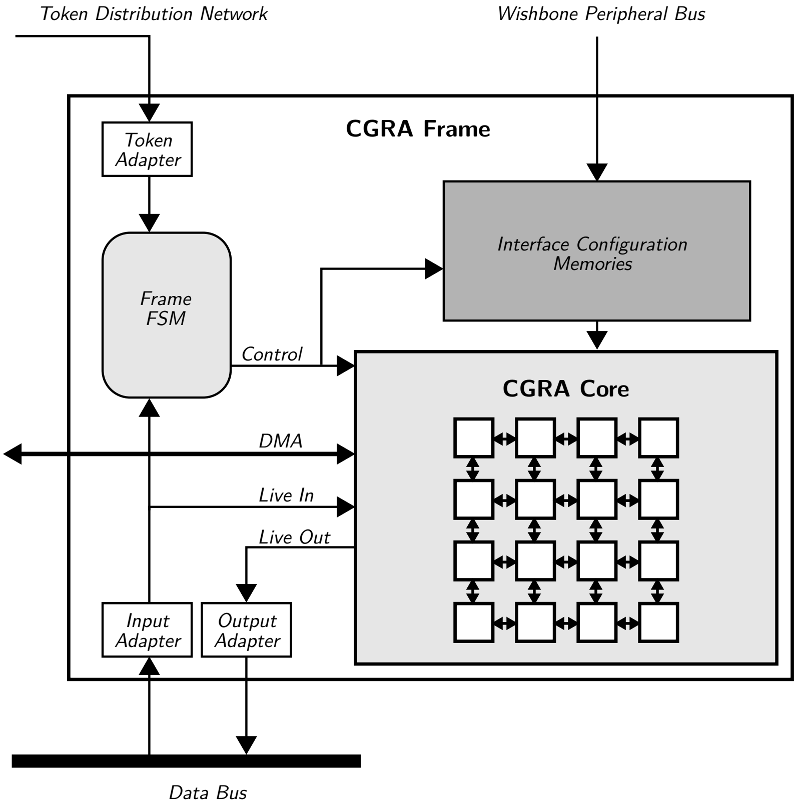


Figure 7.1.: Structure of the CGRA FU

7.1.2. Memory Interface

In order to grant a PE access to the main memory the structure of a PE has to be extended further as shown in Figure 7.5. It has to be noted that in this work two addresses have to be transferred as the memory is addressed indirectly with handle and offset as described before. Similarly to the *Live-In/Out* connections, the data read from the memory is written to the register file while the data to write can be loaded both from the local and the neighboring register files. The handle can only be loaded from the local register file whereas the offset can also be loaded from neighboring register files. This setup makes it possible to calculate the new offset and possibly new data in parallel on another PE while the current memory access is still executed. Afterwards, the next memory access can be started directly without the need to transfer data into the local register file.

In case of a cache miss the data is not available directly and the execution of the whole CGRA has to be stalled. A status input from the cache denotes whether the cache input data is valid.

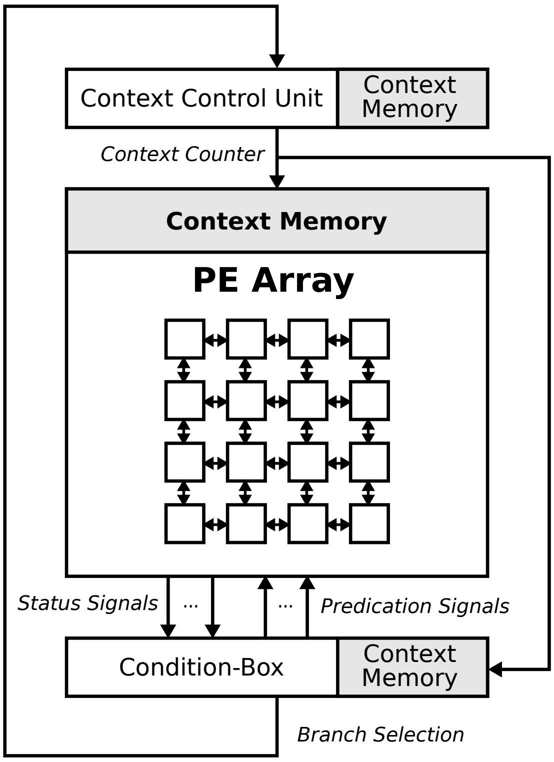


Figure 7.2.: Structure of the CGRA core

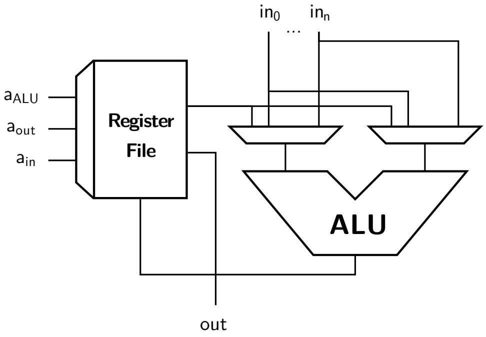


Figure 7.3.: Basic structure of a PE

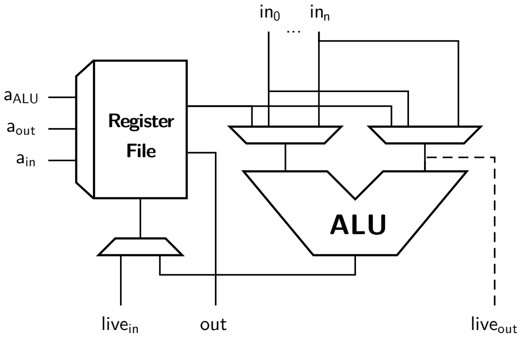


Figure 7.4.: Structure of a PE with *Live-In* and *Live-Out* connections

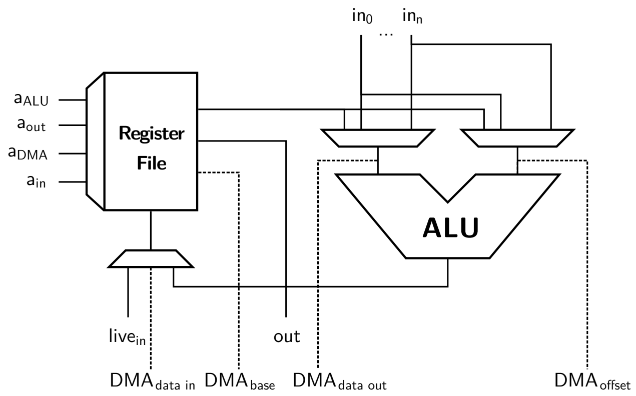


Figure 7.5.: Structure of a PE with *Live-In*, *Live-Out* and memory connections

ALU Opcode	address$_{ALU}$	address$_{in}$	address$_{out}$	MUX$_0$select	MUX$_1$select	RF write enable

Figure 7.6.: Context of a PE shown in Figure 7.3

7.2. Context Memories

From the previous section it is clear that each PE needs to be provided with control signals like for example the register file addresses, the ALU opcode or the multiplexer selection at the ALU inputs. The set of all those control signals is called context and defines the operation of the PE in one time step. Figure 7.6 shows the context of the PE in Figure 7.3. The actual bitwidth is dependent on the composition of the CGRA. One Kernel normally consists of several contexts which are stored in the context memories. Both Context Control Unit (CCU) and Condition-Box (C-Box) also have context memories. All context memories are always addressed with the same address called the *context counter* which is equivalent to the program counter in traditional CPUs.

The last context in the context memory is called the *Idle-Context*. In this context no PE performs an operation and the context counter is not altered as long no kernel is started on the CGRA. The *Idle-Context* is also used to handle the transfers of *Live-In* and *Live-Out* Variables between CGRA and host processor. To store *Live-In* values the *Idle-Context* is modified so that the $live_{in}$ value in Figure 7.4 will be used as register file input. The input a_{in} will be set to the desired address and the write enable of the register file will be set to true. To read *Live-Out* values, the *Idle-Context* will be modified in a similar way.

7.3. Context Control Unit

The CCU calculates the next context counter in every step. In normal mode the context counter is incremented by one each time step. Jumps can be relative or absolute and conditional or unconditional. In the beginning of a loop structure the loop condition is evaluated. If the condition is not met, an unconditional relative jump over the

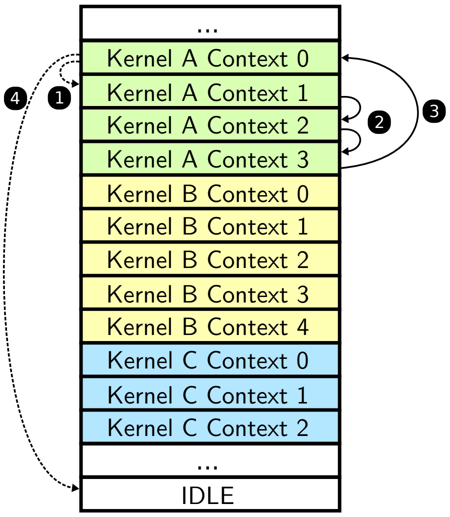


Figure 7.7.: Context memory example

contexts of the loop body is performed to exit the loop. Otherwise the context counter is incremented one by one to execute the loop body. This is shown in Figure 7.7 points 1 and 2. At the end of the loop a relative unconditional jump is performed to the beginning of the loop (point 3). When leaving the outermost loop, an absolute jump to the *Idle-Context* is performed (point 4).

The decision how to calculate the next context counter is stored in the context memory of the CCU.

As mentioned above, the execution has to be stalled when a cache miss occurs. When at least one of the caches denotes that the input data is not valid, the CCU will stop updating the context counter until the data is valid.

7.4. Condition Box

The C-Box evaluates and stores status signals produced by the PEs. Figure 7.8 shows the structure of the C-Box. It consists of an arbitrary number of *Evaluation Blocks* and one common *Condition Memory*. With the *Evaluation Blocks* it is possible to combine status signals arbitrarily to complex boolean expressions. The result can be sent to the CCU in order to decide whether a jump has to be performed or it is stored in the *Condition Memory*. Values from the *Condition Memory* can be sent back to PEs as predication signals (see Section 9.3 for further details).

Code Example 3 shows a case where the if-path (line 4) is only executed when the logic expression $a(b + c)$ holds true. The else-path (line 6) is executed when $a\overline{(b + c)} = a\bar{b}\bar{c}$ holds true. From this example it becomes obvious that both the condition for the if and the else path have to be stored in the *Condition Memory*. Both conditions can be false at the same time if a is false. Thus, the condition of the if-path is not always the negation of the else path. In the following it will be explained how the C-Box evaluates these expressions.

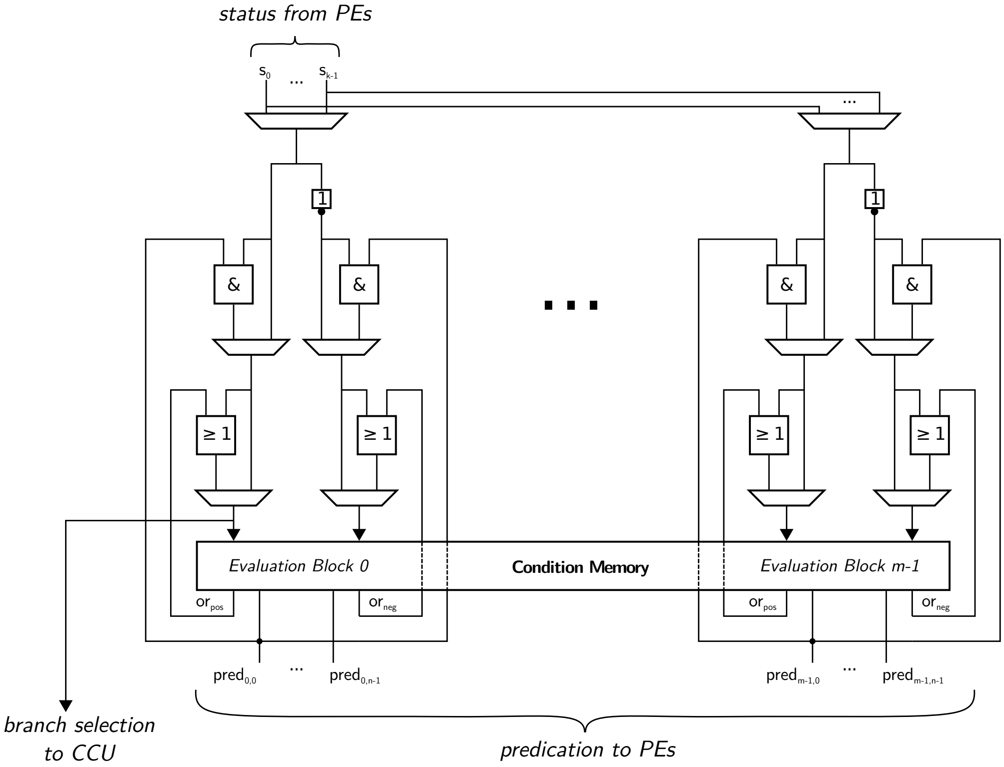
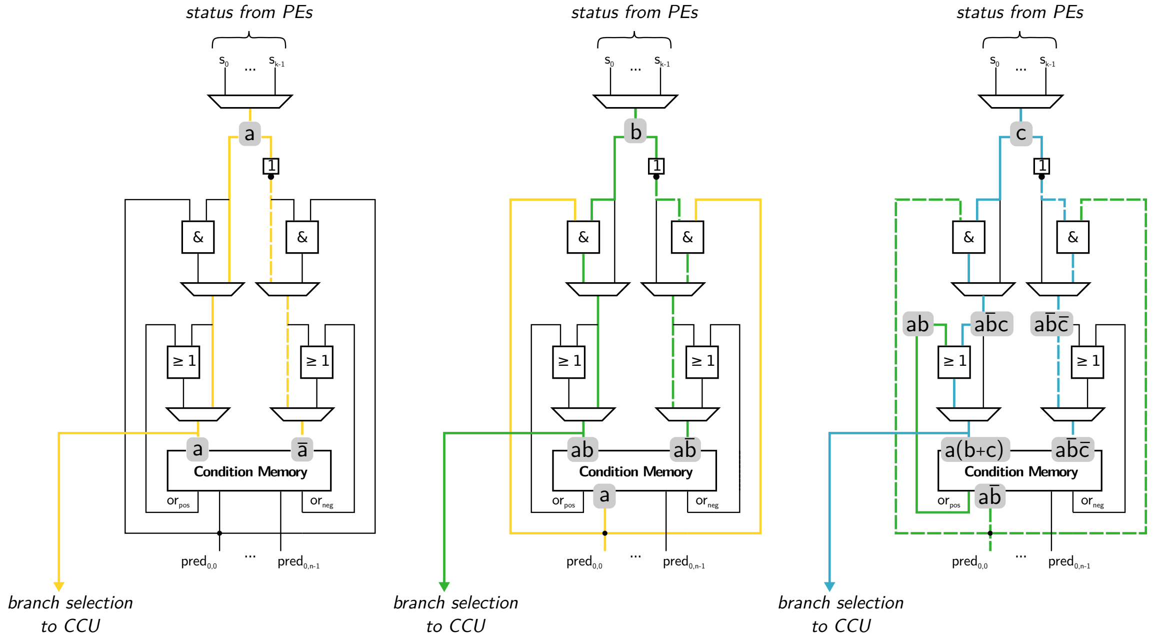


Figure 7.8.: Structure of the Condition Box

When the code is executed, the expression a in line 1 is evaluated by a PE and the result is sent as status to the C-Box. The logic value of a is stored directly in the *Condition Memory* as shown in Figure 7.9(a). The logic gates are all bypassed (yellow line). In the second step the expression b is evaluated as shown in Figure 7.9(b). The lines 2 to 6 in Code Example 3 are only executed if a was true. So a is loaded from the *Condition Memory* (yellow line) and the AND gates are not bypassed (green lines) so that the logic values ab and $a\bar{b}$ are stored in the *Condition Memory.* In the third step c is evaluated and the values ab (green drawn through line) and $a\bar{b}$ (green dashed line) are loaded from the *Condition Memory.* The AND gates are not bypassed and the values $a\bar{b}c$ and $a\bar{b}\bar{c}$ are used further. In the if-branch the OR gate is not bypassed (blue drawn through line) and the value $ab + a\bar{b}c = a(b + \bar{b}c) = a(b + c)$ is stored in the *Condition Memory.* The OR gate in the else-branch is bypassed and the value $a\bar{b}\bar{c}$ is stored into the *Condition Memory* directly (blue dashed line). Now the desired values



(a) Evaluate `a` (b) Evaluate `b` (c) Evaluate `c`

Figure 7.9.: C-Box evaluation steps for Code Example 3

reside in the *Condition Memory* and can be provided to the PEs as predication.

Code Example 3: C-Box example code

```
if $a$ then
    // do something
    if $b$ || $c$ then
        // If path: Executed when $a(b + c)$
    else
        // Else path: Executed when $a\overline{(b + c)} = a\bar{b}\bar{c}$
```

7.5. Important Features of the CGRA

The actual Verilog description of a CGRA instance can be generated fully automatically from a textual description in JSON format (shown in Appendix B). The information that needs to be provided is:

- The number of PEs
- The operations each PE can perform
- The connections between the PEs
- The number of *Evaluation Blocks* in the C-Box

- The number of predication signals from the C-Box to the PEs
- The sizes of the register files, *Condition Memory* and the context memories

Both the CGRA generator and the scheduler that maps a kernel to the CGRA support inhomogeneous operation distribution on the PEs and irregular interconnects between the PEs. Thus, is it is possible to create and evaluate different CGRA structures quickly and adapt them to the current application domain if desired. This is not scope of this work.

The C-Box enables the CGRA to evaluate control flow on the CGRA. Thus, it is possible to execute complex data dependent nested loop structures directly on the CGRA autonomously. No interaction with the host processor is necessary.

8. AMIDAR CGRA Interface

Note: Parts of this section have already been published in [30]. The marking of self-citation is omitted in order to improve the reading flow.

Before the execution on the CGRA can start, the *Live-In* values and an ID of the current kernel have to be sent to the CGRA. When the execution is finished, the *Live-Out* values have to be transferred back. Many approaches use dedicated registers to send data into the CGRA or back to the host processor (e.g. [71]). In this work a more flexible interface between the AMIDAR processor and the CGRA is implemented, in order to be able to support arbitrary applications.

In this chapter the interface is described. Figure 7.1 shows the CGRA Frame that handles the communication between AMIDAR and the CGRA. The *Token Adapter* stores incoming tokens in a token queue and the FSM handles their execution. *Input* and *Output Adapter* handle the data transfers on the data bus and check for matching tags. The CGRA Frame is not involved in the DMA communication.

For each kernel different constants and different *Live-In/Out* Variables have to be transferred and stored in different locations. All necessary information is stored in the *Interface Configuration Memories* which are described in the next section. Afterwards, the bytecodes that handle the communication between AMIDAR and the CGRA are described. In the end of this chapter different *Live-In/Out* strategies are discussed.

8.1. Interface Configuration Memories

In order to ensure that a kernel is executed correctly on the CGRA the following information is needed:

- Which kernel will be executed?
- Which constants have to be stored in the PEs?
- Which local variables have to be transferred to the CGRA (*Live-In*)?
- In which PE and at which register file address will the constants and the *Live-In* variables be stored (*Location Information*)?
- Which local variables have to be sent back to the AMIDAR processor (*Live-Out*)?
- Where are the *Live-Out* variables stored (*Location Information*)?
- Where are the contexts of the kernel?

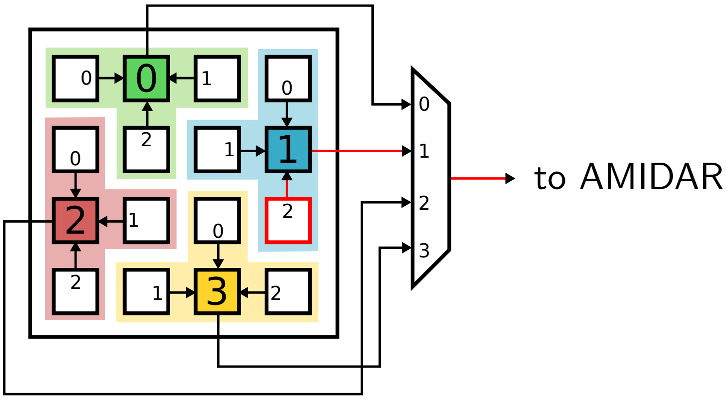


Figure 8.1.: *Live-Out Location Information*

This information will be provided by a a *Constant Memory* and a *Location Information Memory* in the CGRA and a Live-In/Out *Information Memory* in the *Token Machine*. Apart from that both CGRA and *Token Machine* contain a *Kernel Table* that stores the pointers to all those memories for each kernel. All those memories can be accessed via software over the Wishbone peripheral bus. In the remainder of this section those *Interface Configuration Memories* will be described.

Location Information Memory This memory is located in the CGRA and stores both where received values (constants or *Live-In*) will be stored and from where the *Live-Out* values will be read. The *Location Information* for the received values is stored in the format {*PE selection*, *RF address*}. Each bit of *PE selection* corresponds to one PE. If a bit is set, it means that the PE has to store the incoming value in the local register file at address *RF address*. Note, that the *RF address* is the same for all PEs. This has to be taken into account in the register file allocation during in the kernel mapping algorithm (Chapter 9).

As described in Section 7.1.1 the *Live-Out* values can be read both from the local register file and also the register file of the neighboring PEs (see also Figure7.4). Thus, not every PE has to be provided with a *Live-Out* Connection and the *Location Information* for values to be sent is {*Live-Out selection*, *neighbor selection*, *RF address*}. Figure 8.1 shows an example in which a value has to be read from the last PE in the third row (red). Here the *Live-Out selection* is 1 (blue) and the *neighbor selection* is 2.

The bit width of this memory is strongly dependent on the CGRA composition, as the number of neighbors, *Live-Out* connections, register file depth, etc. vary.

Constant Memory As shown in Figure 7.3, all operands in a PE have to be provided by the register file. Thus, the register file also has to contain all constant values that are needed during the execution of a kernel. Before the executions starts, those constants will be read from the *Constant Memory* and written to the register file using the *Location Information*. The *Constant Memory* is also located in the CGRA.

In order to keep the overhead low, small constants will not be read from the *Constant Memory* but they will be coded into the current context and will be directly available. An additional bit is inserted into the context (see Figure 7.6) to the MSB at each register file address denoting that the current register file address is actually no address but a constant stored in two's complement. In that case the register file will not load a value. Instead it will do sign extension and provide that constant value directly. This can only be done for values that have an absolute value smaller than half the register file size so that the two's complement has the same bit width as the register file address. In practice this is sufficient, as the most common constants are small values like ±1 for increments or decrements.

Kernel Table in the CGRA The *Kernel Table* in the CGRA stores the relevant information for all kernels. It contains the pointer to the context memory, the pointer to the *Location Information Memory*, the pointer to the *Constant Memory* and the number of constants that have to be read from the *Constant Memory*.

Live-In/Out Information Memory The *Live-In/Out* Information Memory contains the IDs of all Local Variables that have to be transferred to the CGRA and will be sent back from the CGRA. This ID is used to address the *Framestack*.

Kernel Table in the Token Machine The *Kernel Table* in the *Token Machine* stores the pointer to the *Live-In/Out* Information memory and a valid flag for each kernel. If the kernel is valid, the contexts of the corresponding kernel are currently stored in the context memories of the CGRA and it can be executed directly.

8.2. CGRA Bytecodes

The following three special bytecodes were introduced to start CGRA execution:

- `CHECK_KERNEL <kernel ID>` - Checks whether the contexts of this kernel currently reside in the context memories of the CGRA using the valid flag in the *Kernel Table* in the *Token Machine*. If not, an interrupt is issued and the interrupt service routine loads the corresponding contexts.
- `CGRA_START <kernel ID, nrOfLiveInVariables>` - This bytecode initializes both the CGRA and the *Token Machine* and starts the execution as shown in Figure 8.2. In Step 1, both FUs get the kernel ID as input and load the needed pointers from the *Kernel Tables*. Then, the CGRA starts immediately to load the specified number k of constants from the *Constant Memory* (Step 2) and forwards them with the *Location Information* to the PEs.

  Afterwards, n tokens are sent to the *Token Machine* and the *Framestack* to load the *Live-In* variables (Step 3). At the same time, n tokens are sent to the CGRA to load the *Location Information* for the incoming n *Live-In* variables (Step 4).

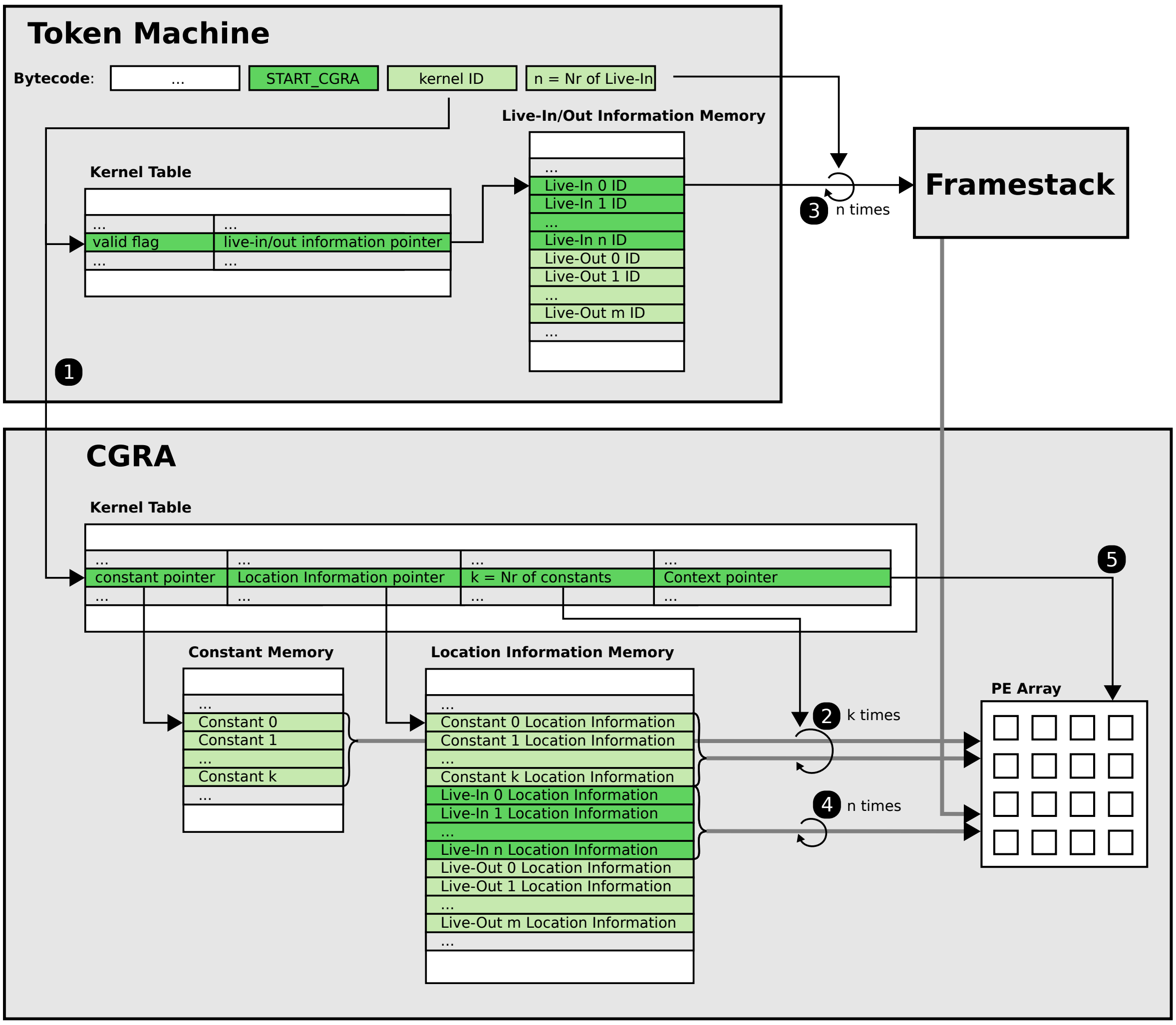

Token Machine
Bytecode:
...
START_CGRA
kernel ID
n = Nr of Live-In
Live-In/Out Information Memory
Kernel Table
valid flag
live-in/out information pointer
Live-In 0 ID
Live-In 1 ID
Live-In n ID
Live-Out 0 ID
Live-Out 1 ID
Live-Out m ID
1
3
n times
Framestack
CGRA
Kernel Table
constant pointer
Location Information pointer
k = Nr of constants
Context pointer
5
Constant Memory
Constant 0
Constant 1
Constant k
Location Information Memory
Constant 0 Location Information
Constant 1 Location Information
Constant k Location Information
Live-In 0 Location Information
Live-In 1 Location Information
Live-In n Location Information
Live-Out 0 Location Information
Live-Out 1 Location Information
Live-Out m Location Information
2
k times
4
n times
PE Array

Figure 8.2.: Execution of bytecode START_CGRA

When this part is finished, the CGRA is started with the correct context pointer (Step 5).

- CGRA_STOP <nrOfLiveOutVariables, jumpDistance> - This bytecode is translated into tokens to transfer *Live-Out* Variables back to the *Framestack*. This is done analogously to the *Live-In* values. Afterwards, a jump is performed in AMIDAR in order to skip the bytecode of the kernel that has just been executed on the CGRA.

Note that the bytecodes CHECK_KERNEL and CGRA_START can not be merged into one bytecode because AMIDAR needs to be able to invoke the interrupt service routine to load missing contexts. The *Thread Scheduler* is only able to do this, when all issued tokens are executed. All tokens of one bytecode are sent at once. Thus, the bytecode for checking the valid flag and the initialization have to be separated, so that the interrupt service routine can be called in between. Additionally, this separation makes it possible to execute the check of the valid flag some time before the actual execution in oder to mask transfer times of the needed contexts in case they are not yet in the CGRA. This option was not used during this work.

Table 8.1.: CGRA tokens

Token	Function	Input	Output
INIT	Initializes the CGRA and loads constants into the CGRA	Kernel ID	-
RECEIVELOCALVAR	Stores a *Live-In* value in the CGRA Core	*Live-In* Value	-
RUN	Starts the kernel execution on the CGRA	-	-
SENDLOCALVAR	Sends a *Live-Out* value over the data Bus	-	*Live-Out* Value

Table 8.2.: Token Machine tokens (excerpt)

Token	Function	Input	Output
CHECK_KERNEL	Checks whether the contexts of the desired kernel are in the contexts memories. If not an interrupt is issued	Kernel ID	-
INIT_LIVE_IN_OUT	Loads the *Live-In/Out* Pointer from the Kernel Table	Kernel ID	Kernel ID
LOAD_LIVE_IN_OUT	Load the Index of the *Live-In/Out* Variable and increments the *Live-In* / Out pointer	-	*Live-In/Out* Index

8.2.1. Translation Into Tokens

The CGRA FU can handle the tokens shown in Table 8.1 and Table 8.2 shows an excerpt of tokens of the *Token Machine* that are needed for the communication with the CGRA.

Table 8.3 shows how the bytecodes described in the previous section are translated into Tokens.

8.3. Live-In/Out Strategies

Different strategies to provide *Live-In/Out* values to the CGRA can be applied. These range from flexible (*Live-In/Out* to each PE) to resource saving (just a single PE with *Live-In* and *Live-Out* connections). The following sections will describe the different strategies in detail.

Table 8.3.: Translation from CGRA bytecodes to tokens

		Token Machine	Destination	Framestack	Destination	CGRA	Destination
Bytecode		Token	FU / Port	Token	FU / Port	Token	FU / Port
`CHECK_KERNEL<ID>`	-	*CHECK_KERNEL<ID>*		-		-	
`CGRA_START<ID,N>`	-	*INIT_LIVE_IN_OUT<ID>*	→CGRA / 0	-		*INIT<ID>*	
	N	*LOAD_LIVE_IN_OUT*	→Framestack / 0	*LOAD32<I>*	→CGRA / 0	*RECIEVELOCALVAR<V>*	
	-					*RUN*	
`CGRA_STOP<M>`	M	*LOAD_LIVE_IN_OUT*	→Framestack / 0	*STORE32<I>*		*SENDLOCALVAR*	→Framestack / 0

Full Live-In/Out Connections In order to minimize the timing overhead of the transfer of values between processor and CGRA, all PEs will be provided with *Live-In/Out* connections. Thus, it is possible to receive and send values to each PE directly in one step.

In order to be able to drive the bus by the correct PE, a N to 1 multiplexer is needed for a CGRA with N PEs. With the information given in Section 7.1.1 it is obvious that the same performance can also be achieved with less hardware effort.

Full Logic Live-In/Out Connections The aim of this strategy is to access each PE in one step but to use less hardware than in the previous strategy. As transferring values from one PE to the register file of another PE would always take one clock cycle (see Section 7.1.1), each PE needs a separate *Live-In* connection. In contrast to that, in CGRAs with regular mesh structure, approximately every fourth PE needs a *Live-Out* connection, as values of register files from neighboring PEs can be sent directly. Thus, only a $\frac{N}{4}$:1 multiplexer is needed to drive the bus because the multiplexer in front of the ALU input is reused.

Figure 8.3 shows the the full logic *Live-Out* connections for a 4x4 and an 8x8 CGRA. It can be seen that only a fourth of the PEs need a *Live-Out* connection in both cases. Note that for some CGRA structures more than $\frac{N}{4}$ *Live-Out* connections are needed to provide full logic *Live-Out* connections. For example a 5x5 CGRA needs 7 *Live-Out* connections ($\approx \frac{N}{3.6}$) and a 6x6 CGRA needs 10 ($= \frac{N}{3.6}$). CGRAs with toroidal interconnect are not considered as this leads to very low clock frequencies due to long routing paths.

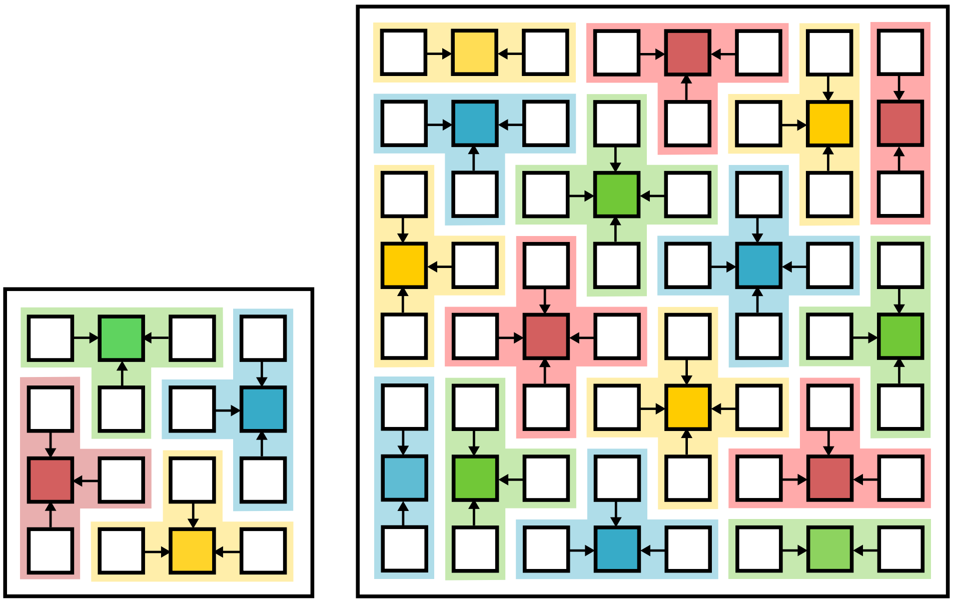


Figure 8.3.: 4x4 and 8x8 CGRA with full logic *Live-Out* connections. Colored PEs have a *Live-Out* connection[30]

It is obvious that full logic *Live-In/Out* connections are always better than naive full *Live-In/Out* connections. Thus, full *Live-In/Out* connections are not considered in the rest of this work.

Reduced Live-In/Out Connections As full logic *Live-In/Out* connections come at the cost of increased hardware and thus possibly longer critical paths, reducing the number of *Live-In/Out* connections has to be considered.

Figure 8.4 and 8.5 show a 4x4 CGRA with two *Live-In* and two *Live-Out* connections respectively. It can be seen that in this case only two PEs can receive values directly. For 7 of the PEs one routing step is needed (1 hop), for 6 PEs two hops are needed. One PE is even reachable only after three hops.

When sending back local variables to the general purpose processor, 9 PEs can provide their values directly while 6 PEs need one hop. Only one PE needs two hops to provide a value to the *Live-Out* connection.

While reducing the number of *Live-In/Out* connections reduces the hardware resources, it also increases the timing overhead when transferring data to the CGRA. Additionally, it increases the memory overhead, because routing information on the CGRA has to be stored. This routing information can either be included as a prologue in the CGRA schedule or it can be encoded in the instructions used to transfer the values.

The usage of dedicated input registers, which is done in many other approaches (see Section 3) is a special case of this strategy.

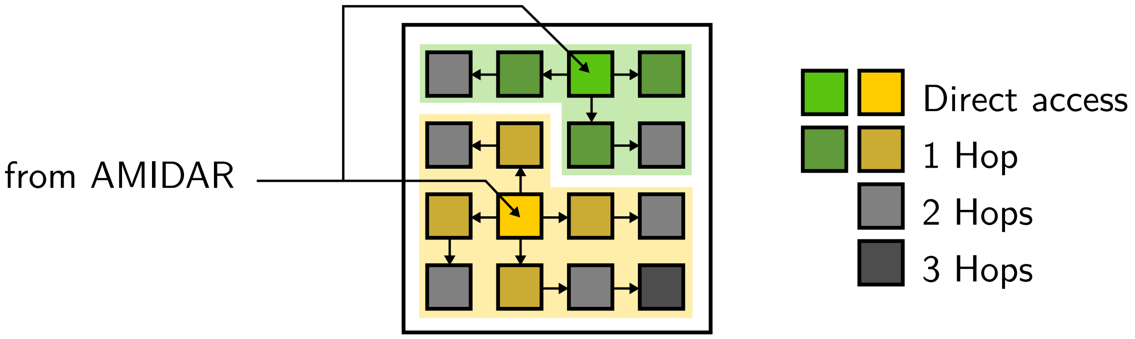


Figure 8.4.: CGRA with two *Live-In* connections [30]

Using Shared Memory instead of Live-In/Out Even CGRAs that don't have *Live-In/Out* connections and communicate with the processor via shared memory are thinkable [34]. As CGRAs need to have DMA anyways, this would reduce the hardware overhead even further. At the same time the timing overhead and the effort to schedule the data transfer would increase as memory accesses on both sides and routing the values through the CGRA have to be handled.

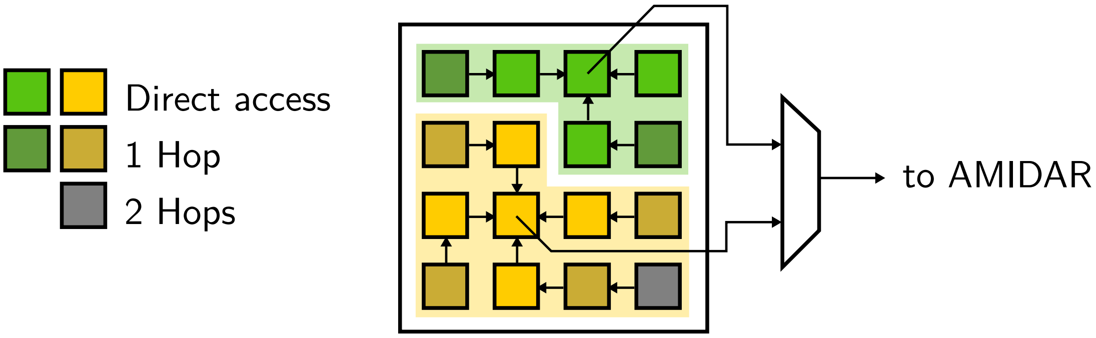


Figure 8.5.: CGRA with two *Live-Out* connections [30]

Comparison All strategies have in common that all values have to be transferred consecutively. As a consequence timing overhead introduced by copying values from one PE to another can be masked by pipelining. In this case, values are received via *Live-In* connection while at the same time another value is copied to a neighboring PE. This increases the complexity of the interface because the following challenges have to be faced. First, the sending order of the values has to be determined (send values that need more hops first, to mask the routing delay, while avoiding routing conflicts). Second, the designer has to find the optimal location to store the routing information. Encoding of the routing information in the CGRA is very costly while storing this information in the host processor introduces additional overhead by transmitting this information to the CGRA. Apparently, there is a trade-off between complexity of the interface and the timing overhead for routing values on the CGRA.

Note that *Live-In* and *Live-Out* connections are independent. Thus, it is possible to use different strategies for *Live-In* and *Live-Out*.

Table 8.4 compares the different strategies. (Full *Live-In/Out* connections are omitted, as full logic *Live-In/Out* connections are always better.) It can be seen that full logic *Live-In/Out* connections only have the disadvantages of increased hardware and thus probably lower clock frequencies.

In [30] it was shown that the hardware effort for full logic *Live-In/Out* connections is negligible and that the maximum clock frequency is mainly unaffected by the *Live-In/Out* strategy for an implementation on an FPGA. Thus, full-logic *Live-In/Out* connections will be used in the rest of this work.

Table 8.4.: Comparison of the *Live-In/Out* connection strategies [30]

	Full Logic Live-In/Out Connections	**Reduced Live-In/Out Connections**	**No Live-In/Out Connections**
Effort to schedule the transfer of *Live-In/Out* values	low	medium (values have to be routed through the CGRA)	high (memory access + values have to be routed through the CGRA)
Time needed for data transfer	low	increased effort due to routing values through the CGRA (can be masked by complex pipelining)	increased effort due to routing values through the CGRA (can be masked by complex pipelining) and memory accesses
Memory overhead	low	memory needed to store routing information	memory needed to store routing information
Amount of additional hardware	medium	low	none
Maximal clock frequency (*estimation*)	medium	high	high

9. Kernel Mapping Algorithm

In this section the kernel mapping algorithm is described. It was partly already discussed in [15], [11], [13] and [12]. The mapping algorithm will be executed on the AMIDAR processor itself as a system thread. Figure 9.1 shows the processing steps that are necessary to map a kernel to the CGRA. First, the profiler in the AMIDAR processor detects kernels in hardware during runtime. The mapping thread reads the profiler memory periodically via Wishbone peripheral bus. The best loop candidate is then selected. If its runtime exceeds a threshold of 30 % of the total runtime in the last period, it will be mapped to the CGRA. First, speculative method inlining and optionally partial loop unrolling will be performed. Then, an instruction graph is created which is used to generate a Control and Data Flow Graph (CDFG). A modified list scheduler will map and bind all operations to the CGRA with respect to the resource and routing constraints. Afterwards, the contexts are generated from the schedule and transferred to the CGRA via Wishbone peripheral bus. In the last step, the kernel tables are filled and the bytecode is patched in the instruction memory.

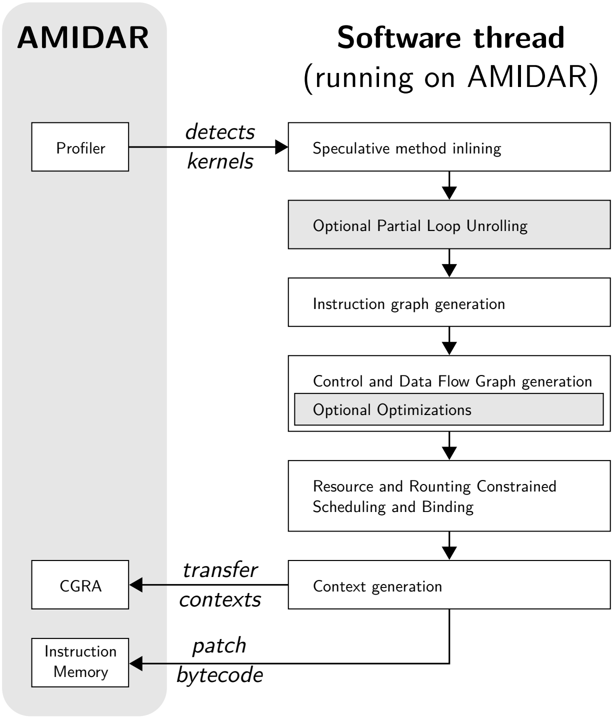

AMIDAR
Software thread
(running on AMIDAR)
Profiler
detects kernels
Speculative method inlining
Optional Partial Loop Unrolling
Instruction graph generation
Control and Data Flow Graph generation
Optional Optimizations
Resource and Rounting Constrained Scheduling and Binding
CGRA
transfer contexts
Context generation
Instruction Memory
patch bytecode

Figure 9.1.: Processing steps of the kernel mapping algorithm

The next sections will explain all steps in detail. Only the optional steps (marked in gray) will be explained in Chapter 10.

9.1. Speculative Method Inlining

Method inlining is done on bytecode level. In the first step the code of the called method has to be retrieved and inserted in the bytecode of the calling method. In the second step the bytecode parameters have to be adapted in order to ensure correct jumps and variable handling.

Retrieve method code In order to read the code of the called method from the instruction memory, the AMTI is needed. With the AMTI the physical address of the kernel code can be read from the *Method Table*.

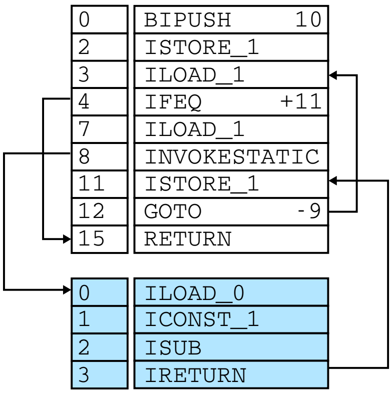
As mentioned in Section 5.3, virtual methods can not be resolved during compile time. When the kernel mapping algorithm is started during runtime, the method has probably been called before on a certain object. It is assumed that when the method is called the next time, the class of the object on which the method is called, will be the same as in the previous method call. Therefore, a ring buffer of size 256 was implemented in the *Token Machine*. It stores for all method invocations the address of the `INVOKE` bytecode, the CTI of the object on which it was called and the AMTI of the method that was invoked last time. Old entries of the ring buffer are overwritten. When a virtual method has to be inlined, the ring buffer will be read out backwards in order to find the last invocation on the current bytecode address to find the AMTI. As only kernels that are executed often are mapped to the CGRA, the method that has to be inlined is likely to have an entry in that ring buffer as it was called very often. If the method is not found in the ring buffer, the kernel can not be mapped to the CGRA.

In order to verify that the speculation about the object class was correct, the CTI is stored in the CGRA as a constant. During runtime the actual CTI of the current object is loaded via DMA from the heap. If both CTIs differ, the assumption was wrong and an interrupt is activated and a rollback mechanism has to be invoked. The instructions to handle this verification are inserted later during CDFG generation. It has to be ensured that these instructions are scheduled before all instructions that potentially change the memory based on a false assumption. When the rollback mechanism is started, the local variables have to be transferred from the CGRA back to the AMIDAR processor. Then the last step in Figure 9.1 has to be reversed and the original bytecode of the kernel has to be restored. Finally the program counter of the AMIDAR processor has to be set to the correct bytecode address depending on the current context counter in the CGRA. Afterwards the AMIDAR processor can resume to the kernel execution at the correct address. Currently, no rollback mechanism is implemented. Instead the execution is simply aborted. This case did not occur in any of the benchmarks used for this work.

In contrast to that, for non-virtual methods no assumption has to be made. Static methods can always be identified during compile time as they are called on classes not on objects. Thus, the AMTI can directly be calculated from the parameter of the `INVOKESTATIC` bytecode. The same holds for private object methods, as they can only be called in the declaring class. Thus, always the method of the declaring class

Code Example 4: Method inlining Example

```
static void main(String[] args)
    int i = 10;
    while (i != 0) do
        i = decrement(i);
static int decrement(value)
    return value − 1;
```



```
0   BIPUSH        10
2   ISTORE_1
3   ILOAD_1
4   IFEQ         +11
7   ILOAD_1
8   INVOKESTATIC
11  ISTORE_1
12  GOTO          -9
15  RETURN

0   ILOAD_0
1   ICONST_1
2   ISUB
3   IRETURN
```

Figure 9.2.: Bytecode of Code Example 4

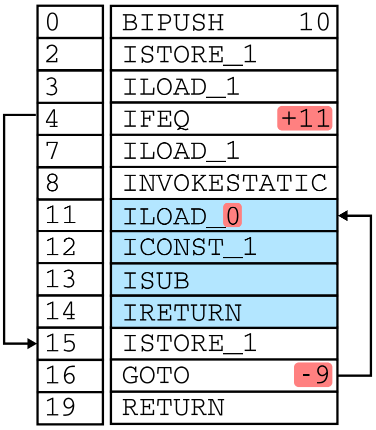
is executed even if the object is an instance of a child class which overwrites that method.

Bytecode Parameter Adaptation The method bytecode parameter adaptation will be explained on the basis of Code Example 4 whose simplified bytecode is shown in Figure 9.2. The bytecode of the main method is shown in white. The bytecodes at address 0 and 2 initialize the variable `i` with the value 10[1]. The bytecodes 3 to 12 realize the while-loop. In the beginning the variable `i` is loaded (3) and then compared to zero (4). If i is equal to zero the loop is exited by jumping 11 bytecodes to address 15. Otherwise `i` is loaded (7) and the static method `decrement` is called with `i` as parameter (8). The result of this method stored in `i`(11) and then a jump 9 bytecodes back to the beginning of the loop is performed.

The bytecode of `decrement` is shown in blue. It loads the variable 0 (`value`) and the constant 1 and then calculates the difference of both values and returns the result.

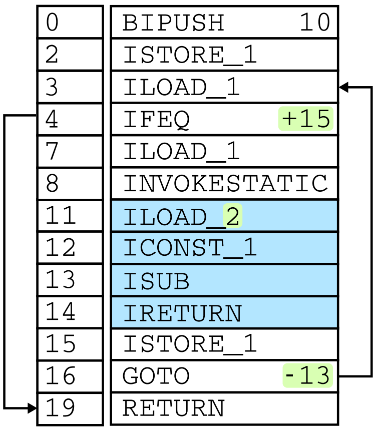
When the bytecode of the called method is inserted in the code, the bytecode addresses of all following bytecodes is changed. Therefore, all relative jumps that jump over the `INVOKE` bytecode, point to the wrong bytecode as shown in Figure 9.3. Additionally, the bytecode 11 should load the variable `value` which had the index 0 in the method `decrement`. When the code is inlined into the main method, the context changes and the local variable with index 0 is `args`. Thus, all indices of local variable accesses in the inlined code need to be incremented by an offset that is greater than all local variable indices used before. These new local variables are called virtual local variables. In the CDFG generation step instructions are inserted for the bytecode `INVOKESTATIC` that copy the value of the parameter `i` in the virtual local variable `value`. Figure 9.4 shows the corrected relative jumps and local variable accesses. After this step is finished, the bytecode of the loop (3 to 16) is used for graph generation.

[1] The parameter of the bytecode `BIPUSH` is stored at address 1



0	BIPUSH	10
2	ISTORE_1	
3	ILOAD_1	
4	IFEQ	+11
7	ILOAD_1	
8	INVOKESTATIC	
11	ILOAD_0	
12	ICONST_1	
13	ISUB	
14	IRETURN	
15	ISTORE_1	
16	GOTO	-9
19	RETURN	

Figure 9.3.: Bytecode of Code Example 4 after code insertion



0	BIPUSH	10
2	ISTORE_1	
3	ILOAD_1	
4	IFEQ	+15
7	ILOAD_1	
8	INVOKESTATIC	
11	ILOAD_2	
12	ICONST_1	
13	ISUB	
14	IRETURN	
15	ISTORE_1	
16	GOTO	-13
19	RETURN	

Figure 9.4.: Bytecode of Code Example 4 after complete inlining

9.2. Instruction Graph Generation

The control and dataflow graph (CDFG) generation was already described in [14]. In the following, instructions in the CDFG will be named after the scheme <bytecode address>:<bytecode>:<parameter>. For example the node corresponding to the bytecode at address 11 in Figure 9.4 will be named `11:ILOAD:2`. The instruction memory is part of the *Heap Memory*, so it can be read with the native method `AmidarSystem.readAddress(int address)`. In order to find control dependencies, an instruction graph as shown in Figure 9.5 is created. Starting from a `START` and a `STOP` node, all nodes are successively added to the instruction graph and connected to their predecessor. Branch nodes like `4:IFEQ:+15` are called *controller* because they decide whether the instructions in the if- and else-branch (marked in green and red) are actually executed. All the nodes in the branches store `4:IFEQ:+15` as their *controller* and whether they are part of the if or else branch. Phi nodes are used to join different branches.

9.3. Control and Dataflow Graph Generation

In the next step all nodes are added to the CDFG and data and control dependencies are added as described in the following.

Data Dependencies In order to find data dependencies, the bytecode of the loop that will be mapped to the CGRA is executed on a virtual stack. In this stack not the results of bytecodes are pushed to or popped from the stack but the corresponding node from the CDFG. When for example the bytecode `13:ISUB` is executed, the nodes `11:ILOAD:2` and `12:CONST:1` are on top of the stack. `13:ISUB` pops both those nodes from the stack. Thus, a data dependency from both `11:ILOAD:2` and `12:CONST:1` to `13:ISUB` are added to the CDFG as shown in Figure 9.6.

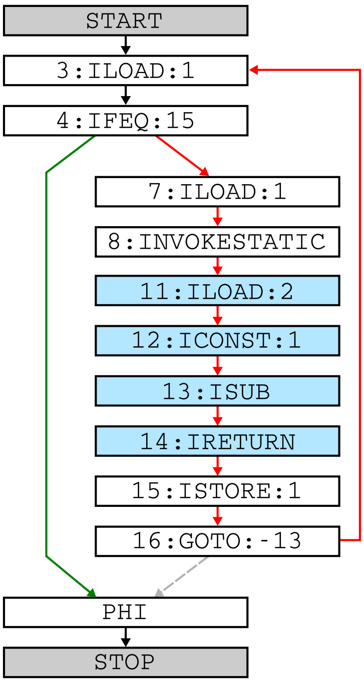


Figure 9.5.: Instruction graph of Code Example 4

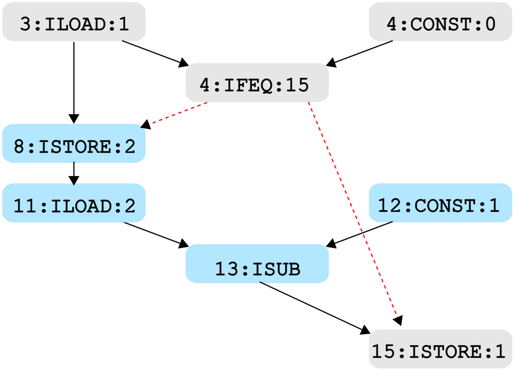


Figure 9.6.: Control and Data Flow Graph of Code Example 4 after complete inlining

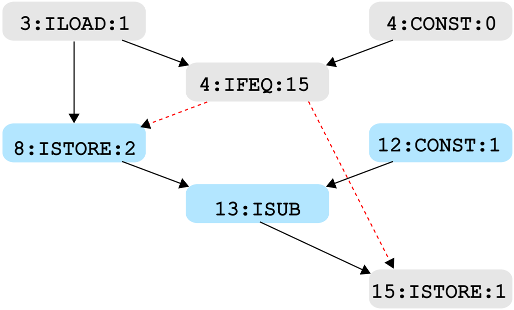


Figure 9.7.: Control and Data Flow Graph of Code Example 4 after complete inlining and optimization

For each local variable and each memory object an access history is maintained in order to find read/write dependencies.

Method Invocations and Returns As mentioned before `INVOKE` bytecodes need special handling. They don't just pop all method parameters from the stack but also create new nodes that store the values of the parameters in the correct virtual local variable of the inlined method. When references of memory objects are passed to the method, the history of the virtual local variable that will hold this reference will store the origin of this reference (this may be another local variable or an access to another memory object) in order to find the correct dependencies between all accesses to the actual memory object. In the current example, the node `8:INVOKESTATIC` creates a node

8:ISTORE:2 which gets its data from 7:ILOAD:1. As both 3:ILOAD:1 and 7:ILOAD:1 access the same local variable and this local variable is not written in between, only one of those nodes is necessary and 8:ISTORE:2 will get its data from 3:ILOAD:1.

If there are multiple return statements in one inlined method, all of them are transformed to store the return value to an additional virtual local variable. Only the last return in the inlined code also creates a node that loads that new virtual local variable to push it on the virtual stack so that the following node can consume the correct return value. If there is just one return statement, nothing has to be done and the following node can pop the return value from the stack directly.

Control Dependencies In Figure 9.5 it is obvious that the basic block from 7 to 16 is only executed when 4:IFEQ:15 is evaluated to false. Thus, 4:IFEQ:15 is the *controller* of all instructions in the basic block. Logic and arithmetic instructions of this basic block will be executed speculatively. Write accesses to the memory (*Local Variable Memory* and *Heap Memory*) will only be executed if the corresponding predication signal provided by the C-Box is true so that no errors are introduced. Read accesses to the *Heap Memory* are also predicated, so that no unnecessary stalls are introduced when cache misses occur while accessing data that is actually not needed. In this example 8:ISTORE:2 and 15:ISTORE:1 are controlled by 4:IFEQ:15. Thus, a control dependency is added between those nodes as shown with red dotted lines in Figure 9.6.

Further Optimizations Some nodes take just one operand but have be transformed to more generic operations that map to the operations implemented on the CGRA. The node 4:IFEQ:15 for example compares the operand to zero. It will be transformed to a operation that compares if two operands are equal. The additional operand will be the constant 0.

During scheduling each local variable in the graph will be strictly associated with one specific *Location* in one specific register file address on one specific PE (abbreviated by PE<*PE number*>.<*address*>). A store instruction always has to ensure that the value is written to that exact *Location*. The next access to that local variable will retrieve the current value from that *Location*. The corresponding load instruction can be deleted from the CDFG and the following instruction will get its data from the store instruction. In Figure 9.7 the instruction 11:ILOAD:2 is deleted and 13:ISUB gets its data directly from 8:ISTORE:2. Intermediate values like the result of the subtraction can be stored at any *Location*.

Predication and Speculation Code Example 5 shows an if-else construct in which the variable a gets different values in dependence of the value of i. In this work the control flow is handled with a combination of speculation and predication. The arithmetic and logic instructions are executed speculatively. The results are stored in arbitrary free *Locations*. Write accesses to memories (both *Local Variable Memory* and *Heap Memory*) are executed only predicatively. Thus, in Figure 9.8 the instructions

Code Example 5: Predication and speculation example

```
if (i != 0) then
    a = b + c;
else
    a = b − c;
```

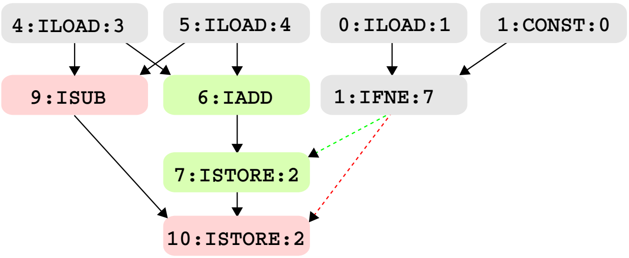


Figure 9.8.: CDFG of Code Example 5

6:IADD and 9:ISUB are executed speculatively and there is no dependency to the comparison 1:IFEQ. In contrast to that both 7:STORE:2 and 10:STORE:2 are dependent on 1:IFEQ:7 whose results are stored in the C-Box as described in Section 7.4. The corresponding values are loaded from the *Condition Memory* and only the instruction whose condition is true, writes its value to the *Location* of the corresponding local variable.

It can be seen, that this scheme increases parallelism as it allows to start the actual calculation (instruction 6:IADD and 9:ISUB) before the comparison was executed. Unfortunately, at the same time it may result in long sequences of predicated stores like 7:ISTORE:2 and 10:ISTORE:2. In practice those sequences do rarely introduce a delay because other instructions have to be executed at the same time as well.

In the following, simplified CDFGs like in Figure 10.1 will be used. The nodes will represent expressions on a higher level like if(i<10) and they will be arranged so that the schedule length of an ASAP (as soon as possible) schedule can be seen directly.

9.4. Resource and Routing Constrained Scheduling

The resource and routing constrained scheduler described in [63] is used to map the CDFG to the CGRA. This scheduler is based on a list scheduler and gives near optimal solutions [62]. Figure 9.9 shows the schedule of the CDFG shown in Figure 9.7. In timestep 0 PE 0 executes 4:IFEQ:15. The first operand is the local variable with index 1 (3:LOAD:1) which is stored at its *Location* in PE0 at register file address 0. The second operand is a constant which is read from register file address 256 in PE1. As described in Section 8.1 the MSB of the address denotes whether the following bits code an address or a constant. In this case the MSB is set ($256_{10} = 100000000_2$) and the following bits ($00000000_2 = 0_{10}$) will be provided to PE0 as the constant 0. The result of 4:IFEQ:15 will be stored in the C-Box at the address 0. At t=1 PE1 stores the value of local variable 1 in the local variable 2 (*Location* PE1.1) in dependence of the predication value that the C-Box loads from address 0 in the *Condition Memory*. In the last step PE0 performs the subtraction and the result is stored conditionally back to the *Location* of local variable 1. Note that here the nodes 13:ISUB and 15:ISTORE:1 are mapped to the same FU and can therefore be fused into one conditional subtraction.

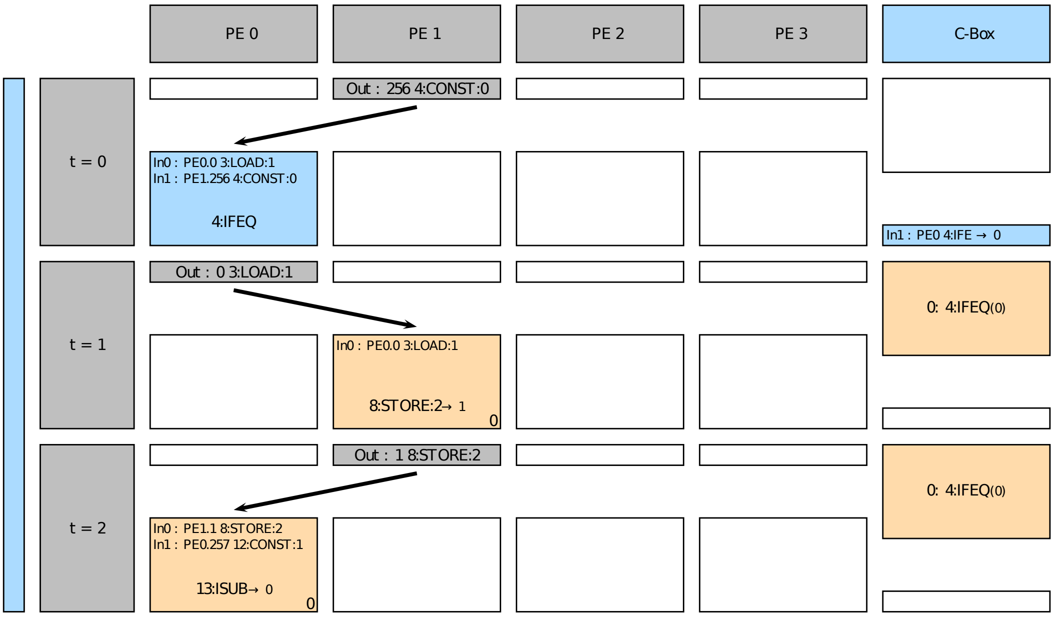


Figure 9.9.: Schedule of CDFG shown in Figure 9.7 on a CGRA with four PEs

The blue bar on the left side of the schedule shows that the loop goes from t=0 to t=2. This is helpful if there are nested loops which are marked in the same way. Here, it means that at t=2 the *Context Control Unit* performs an unconditional relative jump two contexts back to the beginning. At t=0 a conditional absolute jump is performed to the idle context depending on the result of `4:IFEQ:15`. Nested loops are left with a relative jump to the first context behind the nested loop as shown in Appendix C. By using absolute jumps only to exit the outermost loop, it is possible to store the contexts of this schedule contiguous at any address in the context memory without having to adapt any jump address. That way, kernels can be easily be swapped in and out of the context memory in case many different kernels have to be executed on the CGRA and the context memory is to small to hold all of them.

When an operand is not directly accessible to a PE due to routing constraints, copy nodes will be added to the CDFG in order to route the desired value to the PE via connected PEs. Copy nodes are marked gray in Appendix C. Note that only one operand can be provided from the local register file. This means that some times one of the operands has to be copied to a neighbor so that it can be used.

9.5. Context Management

When the scheduler is finished, the contexts of this kernel and the contents of all *Interface Configuration Memories* are generated. The contents of kernel tables in both CGRA and *Token Machine* have to be written directly via Wishbone peripheral bus. All other *Interface Configuration Memories* entries and the contexts can be written optionally. If they are not written and they are needed later on, an interrupt is issued and the interrupt service routine will load the data with the help of the information in the kernel tables.

When not all data of all kernels fits in the memories at once, the data of other kernels have to be replaced so that the current kernel can be loaded and executed. The decision which kernels to replace is related to the replacement strategies in data caches, but the strategies cannot simply be reused, because different kernels have different sizes. So sometimes is is necessary to replace two or more older kernels in order to load one large kernel. Two strategies are possible:

In the first strategy, the n kernels that were least recently used are replaced. The number n has to be as small as possible while still enough space is freed to fit the large kernel to be loaded. Unfortunately, it is not guaranteed that these n kernels lie in a contiguous region of the context memories. Thus, the kernel to be loaded has to be fragmented and new relative jump addresses have to be added to the context of the *Context Control Unit* and existing relative jump addresses have to be adapted.

In the second strategy only continuous memory regions are freed so that the kernel to be loaded will not be fragmented and the contexts don't have to be adapted. This leads to the question which set of kernels will be replaced: Is it better to replace a kernel that was almost never used and one that is frequently used or is it better to replace two kernels that are used sometimes? One possible approach is to use PLRU (pseudo least recently used) on different levels. On each level the context memory is divided in different parts of equal size. The higher the PLRU level, the bigger is the size of the memory parts. On every access all levels of PLRU are updated. When a new kernel has to be loaded, the lowest PLRU level whose memory part size fits the kernel, is used to decide where to store the kernel. All kernels that are stored in this part have to be replaced.

This question is not covered in this work. Instead a simple FIFO solution was implemented which replaces the kernel(s) that reside in the context memory for the longest time[66]. This is a suboptimal solution but it ensures the correct functionality of the CGRA.

9.6. Bytecode Patching

When the kernel tables are filled in both the CGRA and the *Token Machine*, the bytecode that corresponds to the kernel can be patched so that the kernel will be executed on the CGRA from now on. First, the original bytecode has to be stored, so that the patch can be revoked if needed. Afterwards, the bytecodes shown in Section 8.2 are written in the instruction memory at the beginning of the kernel code. This is done using the native method `AmidarSystem.writeAddress(int address, int value)`. In the end the instruction cache has to be invalidated, by writing to a special peripheral register in the *Token Machine*, so that the patched bytecode is actually loaded and executed.

When the bytecode is patched, it has to be ensured, that the program counter is not in the patch region. Otherwise, wrong bytecodes will be executed. For that purpose the start and the end of the patch region are written to peripheral registers in the *Token Machine* beforehand. Additionally, a status bit denotes whether a method call is contained in the patch region. If not, it is safe to patch the bytecode when the PC is lower than the start of the patch region or when the PC is higher than the end of the patch region. If there is a method call in the patch region, it is possible that this method is executed right now and when the method is finished it will return to the patch region. Thus, if there is a method call in the patch region, it is only safe to patch the code, if the PC was recently equal to the end of the patch region and it was not equal to the start of the patch region since. The functionality to check whether the bytecode can be patched right now is implemented in hardware and is included in the *Token Machine*. The result can be read via Wishbone peripheral bus. If the code can not be patched, the mapping thread will sleep for 10 seconds and try again afterwards. To avoid polling, an interrupt that is issued when the code can be patched, could be used (not implemented in this work). During the patching process the thread may not be switched. Otherwise, the *Token Machine* might start decoding code from the patch region. Thus, the *Thread Scheduler* is disabled via software during that process.

Part III.

Memory Subsystem Optimization

10. High-Level Compiler Optimizations

When the synthesis algorithm is executed as described above without further optimizations, successive loop iterations are always executed sequentially. Thus, the utilization of the PEs is not satisfactory and the memory ports are not used to their full potential. Thus, several optimizations have to be applied in order to be able to fully evaluate the memory subsystem. In this chapter software pipelining and *Aliasing Speculation* will be discussed. These optimizations will on one hand result in higher parallelism so that more memory accesses can be executed in parallel. On the other hand this also reduces the total number of memory accesses as described in Section 10.1.3.

10.1. Software Pipelining

Note: Parts of this section have already been published in [31]. The marking of self-citation is omitted in order to improve the reading flow.

The standard approach for software pipelining is to implement a modulo scheduling so that the execution of different loop iterations overlap and the parallelism is increased. As mentioned above, the synthesis algorithm should have a low complexity and should scale well. Therefore, another approach was chosen.

In [1] Aiken and Nicolau describe an algorithm to compute time-optimal loop schedules. In their approach the loop is unrolled u-times and then scheduled. Afterwards, the nodes with a mobility higher than one are moved into different time slots so that recurring patterns emerge in the schedule. These patterns are then again combined to a new loop. The result is then a software pipelined loop with a prologue and an epilogue. In this work we will also unroll the loop u-times but in order to keep the complexity low, the pattern search is omitted and the whole unrolled code will be executed. Doing this naively will not result in the desired short schedules because for example new common subexpressions and constants are created. Thus, common subexpression elimination and constant folding have to be implemented as well. The following sections describe all the optimizations that are implemented to achieve loop pipelining efficiently.

10.1.1. Partial Loop Unrolling

In order to increase the utilization of the PEs and the memory ports, partial loop unrolling of the innermost loops was implemented. The unrolling is done on bytecode level by copying the loop body u-times into the loop.Code Example 7 shows the resulting code when Code Example 6 is unrolled three times. The Figures 10.1 and 10.2 show the resulting dependency graphs, respectively. It can be seen that when simply copying the loop body, all three loop iterations (marked in yellow, green and blue) are still executed

Code Example 6: Minimal code example

```
for (i=0; i<m; i++) do
    f(a[i]);
```

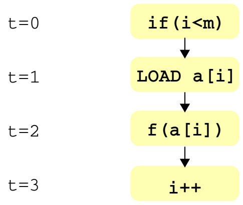


Figure 10.1.: Dependency graph of Code Example 6

Code Example 7: Code example visualizing loop unrolling

```
for (i=0; i<m; i++) do
    f(a[i]);
    i++;
    if i < m then
        f(a[i]);
        i++;
        if i < m then
            f(a[i]);
```

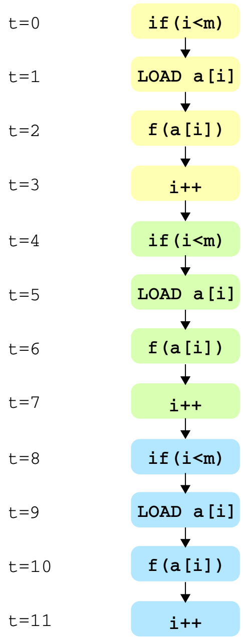


Figure 10.2.: Dependency graph of Code Example 7

sequentially because each iteration depends on the increment of the loop index i in the previous iteration[1].

To overcome this circumstance the instruction `i++` is split into two instructions `i'=i+1` (*calculation*) and `i=i'` (*store*) during graph generation. So different versions of the variable `i` are created which leads to an improved dependency graph shown in Figure 10.3. This is done for each local variable to decrease dependencies between adjacent loop iterations.

Both store instructions `i=i''` and `i=i'''` only receive a write-enable signal if the corresponding `if`-instruction returns true.

Assuming that the method calls `f(x)` are independent, the executions of all iterations can overlap. In practice this is not always the case. Code Example 8 shows the code for the calculation of Fibonacci numbers. It is obvious that each iteration depends

[1] The loop condition `i<m` has to be evaluated each time because the value of `m` is not known during compile time. Additionally, it is possible that `m` is modified in the method call `f(a[i])`.

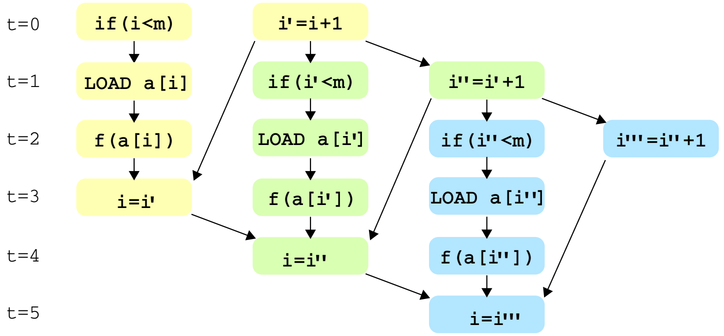


Figure 10.3.: Improved Dependency graph of Code Example 7

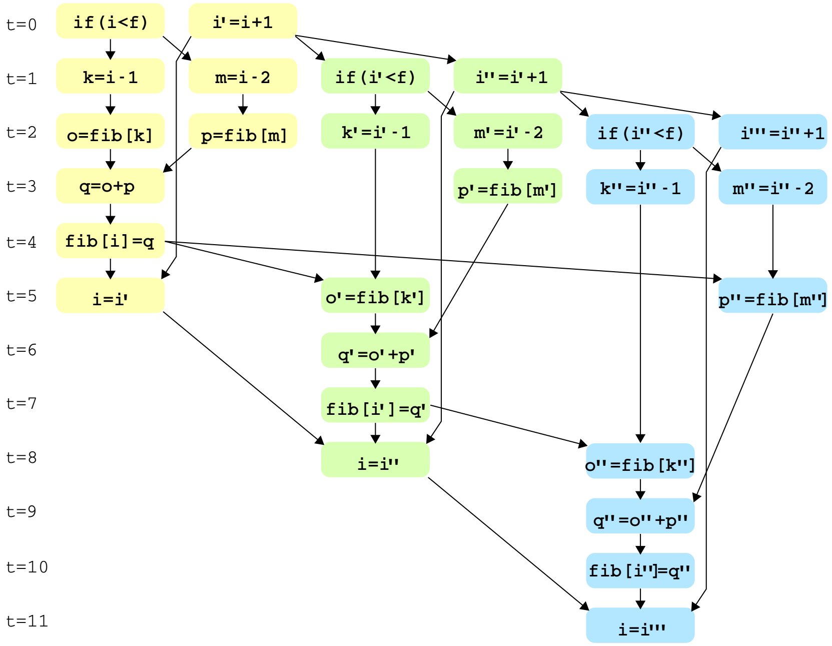


Figure 10.4.: Dependency graph of Code Example 8 when unrolled three times

directly on the previous two iterations. Thus, there are long dependency chains in the dependency graph when the loop is unrolled as shown in Figure 10.4. Note that the auxiliary variables `k`, `m`, `o`, `p` and `q` were introduced in the graph to represent intermediate results.

Code Example 8: Calculation of Fibonacci Numbers

```
for (i=2; i<f; i++) do
    fib[i] = fib[i-1] + fib[i-2];
```

It can be seen that the length of the ASAP schedule for one iteration (yellow nodes) would be 6 time steps. When the loop is unrolled three times the ASAP schedule would have a length of 12 time steps which is 4 time steps per iteration on average.

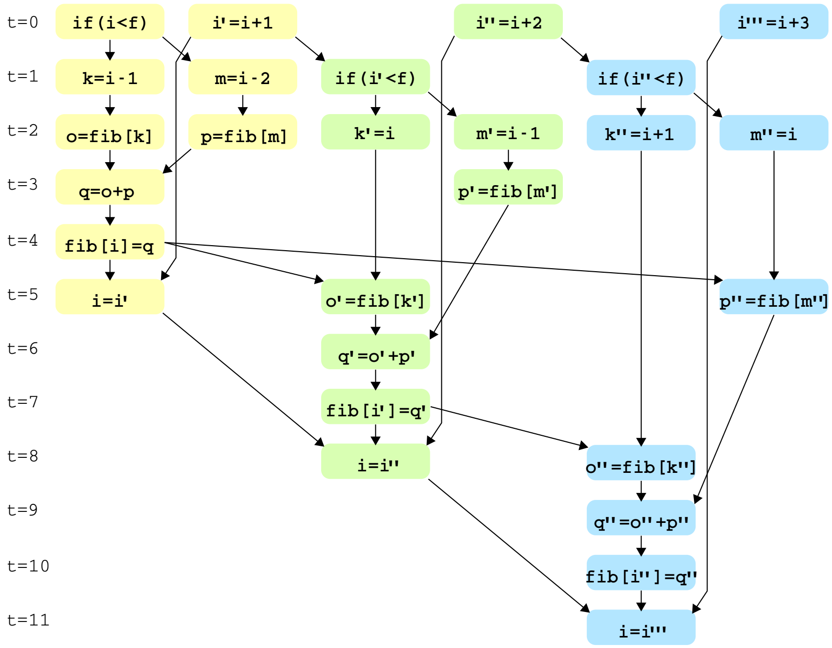


Figure 10.5.: Dependency graph of Code Example 8 with constant folding

10.1.2. Constant Folding

To decrease the ASAP Schedule length further, constant folding was introduced. For example the calculation of the variable `k’` can be simplified to `k’=i’-1=i+1-1=i`. The same can be done for the variables `m’`, `k"`, `m"`, `i"` and `i"`. The resulting dependency graph can be seen in Figure 10.5. It is obvious that there are less calculations to perform and there are less dependencies but the length of the ASAP schedule is unaffected.

10.1.3. Common Subexpression Elimination and Instruction Folding

After the constant folding has been performed common subexpression elimination and instruction folding can be applied efficiently. For example the array access `o’=fib[k’]` can be omitted because `k’=i` and thus `o’=fib[i]=q`. Similarly, the array access `p’=fib[m’]` can be simplified to `p’=fib[k]=o`. Thus, the calculation of the variable `q’` simplifies to `q’=q+o`. A similar calculation holds for `q"`.

That way, load forwarding is implemented and the number of array accesses in the whole graph is reduced from 9 to 5.

Figure 10.6 shows the resulting dependency graph. It is obvious that now the ASAP schedule for three iterations has a length of 8 time steps which is $2.\overline{6}$ time steps per iteration on average.

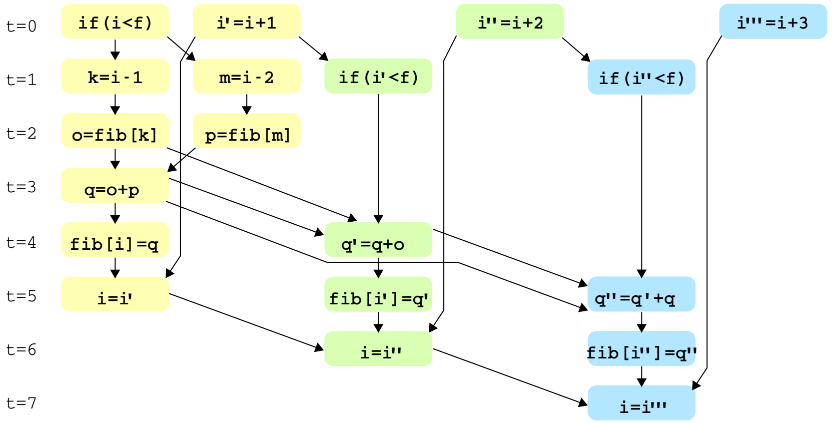


Figure 10.6.: Dependency graph of Code Example 8 with constant folding and CSE

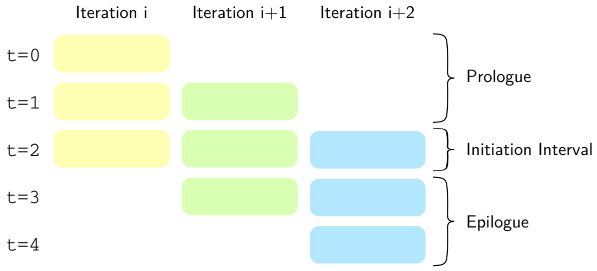


Figure 10.7.: Overlapping Execution of Consecutive Loop Iterations

10.1.4. Relation to Modulo Scheduling

As mentioned above, partial loop unrolling is related to modulo scheduling as it overlaps the execution of consecutive loop iterations. Figure 10.7 shows exemplarily how loop iterations overlap when the loop body is unrolled three times. During the first two time steps the software pipeline is filled while in the last two time steps the software pipeline is flushed. Those intervals are called prologue and epilogue, respectively. Between prologue and epilogue the pipeline is completely filled and all parallelism is exploited. The initiation interval (II) is the time between the start between two loop iterations. The II corresponds to the length of the patterns that are transformed into the new loops in [1].

In modulo scheduling only the II has to be repeated when more than three loop iterations have to be executed in the given example. Thus, the execution time for modulo scheduling with more than three iterations is

$$t_{modulo} = N \cdot II + l$$

while l is the length of prologue plus the length of epilogue and N is the number of iterations.

In contrast to that when loop unrolling is used, also prologue and epilogue will be

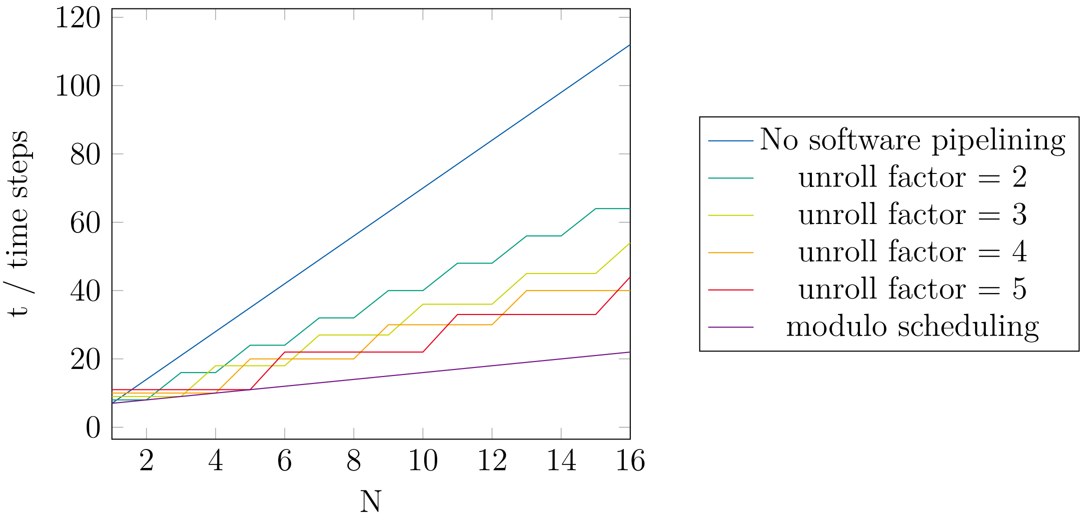


Figure 10.8.: Execution times for different software pipeline mechanisms

executed again. Thus, when an unroll factor u is used the execution time is

$$t_{unroll} = (u \cdot II + l) \cdot \left\lceil \frac{N}{u} \right\rceil$$

Figure 10.8 shows the execution times with $II = 1$ and $l = 6$ for naive execution without software pipelining, software pipelining with different unroll factors and when modulo scheduling is used. It can be seen that for more than one loop iteration unrolling performs always better than naive execution but is never better than modulo scheduling. When $u = N$ modulo scheduling and unrolling result in the same execution times. In the worst case the number of loop iterations N is no multiple of u. Then $u - 1$ unrolled iterations of the original loop body are executed unnecessarily. The number of unnecessarily executed loop iterations is $e = u \cdot \left\lceil \frac{N}{u} \right\rceil - N < u$. From this follows that if $N \gg u$ this effect is negligible. If N is known in advance u should be chosen to be a divider of N so that $e = 0$.[2]

A relative runtime can be defined to compare the performance of partial loop unrolling with modulo scheduling:

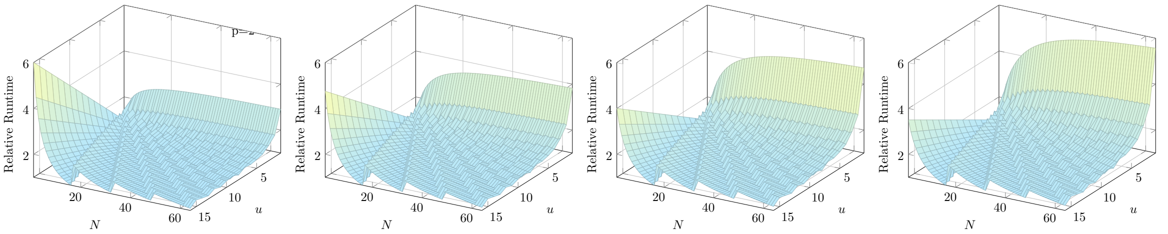
$$t_{relative} = \frac{t_{unroll}}{t_{modulo}} = \frac{(u \cdot II + l) \cdot \left\lceil \frac{N}{u} \right\rceil}{N \cdot II + l}$$

The quotient $p = \frac{l}{II}$ is a good metric to describe the parallelism of a loop, as smaller II result in a higher throughput and a higher value of p. If l is also small, p is smaller but the potential to speed up the execution is also small as naive execution is already fast. With $l = p \cdot II$ follows:

$$t_{relative} = \frac{(u \cdot II + p \cdot II) \cdot \left\lceil \frac{N}{u} \right\rceil}{N \cdot II + p \cdot II} = \frac{(u + p) \cdot \left\lceil \frac{N}{u} \right\rceil}{N + p}$$

Figure 10.9 shows the relative runtimes for different values of p in dependency of N

[2] Taken from [31]



(a) $p = 2$ (b) $p = 3$ (c) $p = 4$ (d) $p = 5$

Figure 10.9.: Relative runtimes for different values of p (higher values of p correspond to more parallelism in the loop)

and u. For loops with little parallelism (small p) and high u and small N the number of unnecessarily executed loop iterations e has a high negative impact on the relative runtimes. For example for $p = 2$, $u = 16$ and $N = 1$ partial loop unrolling performs six times worse than modulo scheduling. For loops with high parallelism (high p) the repeated execution of prologue and epilogue has a negative impact on the relative runtimes when small values of u are used. In order to keep the complexity of the mapping algorithm low, a fixed unroll factor is used for all loops. Empirical studies have shown that $u = 4$ results in good runtimes for common values of p and N.

10.2. Aliasing Speculation

The *Heap Memory* operations described in Section 5.2 perform accesses to the heap. When accesses to the heap are independent they can be scheduled out of order or parallel which can result in a better performance. Thus, it is beneficial to find out for all accesses whether they are independent during compile time. In section 5.2 it is also shown that not always all the information about the heap accesses is available during compile time. For that reason *Aliasing Speculation* was implemented to increase the the performance.

10.2.1. Dependencies Between Heap accesses

Two read accesses are always independent. If at least one of the accesses is a write access, they are dependent when both access the same memory object with the same offset. As this information is not always available during compile time, the following rules help to find dependencies.

Two Heap accesses are independent if:

- Both accesses are executed in different branches.
- Both accesses are read accesses.
- Both accesses are of different types (array access, static field access or object field access) because different types are stored in different memory regions (see Section 5.2).

- Both accesses are array accesses to different array types (e.g. integer and float). Note that in contrast to C it is not possible in Java to cast an integer array to a float array.

- Both accesses are of same type but use different offsets.

For accesses to static fields or object fields it is always possible to determine the offsets during compile time because it is encoded in the bytecode. The index for array accesses is computed during runtime can be data dependent. Thus, it is not always possible to determine whether the offsets are different. The offsets A and B are different if :

- A and B are both constants with different values.

- $A = B + c$ where c is a constant different from 0.

If none of these criteria is true, two cases can be distinguished:

1. Both access definitely the same memory object but it is not clear whether both access the same offset: `arrayA[0]` and `arrayA[index]`.

   During compile time it is not known whether `index` holds a value different from 0. Thus, it is assumed that both accesses are dependent. Note that in some cases it is possible to prove that `index` never holds the value 0 by analyzing the bytecode of the whole application. In this work this is not done in order to keep the algorithm simple as it will be executed during runtime. The assumption that both accesses are dependent is very likely because index variables normally change their value regularly during runtime. Thus, it is likely that `index` holds the value 0 at least once during the runtime. This assumption is pessimistic but ensures that errors are introduced due to wrong memory access order .

2. It is not clear whether both references describe different memory objects: `arrayA[index]` and `arrayB[index]`. Both accesses are assumed to be independent, because it is assumed that the local variables `arrayA` and `arrayB` actually point to two different arrays. This assumption is optimistic and can introduce errors when the two operations are falsely scheduled out of order. Programming conventions ensure that aliasing occurs very seldom, but still it has to be ensured that no errors occur. Note that the references can be loaded from a local variable, an object field or from an array.

Figure 10.10 shows the decision diagram how to handle potential aliases. Blue arrows denote unsure statements where some property cannot be proved. In those cases speculation follows. The green box shows a pessimistic speculation which does not introduce errors while the red box shows an optimistic speculation which improves the performance but may result in false memory access order. In that case a safety mechanism has to be implemented to ensure that this does not result in errors. This mechanism is described in the following.

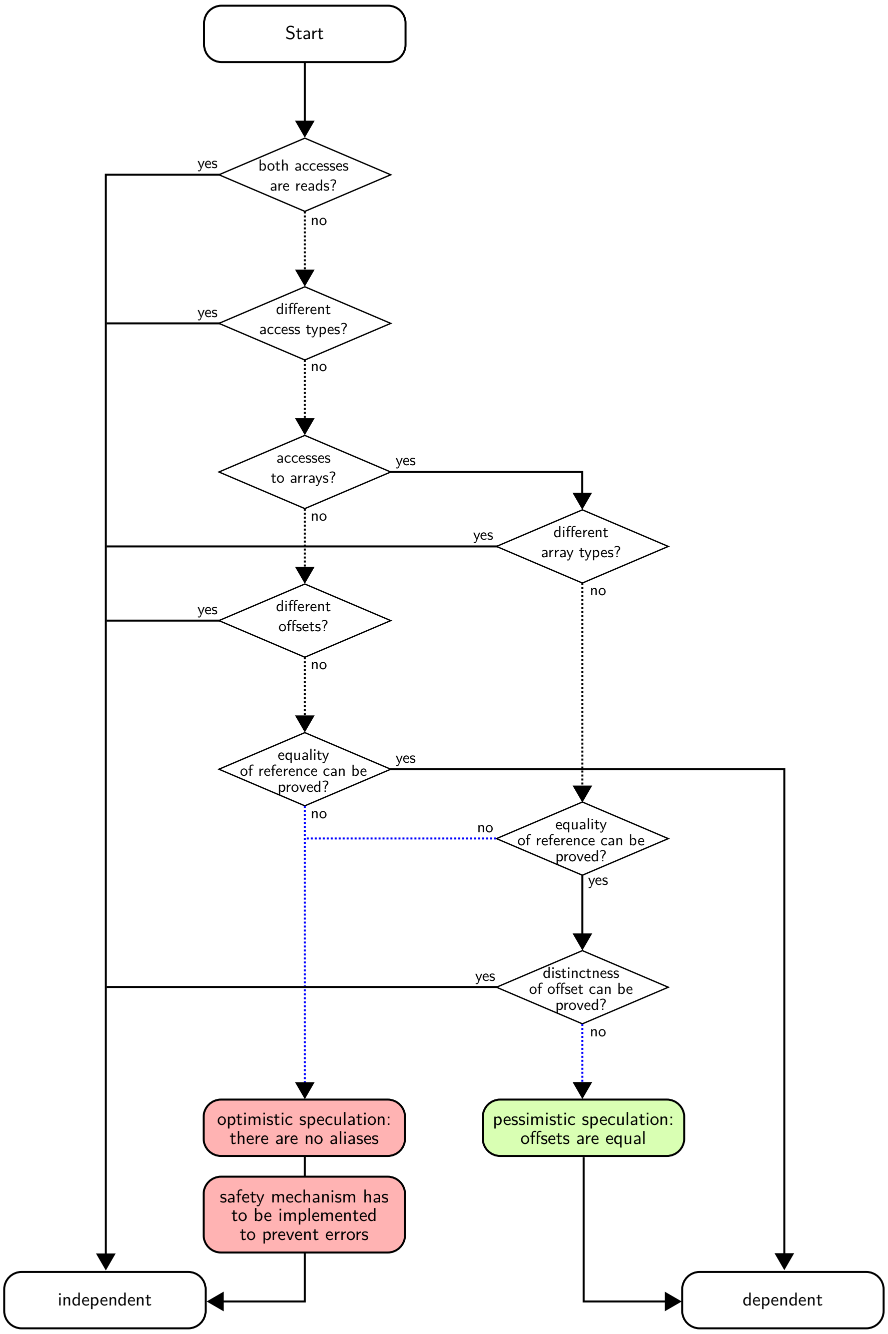

Start
both accesses are reads?
yes
no
different access types?
yes
no
accesses to arrays?
yes
no
different array types?
yes
no
different offsets?
yes
no
equality of reference can be proved?
yes
no
equality of reference can be proved?
no
yes
distinctness of offset can be proved?
yes
no
optimistic speculation: there are no aliases
safety mechanism has to be implemented to prevent errors
pessimistic speculation: offsets are equal
independent
dependent

Figure 10.10.: *Aliasing Speculation* decision diagram

10.2.2. Aliasing detection

In order to ensure correct program execution it has to be checked during runtime whether two references that were assumed to point to different memory objects actually contain different handles. If they are the same, the assumption was false and the execution on the CGRA has to be aborted. An interrupt is issued and a rollback mechanism has to be started that ensures that the code will be executed correctly on the AMIDAR processor. In this work no rollback mechanism was implemented[3]. Instead the execution is aborted with an error message. This case occurred only twice (in Twofish and in Blowfish) in the benchmark set used in work. Here the reference to an array was stored in an object field and it was handed as a parameter to a method that uses the object field.

The speculation takes place during CDFG generation which is described in Section 9.3. All pairs of memory accesses that are assumed to be independent and might therefore introduce errors are stored in a list. Then the CDFG is scheduled and for all those memory access pairs it is checked whether they are actually scheduled out of order. If yes, special instructions are added to the schedule a posteriori to check the inequality of the handles during runtime[4]. An interrupt is issued when the handles are the same. Those instructions can not be added to the schedule at arbitrary time. The check whether the handles are equal has to take place early enough so that the wrong assumption did not cause any irreversible damage like overwriting memory or local variables.

Assume that the first memory access[5] is a read access and the second is a write access as shown in Figure 10.11. The comparison of both handles has to take place before the execution of `DMA 2` and can be executed right after both handles are loaded at the earliest. Thus, the time of the comparison has to hold the following conditions: $t_{cmp} < m \wedge t_{cmp} > k \wedge t_{cmp} > l$. The conditions can not be fulfilled if $l \geq m$ or $k \geq m$. The latter can never occur because there is a data dependency between `DMA 2` and `Handle 2`. Thus, k is always smaller than m. To ensure that $l \geq m$ never occurs a control dependency between `Handle 1` and `DMA 2` (gray dashed line) has to be introduced during CDFG generation. The scheduler will map the comparison to the PE that can access both values the fastest. If the condition $t_{cmp} < m$ cannot be achieved, empty time steps have to be inserted into the schedule, which leads to a significant performance loss.

If `DMA 2` is a read access, the $t_{cmp} < m$ can be relaxed to $t_{cmp} < m'$ where m' is the execution time of the first writing operation (either to the memory or to a local variable) that depends on `DMA 2`.

[3] In Section 9.1 a short description of such a mechanism is given.

[4] Another approach would be to add those special instructions to the CDFG before scheduling. Then the scheduler has to be extended so it can optionally chose not to add those instructions to the schedule because the corresponding memory access pair was not scheduled out of order. This option was rejected as this increases the complexity of the scheduler even further.

[5] First (or second) access is always referring to the program code in this section. The first access always has a lower bytecode address than the second access. When both are scheduled out of order on the CGRA they will still be called first (or second) access to avoid confusion even though the execution order changed.

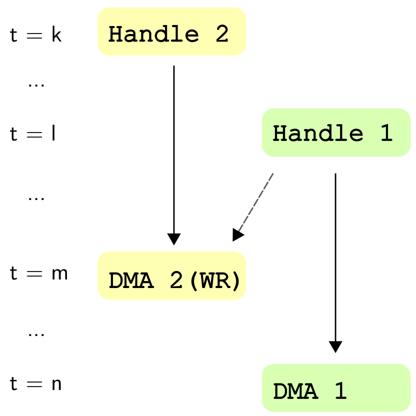


Figure 10.11.: Simplified schedule with out of order execution of two DMA accesses. The nodes `Handle 1` and `Handle 2` denote load operations either from the heap or a local variable, that load the handles for both DMA operations.

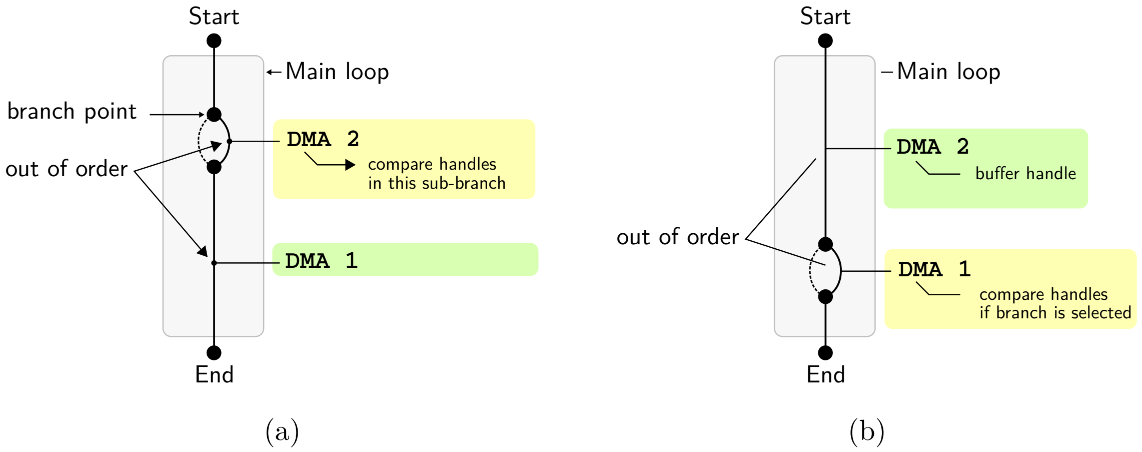


Figure 10.12.: Out of order DMA acesses in different sub-branches

If there are two other instructions `DMA 3` and `DMA 4` that use both the handles `Handle 1` and `Handle 2` and are also executed out of order, no second comparison has to be performed.

Accesses in Different Sub Branches If one of the DMA instructions is part of a subbranch, the comparison of the handles needs only to result in an interrupt if that subbranch was actually executed as shown in the Figures 10.12. This has two implications that have to be considered: First, the comparison can only be executed when the branch decision was already executed. This tightens the constraints for t_{cmp} especially in the case shown in Figure 10.12 (b). This might lead to an increased number of empty time slots in the schedule. Second, if DMA 3 and DMA 4 are executed in different sub-branches a second comparison or additional C-Box instructions to calculate the combined condition have to be inserted.

Because of the second implication, a relaxed but pessimistic scheme was implemented.

The comparison will be executed in the sub-branches of the instructions that load the handles and not in the sub-branch of the DMA instructions. Theoretically, the comparison might now start an interrupt unnecessarily if one of the DMA instructions was not executed. In practice the sub-branches of DMA instruction and the sub-branch of handle load instruction are the same and in our benchmark this case never occurred.

11. Memory Subsystem

The memory subsystem of a hardware accelerator has different requirements than the memory subsystem of a multi-core processor. Both aim to provide the processor cores or the PEs respectively with a high memory bandwidth with low latency. The difference is that in a multi-core processor normally different threads are executed on each core. Good software design tries to ensure that all threads work on disjoint parts of the memory so that the threads can operate more independently. Time consuming synchronization between the threads is minimized. This means that the L1 caches of the multi-core processors mostly hold disjoint data and coherence is a relatively small problem. In a hardware accelerator all PEs work as a network on the same problem. This means they all access the same or at least a similar memory region at the same time. The caches of each PE with DMA will hold mainly the same data. This leads to different requirements because coherence now becomes major challenge and its implementation has a great impact on the memory bandwidth.

Additionally, when a cache miss occurs in one core of a multi-core system, only the thread running on that core has to be stalled. All other threads can continue their work independently. In our CGRA all PEs have to stop working if a cache miss occurs in one cache because all PEs work synchronously with the same context counter (see Section 7.3).

The next sections will provide information about different approaches to implement a cache system with different coherence protocols. This chapter closes with a description how coherence can be supported by smart binding of memory instructions to certain PEs during scheduling.[1]

11.1. Cache Architecture

Figure 11.1 shows the cache hierarchy in the AMIDAR processor coupled with the CGRA. All L1 caches are connected to the L2 cache via the *Coherence Controller*. This means that it is not necessary to flush any caches when the execution of a kernel is started on the CGRA. The L2 cache accesses the DRAM Main memory via AXI bus. A snoop based system is implemented, as there are few caches in the system and they share many cache lines as described in Chapter 4.1.

[1] Parts of this chapter have already been published in [75] and [61]

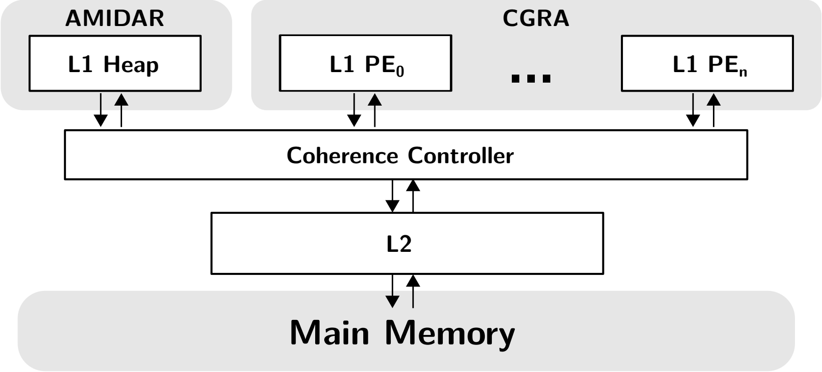


Figure 11.1.: Cache Hierarchy of the AMIDAR processor coupled with a CGRA[75]

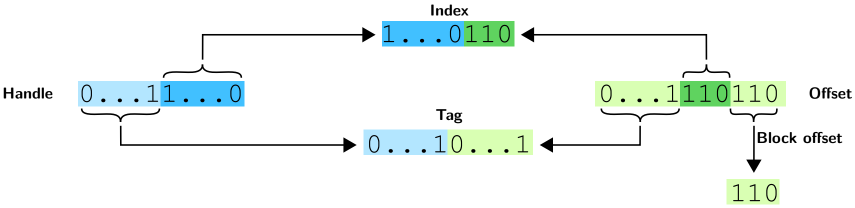


Figure 11.2.: Scheme of index and tag generation in a virtually addressed cache

L1 Caches As already mentioned in Section 6.2, the L1 caches in AMIDAR and in the CGRA are virtually addressed with handle and offset so that the physical address of a memory object has not to be loaded from the *Handle Table*. Index and tag are generated according to the scheme shown in Figure 11.2. In this example a cache with eight words per line is used. Thus, the lowest 3 bits of the offset are used as block offset. The next 3 bits are used as the lowest bits of the index. The rest of the index is filled with the lowest bits of the handle. The remaining bits of both handle and offset are combined to the tag.[2]

A consequence from this scheme is that one cache line can only contain data of one object at a time. This has two disadvantages compared to physically addressed cache. First, valuable cache space is wasted when small objects are loaded because unused slots in one cache line cannot be used to load other objects as shown in the top of Figure 11.3. This will also result in an increased number of write backs. And second, this leads to an increased miss rate: Assume that several small objects lie in the main memory in a contiguous area. When the first object (red in Figure 11.3) is loaded into a physically addressed cache, other objects will automatically be loaded into the same cache line. Accessing the second object (green) later will lead to cache hits. In the virtually addressed cache this can not happen as one cache line only contains one object.

At the same time the virtually addressed cache has the advantage that no address

[2] There are also dynamic schemes that adapt the number of bits from the offset, that are used in the index. The bigger the offset, the more bits are considered for the index. Then the tag has to be extended so that it is clear which scheme was used to generate this tag. This will not be covered in this work. See [40] for more details.

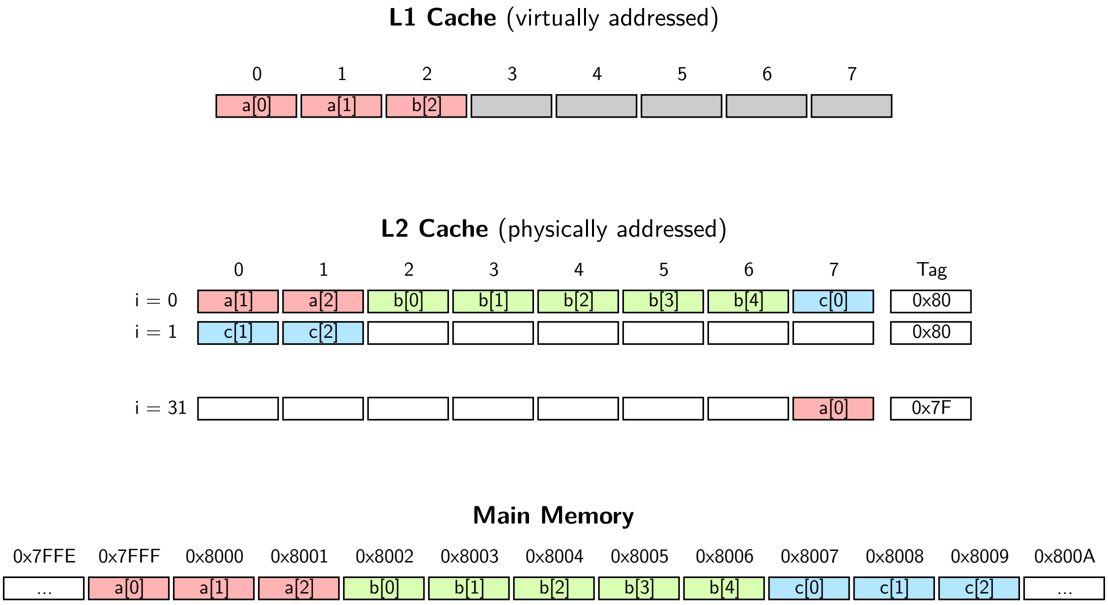


Figure 11.3.: Cacheline Alignment in AMIDAR caches

resolution has to be performed in case of a cache hit and the Garbage Collection is eased as addresses of the moved objects only have to be updated in the *Handle Table*.

As a conclusion one can say that using a virtually addressed cache trades a better access time in case of a cache hit against a slightly worse hit rate.

L2 Cache The question whether the L2 cache should be addressed virtually or physically is not easy to answer.

When the L2 cache is addressed physically, the space in the L2 cache is used more efficiently especially if larger line sizes are used and the miss rate is expected to be lower. The access to the *Handle Table* can be done speculatively and partly in parallel to the access to the L1 caches (see Figure 13.5). Thus, the penalty for the address resolution is low. A disadvantage of a physically addressed L2 cache is that the L2 cache has to be modified when the Garbage Collector moves objects in the main memory (not implemented during this work). Figure 11.3 also shows that cache lines in L1 and L2 are not aligned due to the different addressing schemes. This means that when array `a` (red) is loaded into the L1 cache, two cache lines from the L2 have to be accessed.

When a virtually addressed L2 cache is used, the L2 cache can be accessed earlier which results in better access times in case of a miss on L1 and a hit in L2. Additionally, Garbage Collection can be implemented easier.

In order to find the better solution both virtually and physically addressed L2 caches were implemented in a simulator and evaluated with different amounts of words per line. The results will be presented in Part IV of this work.

11.2. Coherence Protocol

Note: Parts of this section have already been published in [61]. The marking of self-citation is omitted in order to improve the reading flow.

When different caches use the same data, it has always to be ensured that all caches use the correct data. Thus, when one cache writes to a cache line that is also loaded in at least one other cache, all the other caches have to be notified that the value has changed. Well studied coherence protocols provide this functionality. In the following two important protocols that will be used in this work are described.

11.2.1. MOESI Protocol

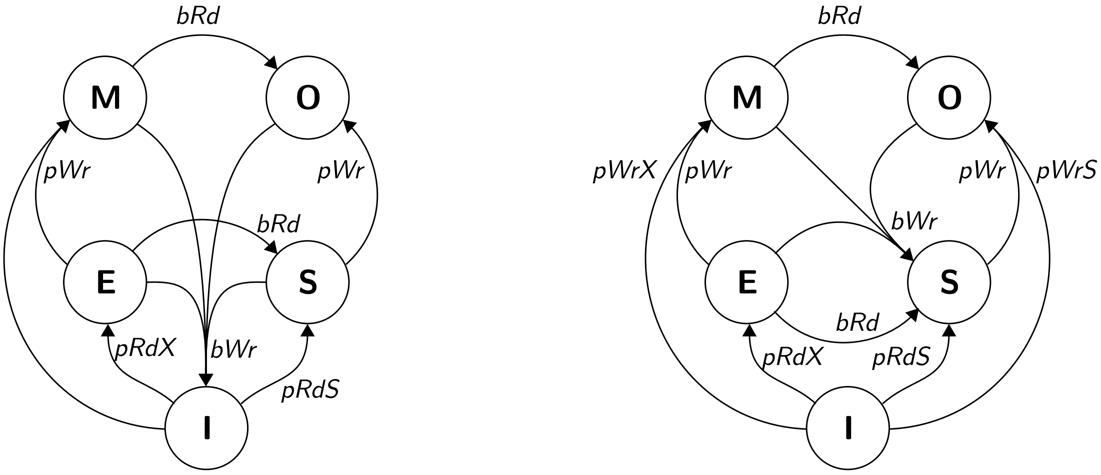
In the MOESI Protocol each cache line can either be invalid or valid. If it is valid it can be either shared or exclusive and it can either be modified or unmodified. This can be described by the states that are shown in Table 11.1. Figure 11.4 (a) shows the state transitions when the connected processor reads from or writes to a cache line (*pRd* and pWr) or another cache sends a read or write request over the bus (*bRd* and bWr). Note that *pRdX* means that the data was read from the next memory level and the data is exclusive in this cache. *pRdS* means that another cache at the same level provided the data and it is now shared. In MOESI $pWrS$ does not exist, because writing to a cache line always invalidates it and makes it exclusive.

11.2.2. Dragon Protocol

The Dragon Protocol allows the same states as the MOESI protocol. The difference between both is that a bWr does not invalidate the corresponding cache line but the newly written value is directly provided and the cache line is updated and goes to the *Shared* state. The update can be done in the background and no delay is needed if no other requests are posted on the bus during that time. Figure 11.4 (b) shows the corresponding state transitions.

Table 11.1.: MOESI states

	exclusive	modified	invalid
Modified	●	●	
Owned		●	
Exclusive	●		
Shared			
Invalid			●



(a) MOESI

(b) Dragon

Figure 11.4.: Cache line state transitions

11.2.3. Comparison

Table 11.2 shows different events that delay an access to the cache. Case 1a) is inevitable while case 1b) only can happen when MOESI is used. The occurrence of case 2) is already minimized in MOESI and Dragon compared to simpler protocols like MSI or MESI (here write backs occur every time another cache wants to read a modified cache line). Case 3a) causes delays in both caches. In MOESI each write access will invalidate the cache line in the other cache and in the next step it has to be reloaded which results in long delays. When Dragon is used, the updates congest the bus and they cannot be done in the background any more. The same holds for Dragon in case 3b) while this case cannot occur when MOESI is used because after the first write access the cache line is invalidated in all other caches and the cache line is not shared any more.

In conclusion one can say that using the Dragon protocol will reduce the miss rate because case 1b) will not occur. At the same time the cache access times might increase if case 3b) occurs depending on the access pattern of application.

The occurrence cases 3a) and 3b) can be minimized by software Access Classification

Table 11.2.: MOESI vs Dragon delay events [61]

	Event	Reason	MOESI	Dragon
1a)	L1 Miss	Line was never read	•	•
1b)		Line was invalidated because of bWr	•	
2)	Write back	Replaced cache line was in state *Owned* or *Modified*	•	•
3a)	Bus congestion	Two caches write to a shared cache line alternately in quick succession	• •	•
3b)		One cache writes to a shared cache line in quick succession		•

and Distribution as described in Section 11.3.

11.3. Access Classification and Distribution

When a kernel is mapped onto the CGRA the combined placer and scheduler [63] determines to which PE a memory access will be mapped. Depending on this decision the number of cache conflicts varies. If write accesses to the same cache line are mapped to different PEs, much information about write accesses has to be exchanged between the corresponding caches.

In order to minimize those cache conflicts, the scheduler tries to classify memory accesses. The accesses are then distributed to the PEs in a manner that memory accesses with the same base address are mapped to the same PE. This processes will be called ACD (Access Classification and Distribution) in the following.

ACD is done with the help of a list for each base address in the kernel. The list contains all PEs on which an access with the corresponding base address was scheduled. The following heuristic is used when binding the memory access instruction to a PE:

1. Find all memory accesses with the same base address
2. If there are more than twice as many read accesses than write accesses[3], there are no restrictions in order to exploit parallelism.
3. If not, each access in this class has to be mapped to a PE which is already in the list of the corresponding base address (if possible).
4. Update the list as follows when an access is mapped:
    - Add PE to the list if the current access is a read access
    - Add PE to the list and remove all other PEs if the current access is a write access

Note that this heuristic will improve the cache access times but at the same time this might lead to longer schedules.

[3] The factor two gives good results and was found empirically.

12. Memory Prefetching

Note: Parts of this section have already been published in [32]. The marking of self-citation is omitted in order to improve the reading flow.

Experiments have shown that a great percentage of the execution time of the CGRA is spent waiting on at least one of the caches (called *cache wait time*). In the worst memory subsystem configuration in the design space (see Chapter 16) the cache wait time was 42.8 %. This shows the need to implement a prefetching algorithm. This chapter describes the prefetching mechanism in two parts. First, the generation of prefetch requests is described and then how the *Coherence Controller* handles these requests.

The first idea for prefetching is to request the subsequent cache line (if it is not already in the cache) when a cache hit occurs [9]. This mechanism is called *Linear Prefetching* during this work (in some publications it is also called sequential prefetching). It is particularly suited for streaming applications. While *Linear Prefetching* reduces the cache wait time by 12.5 % on average over all configurations in the design space, many unnecessary prefetch requests are issued.

Many current memory prefetchers use historical data to predict the memory accesses in the future. This will lead to good results for streaming applications or other applications with regular execution. For applications with irregular execution it is harder to predict the future memory accesses.

To overcome this issue the authors of [24] proposed a *Continuous Runahead Engine* which is part of the memory controller in a multi-core system. This engine calculates memory addresses continuously ahead of time to be able to prefetch data from the memory.

In [17] the authors propose a loop aware prefetcher for conventional processor systems. Here, the program flow is taken into account to predict the memory accesses for the next loop iteration. For this purpose they define the *Cache Block Working Set* (CBWS) which is the ordered vector of all accessed cache lines in one loop. The difference of CBWS from two consecutive loop iterations gives a good prediction. The compiler marks the relevant code blocks with special instructions. These instructions control the calculation of CBWS differentials using dedicated hardware.

The authors of [78] implemented a prefetcher for CGRAs using a similar principle. They precalculate access patterns for kernels that are mapped onto the CGRA. When that kernel is executed, the prefetcher loads the corresponding pattern from a cache (*Context Directed Pattern Matching*). It is then continuously executed, evaluated and updated if necessary. While this leads to good results, it comes at the cost of additional hardware effort.

In this work the approaches of [24] and [17] are combined with partial loop unrolling and applied to CGRAs. For the innermost loops we calculate the addresses of future memory accesses using partial loop unrolling in order to request prefetches. Instead of a dedicated engine, the processing elements (PE) on the CGRA will perform the lookahead execution in parallel to the normal execution. Thus, no additional hardware is needed to find memory access patterns like in [78].

12.1. Lookahead Prefetching

Linear Prefetching assumes that the access pattern is linear for all accesses and all applications which is obviously not true. Pattern-based prefetchers try to extract patterns from previous memory accesses, which might not be accurate in the future. *Lookahead Prefetching* overcomes these issues by precalculating memory addresses of future loop iterations exactly. When the i-th iteration of a loop is executed, the memory addresses of the $i + f$-th to the $i + f + p$-th iterations are precalculated on the CGRA using existing PEs.

This means the prefetcher looks f loop iterations (*fill iterations*) into the future, and prefetches all data needed for the next p iterations (*prefetch iterations*). As described in Section 10.1.1, the innermost loops are unrolled u-times on bytecode level to increase parallelism. Now the loop is unrolled $u + f + p$-times. Then the CDFG for this code sequence is generated and the last p iterations (prefetch iterations) are modified as follows:

- All memory instructions (both store and load) are transformed to prefetch instructions.
- Find groups of prefetches that access a similar memory region assuming that all of them map to the same cache line. Delete all but one of those prefetches. (The groups are found by looking for prefetches with the same handle and similar offsets like *i+1* and *i+2*.)
- All instructions that do not produce data which is relevant for the prefetch instruction are removed.
- All original conditions are removed.
- All prefetch instructions that need data from another prefetch get a new condition: The dependent prefetch is only executed when the previous prefetch is unnecessary because the data is already present in the cache and can be used directly. This is done using the C-Box.

The previous f iterations (fill iterations) are not needed on their own. Thus, all nodes from the fill iterations that are not needed in the prefetch iterations are deleted. Figure 12.1 shows the CDFG for Code Example 7 with prefetching (marked instruction) before and after the modifications.

It is obvious that including the prefetch instructions increases the workload that has to be performed on the CGRA as more instructions have to be executed. At the same

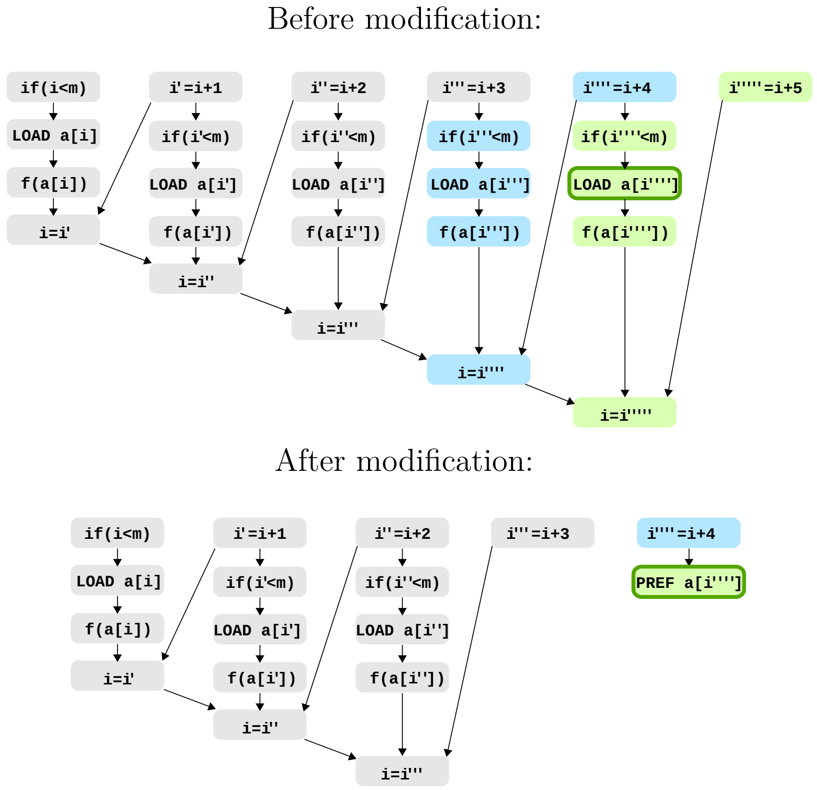


Figure 12.1.: Dependency graph of Code Example 7 with one fill iteration (blue) and one prefetch iteration (green)

time the prefetch instructions reduce the cache wait time. The higher the values of f and p, the higher is the overhead. If f is small, long cache wait times cannot be masked because the data is needed shortly after the prefetch was started. Additionally, it is likely that the data already resides in the cache because of locality. The following heuristic is used to determine values for f and p:

- $p = 0, f = 0$, if more than 40 % of all bytecodes in the innermost loop are memory instructions. Such loops will probably already use the whole memory bandwidth and prefetches bring no benefit.
- $p = 1, f = 4$, if less than 7 % of all bytecodes in the innermost loop are memory instructions. In such loops the overhead for the prefetches is relatively small, so it is beneficial to request prefetches that are further in the future.
- $p = 1, f = 0$, otherwise

All values above were found empirically and lead to good results for our benchmark set[1].

Experiments have shown that some prefetch instructions need complex offset calculations even if $f = 0$ and $p = 1$. In those cases the increased workload on the CGRA outweighs the benefit of prefetching which leads to a decreased performance. Thus, a metric for

[1] Simply setting $f = p = 0$ and increasing u will result in a worse performance because high u decrease performance as shown in [31]

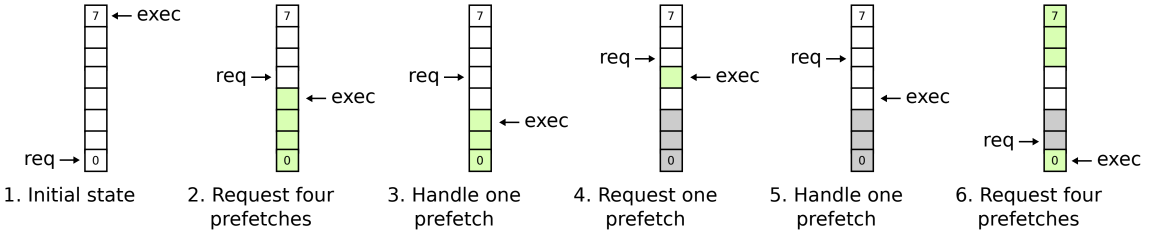


Figure 12.2.: Example for execute pointer and request pointer in the prefetch ring buffer. Green memory cells contain valid requests. Gray memory cells contain discarded requests

prefetch instructions was defined called longest prefetch path . This is the highest number of hops in the CDFG from an instruction that is not part of the prefetch or fill iteration to the prefetch instruction (via instructions of fill or prefetch iterations). All prefetch instructions with a longest prefetch path longer than a threshold are also removed (and all the instruction producing data for this prefetch). For the benchmark set used in this work a threshold of 2 gives good results.

12.2. Prefetch management

12.2.1. Storing Prefetch Requests

Each cache has a ring buffer of length 8 to store prefetch requests. Each ring buffer has a request pointer and an execute pointer. The pointers are updated as follows:

- PE requests a new prefetch:
  1. Store the request at position *request pointer*
  2. Set *execute pointer* = *request pointer*
  3. Set *request pointer* = *request pointer* + 1
     (If the ring buffer is full, the oldest value is overwritten)
- *Coherence Controller* is idle, arbiter chooses the cache and the request at position *read pointer* is valid:
  1. Execute the request at position *execute pointer*
  2. Mark request at position *execute pointer* as invalid
  3. Set *execute pointer* = *execute pointer* - 1

Figure 12.2 shows the position of execute- and request pointer after several prefetches are requested and executed. This scheme results in the following properties: 1. The newest requests are executed first. 2. If the *Coherence Controller* already executed a request in the buffer (Figure 12.2 step 3) and a new request is inserted (step 4), all older requests (marked in gray) will never be executed, because the executed request is now invalid and blocks the decrement of the execute pointer.

This behavior is desired, as prefetches are most efficient when the actual access to that memory position is far in the future so that the memory access time can be masked. The older the prefetch request is, the closer is the actual memory access. So it is beneficial to favor the newer prefetch requests.

12.2.2. Handling Prefetch Requests

When the *Coherence Controller* is idle it starts handling prefetch requests. A static priority list is used to determine the ring buffer whose request is handled[2]. Afterwards, the following steps are performed:

1. The *Coherence Controller* requests the desired data from all other L1 caches. At the same time it loads the physical address of the desired object from the *Handle Table* speculatively.
2. If one of the L1 caches can provide the data it will delivered to the requesting L1 cache and the prefetch request is finished.
3. If none of the L1 caches can provide the data, and the L2 cache is not idle (because it is for example writing data back to the DRAM) the execution of the prefetch request is aborted without delivering the data. Waiting on the L2 cache would block the *Coherence Controller* too long and actual tasks would be delayed.
4. If the L2 cache is idle the data is requested. In case of a physically addressed L2 cache the request will start as soon as the *Handle Table* provides the physical address.
5. If the L2 cache can provide the data, it will be delivered to the requesting L1 cache and the prefetch request is finished.
6. If the L2 cache can not provide the data, the *Coherence Controller* will abort the prefetch request without delivering the data. In the background the L2 cache will load the memory from the main memory.

When the *Coherence Controller* starts executing a prefetch request, the requesting cache is notified so that the cache line which will be replaced can be determined. When the data is eventually delivered, a write back request is sent to the *Coherence Controller* if necessary. The L1 cache will discard the prefetched data if the requested data already resides in the cache or if the prefetch would replace the cache line that is currently accessed by the PE it is connected to.

[2] Tests showed that a more sophisticated round robin arbiter does not improve the performance because prefetches are requested sparsely.

13. Implementation and Timing Analysis

The CGRA is not meant to be an overlay for FPGAs but will be implemented as an ASIC as described in Section 7. Still, a prototype of both the AMIDAR processor and the CGRA will be implemented on a Xilinx FPGA to show the working concept. A detailed description of the AMIDAR processor can be found in [40]. In this chapter the implementation and the timing of the memory subsystem is discussed.

13.1. L1 Cache

Figure 13.1 shows a simplified overview of the L1 Cache implementation. The data and the tag are stored in true dual ported BRAMs. One port is used by the PE to read and write single data words and one port is used by the *Coherence Controller* (CC) to deliver requested or prefetched cache lines, read cache lines requested by another L1 cache or update shared cache lines. Three parallel FSMs control both interfaces. *FSM PE* controls the PE interface and issues requests to *FSM CC OUT* which is responsible to handle outgoing requests to the CC. *FSM CC IN* handles incoming requests from

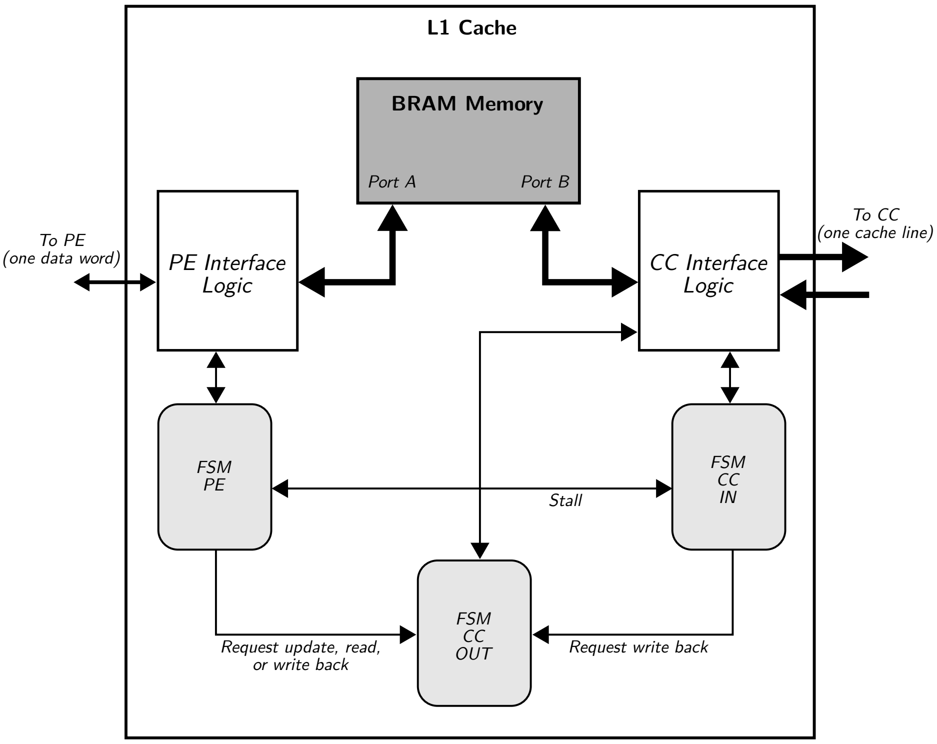


Figure 13.1.: Simplified L1 cache architecture

the CC and prefetches. It can also issue requests if a prefetch causes a write back. Those write backs always have a higher priority than all other requests by *FSM PE*. This architecture allows parallel accesses on both interfaces if different cache lines are accessed.

BRAMs have a clocked address input so that data that is written on Port A becomes visible on Port B only after two cycles (one clock cycle is needed to write it and one to read it on the other port). Thus, the execution on the second port has to be stalled if both caches access the same cache line so that no outdated data is used.

When a modified cache line has to be replaced it is buffered until the new cache line is loaded. Afterwards, a write back request is sent to the *Coherence Controller*.

13.2. L2 Cache

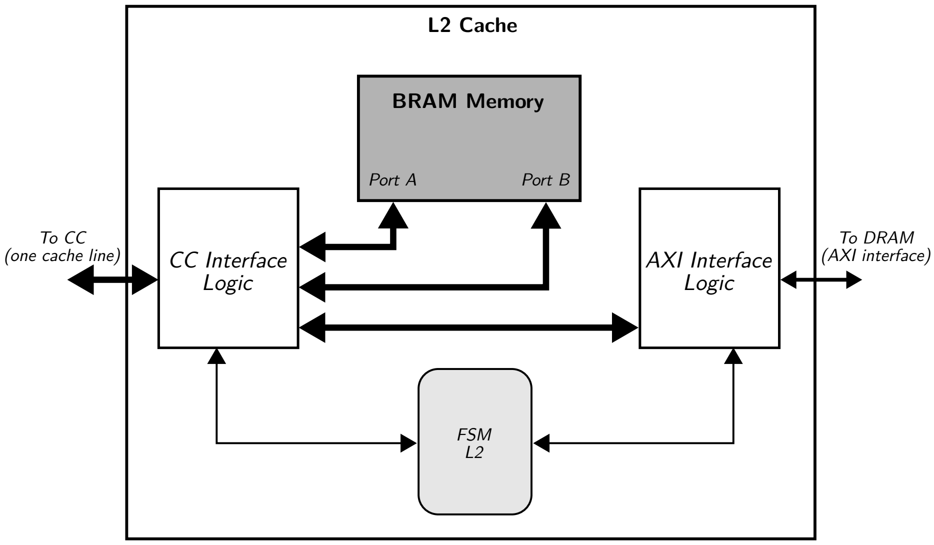


Figure 13.2.: Simplified L2 cache architecture

In the prototype a physically addressed L2 cache was implemented. The data is again stored in a true dual ported BRAM as shown in Figure 13.2. When a cache line is requested by a L1 cache, possibly two cache lines have to be loaded as described in Section 11.1. Thus, both ports are used to access both lines in parallel. The *CC interface logic* handles the alignment of L1 and L2 cache lines. The communication with the DRAM is realized via AXI bus. A single FSM controls the functionality of the L2 Cache. Thus, no parallel accesses from CC and to the DRAM are possible.

13.3. Coherence Controller

The *Coherence Controller* connects the L1 and L2 caches and arbitrates between the L1 requests. A fair arbitration between the different caches is not necessary because

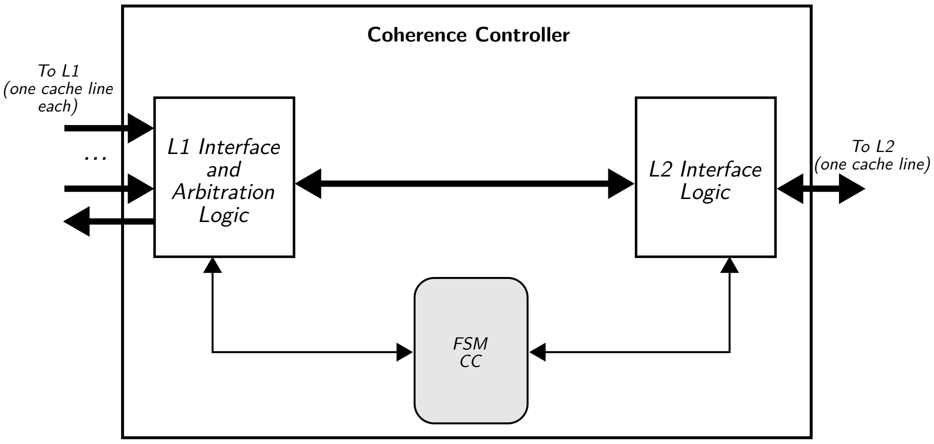


Figure 13.3.: Simplified *Coherence Controller* architecture

if at least one of the caches cannot handle the PE request directly, the whole CGRA execution will stall. Thus, the CGRA will stop its execution until all requests are handled and the order in which the requests are handled is irrelevant. In contrast to that, the class of request is important. Write back requests always have to be executed first. Otherwise, the only valid copy of that cache line lies in the write back buffer of the L1 cache which is not transparent for all other caches and outdated data will be read from the L2 cache.

13.4. Timing Analysis

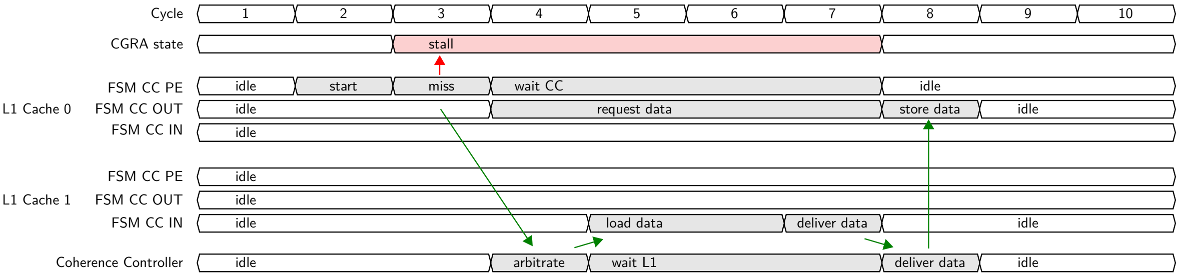


Figure 13.4.: Timing of a L1 cache miss where the desired data can be provided by another L1 cache

Figure 13.4 shows the timing of a read miss in *L1 Cache 0*. In cycle 3 the miss is detected and the CGRA is stalled. A read request is sent to the CC in the next cycle which is forwarded to all other L1 Caches. Then *L1 Cache 1* loads the desired data in the cycles 5 and 6. In cycle 8 the data is stored in the BRAM port B of *L1 Cache 0* and it is forwarded to the connected PE. The CGRA is stalled 5 cycles.

As mentioned before, the new value can only be read at BRAM Port A two cycles later. Thus, if an other access to that cache line follows, the execution has to be stalled until the data is available. Forwarding logic was omitted in this case to keep the complexity low.

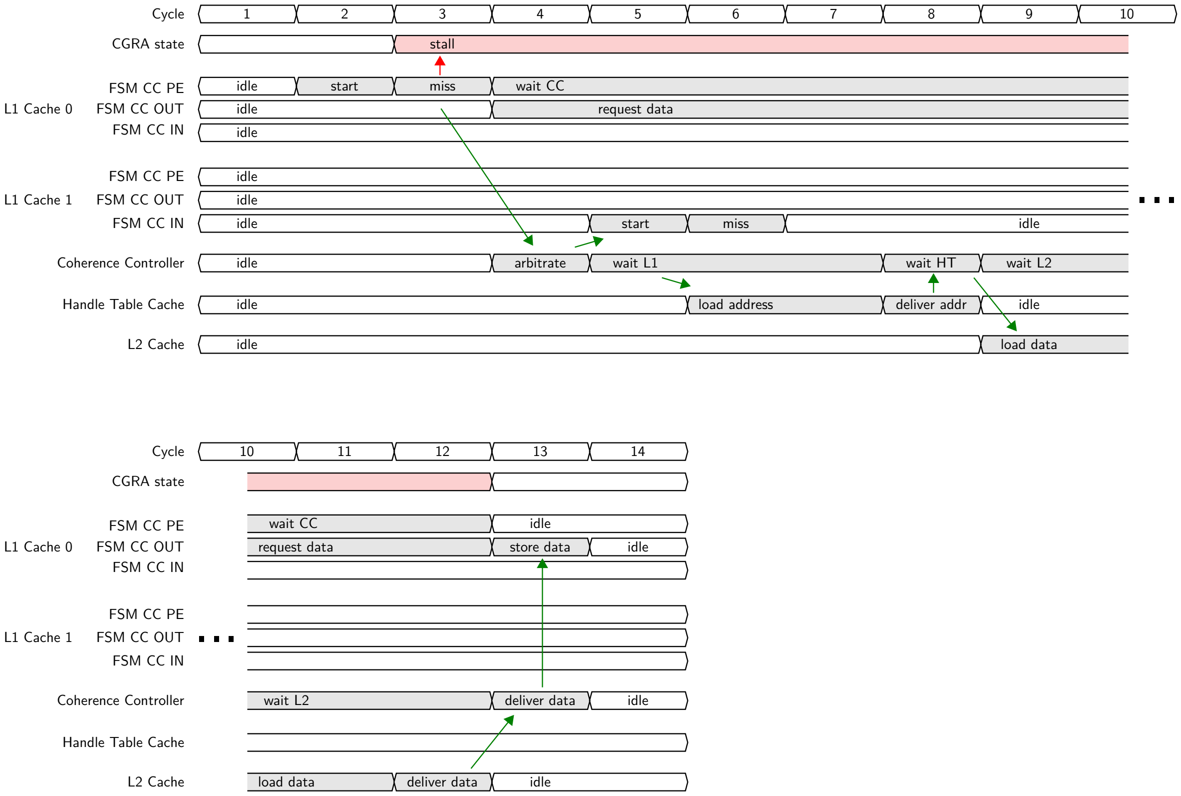


Figure 13.5.: Timing of a L1 cache miss where the desired data can be provided by the L2 cache

Figure 13.5 shows the timing when a miss on L1 occurs and the data is provided by the L2 cache. It can be seen that in time step 6 the physical address is requested from the *Handle Table* speculatively. In time step 8 the *Handle Table* is ready and the data is requested from the L2 cache. In time step 12 the data is available and it will be stored in the L1 cache in the next cycle. The CGRA is stalled 10 cycles.

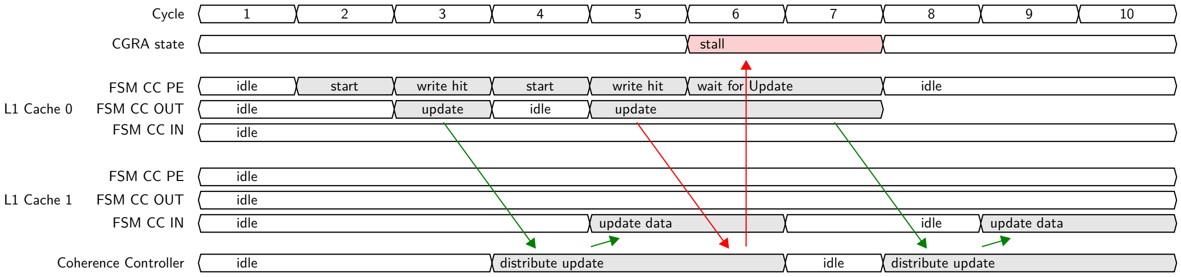


Figure 13.6.: Timing of two consecutive L1 write hits on shared cache lines

Figure 13.6 shows the timing of L1 write hits on shared cache lines. In cycle 3 *L1 Cache 0* sends an update to the *Coherence Controller* which forwards the data to all other caches. In the cycles 5 and 6 the values are updated in the background. At the same time the second write access takes palace in *L1 Cache 0*. Unfortunately, the *Coherence Controller* can not handle the update request, as it is still performing the first update request. Thus, the CGRA has to be stalled for two cycles so that no cache reads outdated data.

If the second access to *L1 Cache 0* was a read hit or a write hit to an exclusive cache line, the CGRA would not have been stalled.

Table 13.1 summarizes the cache timing for a cache system with a physically addressed L2 cache.

Table 13.1.: Cache timing summary

L1 Cache Event	**CGRA Wait time**
L1 miss Latency	
read from another L1 Cache	5 Cycles
read from L2 Cache	10+ Cycles
read from main memory	48+ Cycles
L1 miss Latency + L1 write back	
read from another L1 Cache	15 Cycles
read from L2 Cache	19+ Cycles
read from main memory	57+ Cycles
L1 update	
first of consecutive updates	0 Cycles
following updates	2 Cycles
L2 Cache Background Tasks	**Duration**
Write back of a single cache line	26 Cycles
Write back of two cache lines	51 Cycles

Part IV.

Evaluation

14. AMIDAR Simulator

In order to evaluate the memory subsystem design quickly, the existing AMIDAR Simulator was reimplemented to match the current hardware implementation [50] [69][1]. Using a software simulator gives the following advantages:

- Quick implementation times: New features can be implemented and tested easily in software without the need of a hardware design and time consuming synthesis.
- Easy debugging: Modern IDEs provide rich debugging tools which help tremendously to find bugs for example in the scheduler for the CGRA. Debugging this in Hardware directly is a very complex task.
- Easy measurements: In software it is easy to record any kind of data during the simulation. For example it is easy to track the number of unnecessary prefetches by maintaining a list of all cache lines in the cache that are prefetched and that were not read. When such a line is replaced, the counter has to be increased.
- Easy to parallelize: Several instances of the simulator can be started in parallel to increase the simulation speed when several simulations with different configurations or different applications have to be executed (see Section 14.2).

The next sections will describe the software architecture of the Simulator and how parallel sweeps are realized. Afterwards, the accuracy and the simulation speed are discussed.

14.1. Simulator Implementation

The software architecture is based on the hardware architecture. The main class `AmidarSimulator` reads configuration files and input parameters and creates and configures one or more instances of the class `Amidar`. Then the `AmidarSimulator` calls `simulate()` on the `Amidar` object(s) as shown in Figure 14.1. In `Amidar` a loop is started that only finishes when all simulated FUs are finished. One iteration of this loop corresponds to one clock cycle in the hardware. In the first step of the loop the method `tick()` is called on all FUs. The FUs will then execute their tokens and process the data that they received in the previous cycle. Optionally, a request to access the memory can be sent to the memory subsystem by FUs with DMA.

Afterwards, the method `tick()` is also called on the Bus. This class will grant bus access to one or more FUs depending on the implemented arbiter and the bus structure.

[1] The simulator will be called *Java Simulator* to avoid confusion with the commercial ModelSim simulator.

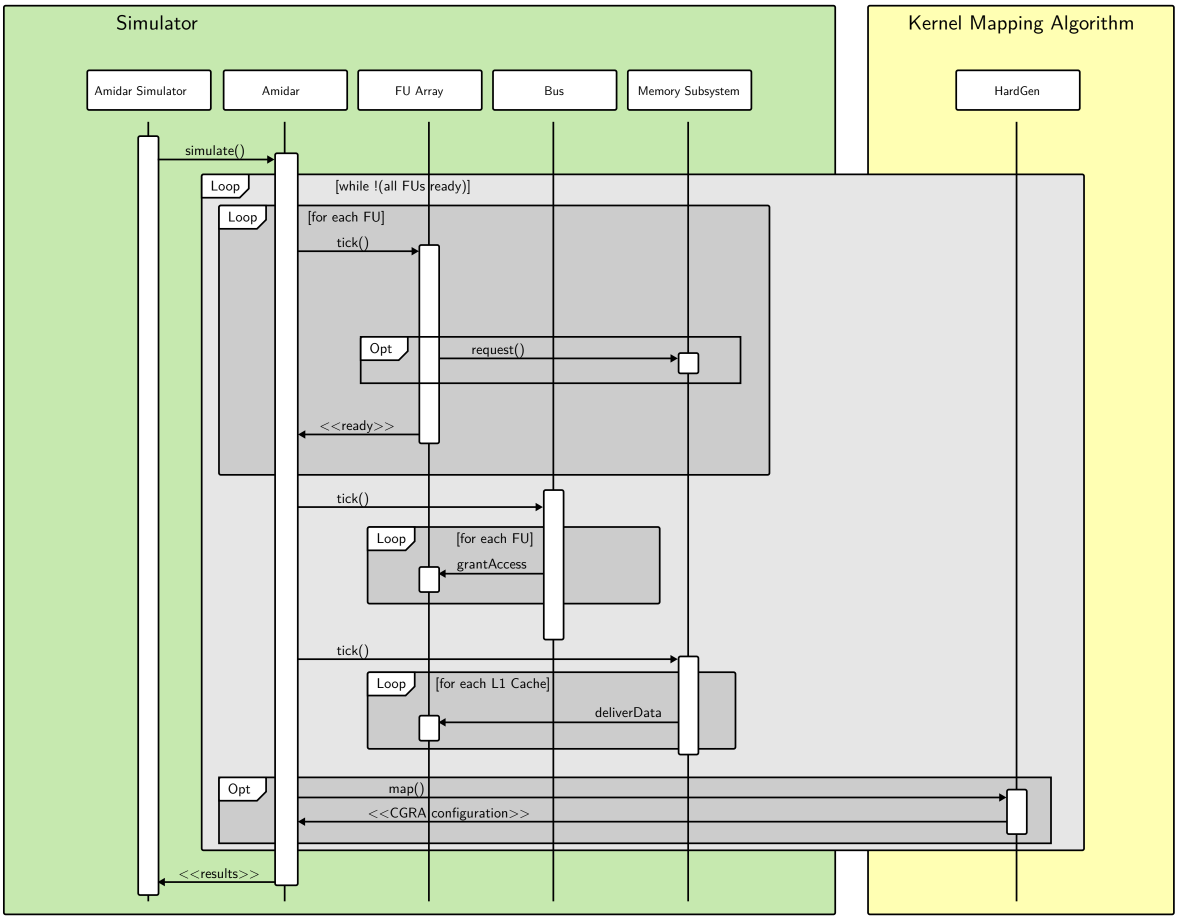


Figure 14.1.: Sequence diagramm of AMIDAR simulator

If the bus access is granted to an FU, it will directly call `setOperand(port, op, tag)` on the destination FU.

Then the method `tick()` is finally called on the memory subsystem. The memory system is simulated with all L1 caches, the *Coherence Controller*, L2 cache and the main memory. When the requested data is available, it will be delivered to the FU.

As a last step in the loop body the kernel mapping algorithm will be called optionally. It directly delivers the CGRA configuration for a kernel. Here, one important difference between hardware and simulator becomes obvious. In contrast to the hardware, the mapping algorithm is called in the simulator instead of executing it on the simulated AMIDAR processor. This has two advantages: First, the IDE debugger can be used to debug the kernel mapping algorithm. Second, not simulating the mapping step saves simulation time, which speeds up the development of the mapping algorithm. The disadvantage of this approach is that only the steady state speedup of the CGRA based accelerator can be measured with the simulator, because reconfiguration will always be done in one clock cycle in the simulator.

Abstraction Like in [55] we will distinguish between the following levels of abstraction:

1. **Statement accurate** - The behavior of a high level statement like `System.out.println("Hello world!")` is modeled correctly without regard to instructions that are executed.
2. **Instruction accurate** - The behavior and the duration of an instruction is modeled correctly. The internal state of the simulated component is not modeled. Thus, the timing of communication with other components can not be reproduced exactly, because the communication is typically handled in different phases of the instruction.
3. **Cycle accurate** - The internal state is modeled exactly so that the state of the simulator corresponds to the state of the hardware after each clock cycle. This is sufficient for this work but not always necessary.
4. **Phase accurate** - Not only the state of the processor after each clock cycle is modeled exactly but also after certain phases during one clock cycle. This is used to simulate asynchronous communication between different components.
5. **Quasi continuous** - The processor state is modeled at theoretically any time. The accuracy is limited by the time resolution.

The higher the level of abstraction the more inaccurate it gets. At the same time the simulation time decreases. Like in hardware, FUs can operate totally independent and can therefore be simulated in different levels of abstraction. Table 14.1 shows the level of abstraction for all parts of the simulator and lists possible inaccuracies.

All parts that somehow affect multiple FUs have to be simulated cycle accurately. Otherwise the timing between two accesses to shared devices like bus or *Coherence Controller* is not modeled correctly and for example stalls might be introduced wrongly. So the token generation in the *Token Machine*, all caches, the *Coherence Controller* and the bus are modeled cycle accurate. All other parts can be instruction accurate, as other parts of the processor are not affected.

However, the CGRA was implemented cycle accurate in order to be able the debug the CGRA and the mapping algorithm better.

Native methods like `System.out.println()` are only implemented statement accurate, because they are mostly only needed before a benchmark to initialize it or afterwards to print the results. Section 14.3 shows that this level ob abstraction is sufficient and still gives good simulation times.

14.2. Parallel Sweeps

As shown in previous sections, there are many parameters in the memory subsystem that have to be tuned. Thus, the design space is enormous. Evaluating all different configurations is a very time consuming task that should be automated and if possible be executed in parallel. The next section will describe how the sweep over the design space can be automated. Afterwards, a scheme to simulate different configurations in parallel on a remote server is explained.

Table 14.1.: Simulator abstraction levels

Component		Abstraction Level	Possible inaccuracies
FU	Token Machine	Mixed (token generation cycle accurate, rest instruction accurate)	• Instruction cache is assumed to be ideal
	FPU, FDIV, IALU, IDIV, IMUL, LALU	Instruction accurate	-
	Object Heap	Instruction accurate	• No Garbage Collection • Memory allocation for more than two dimensional arrays not modeled exactly • Memory Subsystem (see below)
	CGRA	Cycle accurate	• Memory Subsystem (see below)
Memory Subsystem	L1/2 Cache	Cycle accurate	-
	Coherence controller	Cycle accurate	-
	DRAM controller	Instruction accurate	• Access to the DRAM is none deterministic
Bus		Cycle accurate	-
Native Methods		Statement accurate	• Native methods will always be executed in one cycle

Automated Sweeps The class `AmidarSimulator` reads sweep configuration files in JSON format as shown in Listing 14.1. Here the sweep is three dimensional with the sizes 2, 3 and 2. Thus, there are $2 \cdot 3 \cdot 2 = 12$ different configurations that have to be simulated. All those configurations are created automatically and stored in an array with defined order. Then one or more instances of `Amidar` can be created to simulate all those cases either in parallel or sequentially. Afterwards, all measurement results are saved in an ordered list in a separate file next to a *sweepInfo* file. This file contains information which entry of the list corresponds to which configurations. Measurement results can be for example the number of clock cycles, number of cache misses or the

Listing 14.1: Sweep configuration file in JSON format

```
{
  "parameter" :
  {
    "COHERENCE_PROTOCOL" : ["DRAGON", "MOESI"],
  },
  "fu" :
  {
    "CGRA" : ["CGRA_4.json", "CGRA_9.json", "CGRA_16.json"],
  }
  "application" :
  [
    "de/amidar/cacheBench/adpcm/ADPCMencode",
    "de/amidar/cacheBench/adpcm/ADPCMdecode",
  ],
}
```

cache wait time.

Parallel Execution Attempts to parallelize the simulation on thread level did not lead to good results. Executing different simulations in separate threads only gives a limited speedup because the heap space in the JVM is quickly depleted. Another approach was to simulate each FU in a separate thread because they operate independently during one clock cycle. This even lead to a slowdown because of costly synchronization mechanisms in Java. Thus, the simulation of different configurations will be parallelized using different processes each executing one JVM. The different configurations will be transferred to those JVMs via Java Remote Method Invocation (RMI). The JVMs can even be executed on a remote machine.

Before this can be done, the AMIDAR simulator has to be installed on the remote machine and the class `AmidarRemoteManager` has to be started. Scripts to do this via SSH are provided with the simulator. During startup `AmidarRemoteManager` starts the *rmiregistry* and registers itself. Afterwards, `AmidarSimulator` can be started on the local machine. It will call `createServers` on the `AmidarRemoteManager` via RMI as shown in Figure 14.2. The remote manager then starts the desired number of remote simulators in separate JVMs.

On the local machine the simulator creates the same number of threads which directly correspond to the remote simulators. All threads synchronize on one pointer object, that points to the next configuration that has to be executed. This means that only one thread can access this pointer at once. When a thread gets access to this pointer, it will save the current pointer value and transfer the corresponding configuration to its remote simulator and start the simulation. Afterwards, the pointer is increased by one and the next thread is allowed to access the pointer object. When the remote simulator has finished the simulation it will return the results via RMI and the results are stored in the local machine in an array at the position of the saved pointer value.

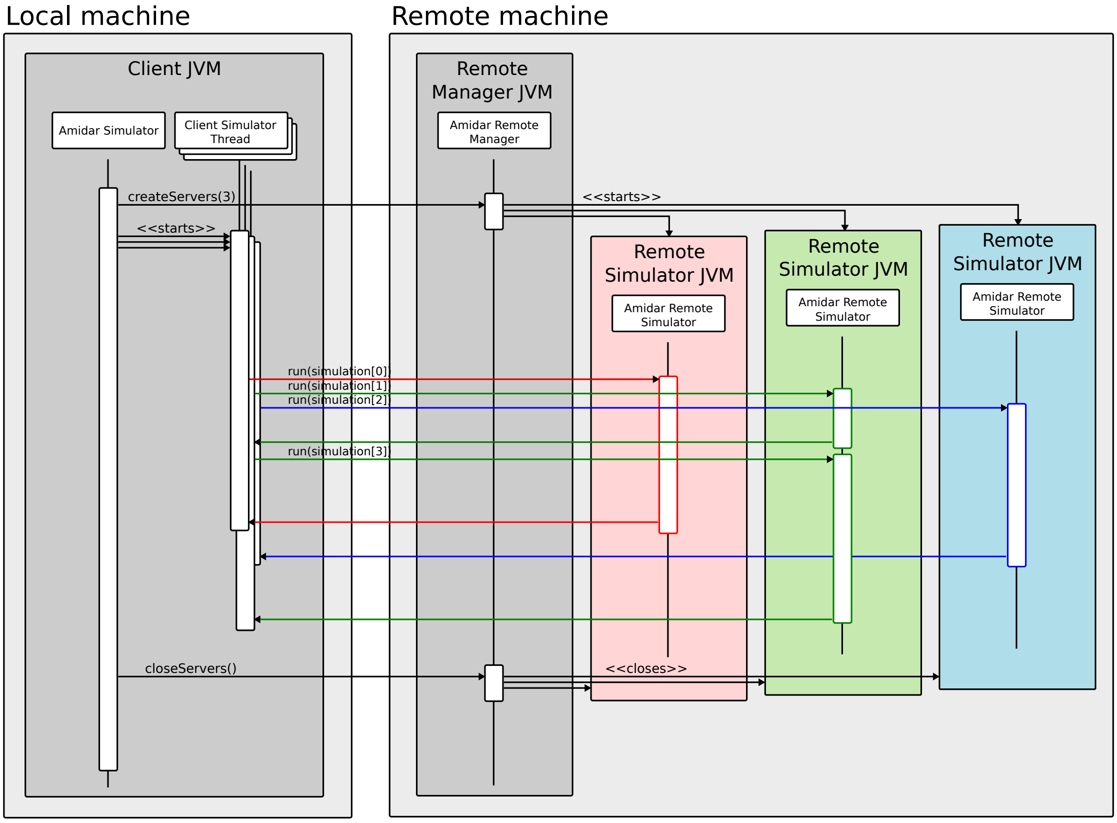


Figure 14.2.: RMI communication of the *Amidar Remote Simulator*

Then the thread immediately tries to get access to the pointer again to start the next simulation. This scheme allows to dynamically distribute the workload equally on all remote simulators without a fixed predefined scheme as shown in Figure 14.2. Here, the second remote simulator (green) finished its first simulation quickly and immediately starts the next configuration.

14.3. Performance

This section discusses the performance of the simulator in terms of accuracy and simulation speed. A common problem is, that typically first the simulator is implemented to find a good point in the design space for the hardware implementation. In order to do this, some assumption about the hardware implementation have to be made. Those assumptions are likely to be inaccurate in this early stage of development. When a potentially good point in the design space is found, the hardware is implemented. Then sometimes the assumptions have to be revoked or adjusted and it might turn out that the design point is not optimal. This was also the case in this work. First, a physically addressed L2 cache seemed to be the best solution. After the prototype was implemented, the simulator was tuned and it turned out that a virtually addressed L2 cache actually performs better. In the following the tuned simulator will be evaluated.

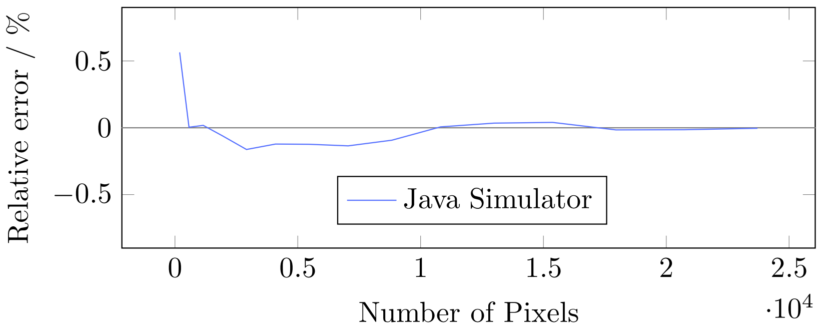


Figure 14.3.: Relative error of the simulator in number of clock cycles for sobel filter

Listing 14.2: Code to test the simulation model of the DRAM controller

```
for(int i = 0; i < a; i++) {
        ar[i] = i;
}
int cnt = 0;
for(int i = 0; i < a; i++) {
        cnt +=ar[i];
}
```

14.3.1. Accuracy

For all parts of the processor that have instruction accuracy the duration was tuned to match the hardware by analyzing execution of tokens in ModelSim. The accuracy was tested using a sobel filter shown in Appendix A. Figure 14.3 shows the relative error in the number of clock cycles for different picture sizes. Simulator and hardware use different AXT files because most peripheral devices are modeled as native methods in the simulator. This means that the same objects have different handles and physical addresses in simulation and hardware. This leads to slightly different behavior when the caches are filled. For larger pictures this effect is negligible. It can be seen that for pictures with more than $1 \cdot 10^4$ pixels the relative error is smaller than 0.05 %. For smaller pictures the relative error is slightly higher.

In order to prove that the majority of the error is caused by the DRAM controller model, a second synthetic test is used. Both loops shown in Listing 14.2 are executed on a CGRA with four PEs with mesh interconnect. Two of the PEs have DMA and an unroll factor of four is used. This benchmark will create a heavy load to the DRAM controller in order to get a deeper insight of the simulation quality of the simulation model of the DRAM controller. As a comparison the same benchmark will be simulated in ModelSim using the Verilog description of the AMIDAR processor. As in our simulator the DRAM controller is only instruction accurate due to its high complexity caused by refresh cycles in the DRAM. This gives a cycle accurate simulation of the whole processor with the only exception of the DRAM controller. Figure 14.4 shows the relative error in the number of clock cycles for our simulator for different values of `a`. Additionally this figure also shows the relative error when the Modelsim Simulator is

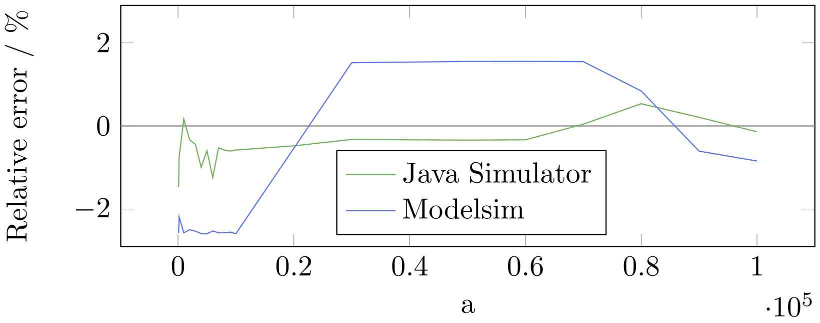


Figure 14.4.: Relative error of the simulator in number of clock cycles the application from Listing 14.2

used[2]. Two key findings back the claim that the biggest part of the error is caused by the DRAM controller model. First, in this benchmark where the DRAM controller is used heavily, the error increases. Second, the error in the Modelsim simulation is even higher than in the Java simulator and in ModelSim the DRAM controller is the only source of errors. This means that the DRAM controller that was used in Modelsim is worse than the model used in the Java simulator. The changes in the relative error at $a \approx 2 \cdot 10^4$ and $a \approx 8 \cdot 10^4$ are caused by changing DRAM access pattern because at those sizes the L1 or L2 caches respectively are completely filled. Note that prefetching combined with the non ideal DRAM controller model can cause unexpected effects. If for example the real DRAM performs a refresh when the L2 cache wants to write back some data it will be delayed. During this time no data can be prefetched from the L2 cache and the next access to that data on the L1 cache is a miss. In the simulator the refresh cycle is not modeled and the write back is not delayed. Thus, the data can be prefetched from the L2 and no cache miss occurs in the L1 cache. Thus, the non ideal DRAM controller model can result not only in small errors in runtime but also in small errors in other key figures.

Implementing a more accurate DRAM controller model is not productive because Figure 14.3 shows that the accuracy in a real applications like sobel is already sufficient for large inputs. Thus, a more accurate model would only give a small benefit at the cost of increased simulation time.

14.3.2. Simulation Speed

Measurements on an Intel i7-6700 processor with 16 GB RAM have shown that the Java simulator can simulate AMIDAR with a CGRA with 4 PEs at a speed of around $9.3 \cdot 10^5$ cycles per second while the Modelsim simulator only achieves $7.6 \cdot 10^3$ cycles per second. This means that the Java simulator runs about 122 times faster while giving a better accuracy concerning the execution time. When a remote simulator is used on a server with two AMD EPYC 7501 32-Core processors running 30 remote instances, the execution can be accelerated further by a factor of 11.9.

[2] Modelsim is theoretically able to simulate the processor quasi continuous. Still, only a behavioral simulation and no post place and route simulation was performed which results in a cycle accurate simulation.

Tests have shown the simulation speed depends on the number of PEs in the CGRA. When 16 PEs are used, the simulation speed decreases to $3.7 \cdot 10^5$ cycles per second and around 86 % of the simulation time is spent to simulate the CGRA.

In order to speed up the simulation more, it would be beneficial to implement a second CGRA model for the simulator with higher abstraction. Depending on the goal (fast design space exploration or debugging) the more appropriate model can be used for simulation.

14.4. Measurement Procedure

The aim of this work is to optimize the memory subsystem of the CGRA based accelerator. Thus, the transient behavior when the kernels are mapped to the CGRA is not relevant. Each benchmark will be executed in a short and in a long version. Then the difference of the relevant figures is taken to calculate speedups. The long versions of the benchmark starts with the exact same calculations as the short version but performs additional steps. When the short part is finished, all caches are invalidated so that the actual figures are not measured on hot caches.

15. Prerequisites

15.1. Benchmark Applications

A great variety of 24 benchmarks was used in order to find a good memory subsystem configuration for general purpose acceleration. The benchmarks can be divided into four groups: Cryptographic benchmarks, hash functions, filters and whole applications.

Cryptographic benchmarks and Hash Functions The cryptographic benchmarks used in this work are AES, DES, Blowfish, IDEA, RC6, Serpent, Skipjack, Twofish and XTEA. The benchmarks BLAKE256, CubeHash512, ECOH256, MD5, RadioGatun32, SHA1, SHA256 and SIMD512 represent the group of hash functions. Those applications typically consist of regular loops with predefined boundaries and contain lots of logic bit operations like XOR or shift. In some applications lookup tables like the S-box in AES are used. These accesses are dependent on the input data or the used cryptographic key. The input data array is read linearly in all cases. In many of the implementations status values or intermediate results are stored in object fields which are also stored in the heap.

Filter Benchmarks The filter benchmarks used in this work are ContrastFilter, GrayscaleFilter, SobelFilter and SwizzleFilter. Filters consist of regular loops with fixed loop boundaries and lots of arithmetic operations. Additionally, they contain lots of control flow which is used to handle corner cases like the end of a pixel row or an overflow in a color channel. The access to the processed picture is linear for Contrast-, Grayscale- and SwizzleFilter. In SobelFilter a two dimensional convolution is calculated. Thus, also pixels from neighboring rows are accessed.

Whole Application Benchmarks This benchmark group contains the benchmarks ADPCM de- and encoder and JPEG encoder. All three consist of several different kernels with different characteristics. The ADPCM encoder calculates an optimal prediction filter using gaussian elimination and encodes the input data using this filter. The ADPCM decoder uses this prediction filter to decode the data. Both contain data dependent loops.

The JPEG encoder contains different kernels like discrete cosine transformation, color transformation and Huffman encoding which contains data dependent control flow.

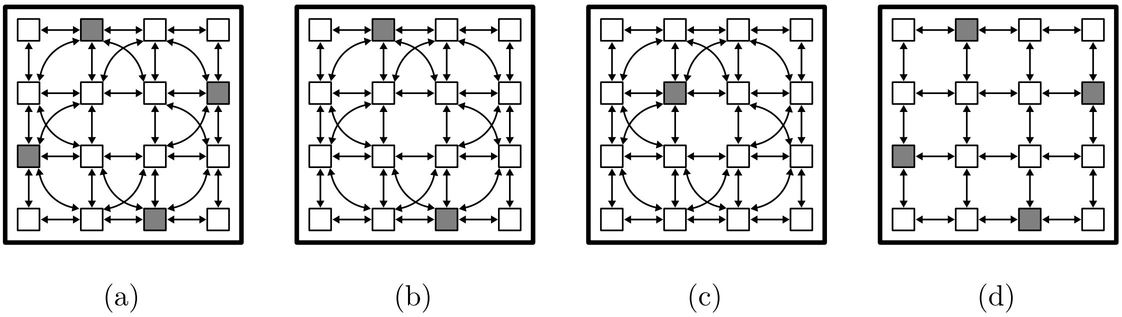


Figure 15.1.: Evaluated CGRA instances

Benchmark scale Tests have shown that the size of the processed data has a great impact on the different design options. If for example only little data is processed, prefetching brings no benefit but only generates overhead. Thus, all benchmarks were implemented scalable so that different data input sizes can be evaluated. For each benchmark the smallest input data size was defined (for example a picture with 3x3 pixels for the sobel filter). The input data size was then scaled with a factor 2^s with the $s =$ *benchmark scale*.

15.2. CGRA Comparison

The performance of the memory subsystem is also dependent on the CGRA structure itself. The CGRA instances shown in Figure 15.1 will be used to evaluate the memory subsystem design. The CGRAs have different amount of caches (gray PEs have a cache) but in sum they all have the same L1 cache size. CGRA (a) has a complex interconnect with diagonal connections and uses four 128 kB caches. The CGRAs (b) and (c) will be used to find out whether it is better to have many smaller caches (like in CGRA (a)) or less larger caches. Thus, they have the same interconnect but only two 256 kB or one 512 kB cache, respectively. CGRA (d) again has 4 caches but has only mesh interconnect.

All four CGRAs including its L1 caches, heap L1 cache with a size of 512kB and a 2 MB L2 cache are mapped to a Virtex 7[1] FPGA. The cache sizes are chosen that big in order to avoid artifacts in the BRAM usage. If the caches were smaller, all caches in CGRA (a) would use the BRAM address space only up to 70 %. The CGRAs (b) and (c) could then use the BRAM capacity more efficiently which results in higher frequencies compared to CGRA (a). In the prototype the caches will be smaller. All PEs can perform comparisons, integer addition, subtraction and multiplication. Table 15.1 shows the results.

In CGRA (a) and (d) 5 L1 caches have to be connected to the *Coherence Controller* which results in a long critical path which limits the maximum clock frequency to 115 MHz or 114 MHz respectively. In CGRA (b) only 3 L1 caches are connected so the

[1] Mapped onto XC7VX485T-2FFG1761 with Vivado 2017.2 using the design goals *Flow_PerfOptimized_high* for Synthesis and *Performance_explore* for Implementation.

Table 15.1.: Comparison of CGRAs (a) to (d)

CGRA	Max Frequency	Critical path
(a)	115 MHz	Data bus between L1 Caches
(b)	118 MHz	Data bus between L1 Caches
(c)	114 MHz	Way selection path: Cachline information BRAM output to data BRAM write enable within one L1 cache
(d)	114 MHz	Data bus between L1 Caches

critical path is shorter and the maximum frequency increases to 118 MHz. In CGRA (c) the increased cache size has an impact so that the critical path is now within one L1 cache. The maximum clock frequency is again 114 MHz.

16. Design Space Exploration

In order to find the best memory subsystem for the AMIDAR FPGA implementation a design space exploration was done with the AMIDAR simulator. Table 16.1 shows all parameters that were fixed in order to keep the size of the design space in a reasonable range. The CGRA parameters were chosen in a way that all applications can be mapped onto the CGRA and artifacts due to limited memories or registers in the CGRA are minimized. Te register file allocation for example is not yet optimized in the CGRA scheduler. Thus, sometimes kernels cannot be mapped to the CGRA because the register files in the PE are not large enough, even though it would be possible to find an allocation that fits the given size.

Prior tests have shown that 4 way L1 caches with 8 words per line give generally good results [61]. In contrast to that, the words per line in the L2 cache is not fixed, as it is expected that the words per line might influence the choice of the addressing scheme of the L2 cache.

Table 16.1.: Fixed parameters

Parameter	**Value**
L1 Cache(s) in CGRA	16 kB (sum of all caches), 8 words per line, 4 ways
L1 Cache in Heap	16 kB, 8 words per line, 4 ways
L2 Cache	256 kB, 4 ways
Context Memory	4096 Contexts
C-Box Condition Memory	128 Bit
PE register file	256 Entries

Table 16.2 shows the parameters that were tested[1]: This results in a seven dimensional

Table 16.2.: Design space parameters

Parameter	**Possible values**
Prefetching	*none*, *Linear*, *Lookahead*
Coherence protocol	*Dragon*, *MOESI*
SW supported coherence	*off*, *on*
Aliasing Speculation	*off*, *on*
CGRA instance	Figure 15.1 (a), (b), (c), (d)
L2 cache design	*physically addressed*, *virtually addressed*
Words per L2 cache line	8, 16

[1] The corresponding sweep configuration and the results can be found in Appendix D

Table 16.3.: Top 5 memory subsystems for CGRA (a)

Rank	**1**	**2**	**3**	**4**	**5**
Speedup	26.09	26.04	26.01	25.95	25.84
Prefetching	*Lookahead*	*Lookahead*	*Lookahead*	*Linear*	*Lookahead*
Coherence protocol	*MOESI*	*MOESI*	*Dragon*	*MOESI*	*Dragon*
ACD	*on*	*on*	*on*	*on*	*on*
Aliasing Speculation	*on*	*off*	*off*	*on*	*on*
L2 cache address	*virtually*	*virtually*	*virtually*	*virtually*	*virtually*
Words per L2 cache line	16	16	16	16	16

Table 16.4.: Top 5 memory subsystems for CGRA (b)

Rank	**1**	**2**	**3**	**4**	**5**
Speedup	26.35	26.04	25.88	25.77	25.73
Prefetching	*Linear*	*Linear*	*Lookahead*	*Linear*	*Linear*
Coherence protocol	*MOESI*	*MOESI*	*MOESI*	*Dragon*	*MOESI*
ACD	*on*	*on*	*on*	*on*	*on*
Aliasing Speculation	*on*	*off*	*on*	*on*	*on*
L2 cache address	*virtually*	*virtually*	*virtually*	*virtually*	*physically*
Words per L2 cache line	16	16	16	16	16

design space with 384 different configurations. All configurations have to be simulated with all 24 benchmarks with the benchmark scales 5 and 12 in order to evaluate different sizes. In total 18432 measurements have to be performed with two simulations each (a long and a short version of the benchmark) plus the baseline simulation. This was done remotely with the AMIDAR simulator on a server with two AMD EPYC 7501 32-Core processors running 30 remote simulator instances. The whole simulation took 16 hours 7 minutes and 24 seconds.

The average speedup in clock cycles over all 24 applications and benchmark scales was calculated for all configurations[2]. The values range from 19.30 for the worst configuration to 26.35 for the best configuration. The mean value is 23.04 with a variance of 2.34. The results are shown in Appendix D.

The configuration with the highest average speedup is considered the best memory subsystem configuration for the general case. The Tables 16.3 to 16.6 show the best five configurations for the CGRAs (a) to (d). The distinction between the CGRAs was done because the different CGRA structures result in different clock frequencies which is further discussed in Section 16.5.

It can be seen in Table 16.5 that CGRA (c) is a special case. It has only one PE with DMA, so ACD on the CGRA has no effect, as all memory accesses will be scheduled to the one single PE with DMA anyways.

[2] If not stated otherwise, the term speedup refers always to speedup in clock cycles.

Table 16.5.: Top 5 memory subsystems for CGRA (c)

Rank	**1**	**2**	**3**	**4**	**5**
Speedup	24.48	24.33	24.22	24.17	24.04
Prefetching	*Linear*	*Linear*	*Linear*	*Linear*	*Linear*
Coherence protocol	*MOESI*	*MOESI*	*MOESI*	*MOESI*	*Dragon*
ACD	-	-	-	-	-
Aliasing Speculation	*on*	*on*	*off*	*off*	*on*
L2 cache address	*virtually*	*physically*	*virtually*	*physically*	*virtually*
Words per L2 cache line	16	16	16	16	16

Table 16.6.: Top 5 memory subsystems for CGRA (d)

Rank	**1**	**2**	**3**	**4**	**5**
Speedup	23.14	23.07	22.02	22.91	22.87
Prefetching	*Lookahead*	*Linear*	*Lookahead*	*Lookahead*	*Lookahead*
Coherence protocol	*Dragon*	*MOESI*	*MOESI*	*Dragon*	*Dragon*
ACD	*on*	*on*	*on*	*off*	*on*
Aliasing Speculation	*on*	*on*	*on*	*on*	*off*
L2 cache address	*virtually*	*virtually*	*virtually*	*virtually*	*virtually*
Words per L2 cache line	16	16	16	16	16

For almost all shown configurations the words per line in the L2 cache is 16 and ACD is *on*. All other parameters vary, so no quick conclusion can be drawn, which configuration is the best. In the following the influence of all parameters is discussed in detail. This will be done in the following way: As it is not possible to display a 7 dimensional graph, the design space will be analyzed in different steps. In each step one or two parameters will be picked and their influence of the speedup of the accelerator is evaluated. This will be done by printing key figures like speedup or miss rate on the y-axis over all the configurations of the design subspace, that remains when the observed parameter(s) are fixed to one value. The aim is not to analyze and explain the exact behavior of the graph but rather to find general statements like "*Lookahead Prefetcher* has a higher miss rate than the *Linear Prefetcher* for all configurations in the design subspace" (see Figure 16.3). The ordering of the configuration on the x-axis varies but is always explained if necessary.

16.1. Discussion of Prefetching

Figure 16.1 shows the speedup of both *Linear Prefetcher* and *Lookahead Prefetcher* in relation to the execution without prefetcher for all configurations. In all configurations which use the CGRAs (a), (b) and (d) the *Lookahead Prefetcher* and the *Linear Prefecher* have a similar performance. Only for the CGRA (c) the *Linear Prefetcher* performs clearly better.

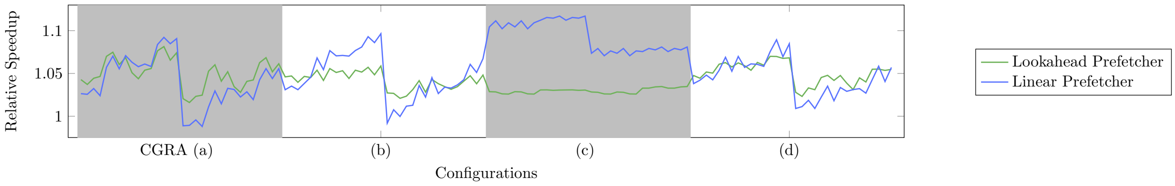


Figure 16.1.: Speedup for prefetchers relative to execution without prefetcher

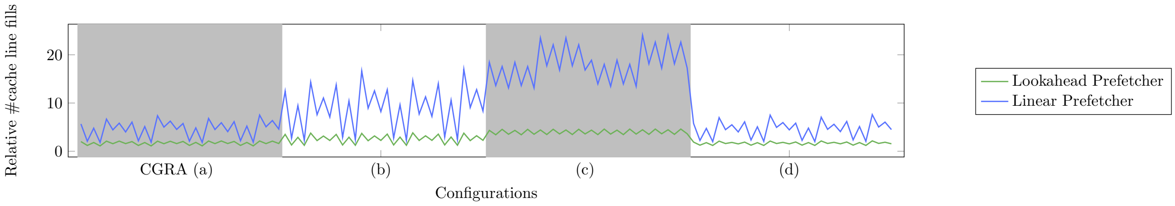


Figure 16.2.: Number of loaded cachelines for prefetchers relative to execution without prefetcher

When the *Lookahead Prefetcher* is used, the scheduler maps the prefetch instructions only on selected PEs. The prefetches are executed in a targeted way so that little duplicates of data lie in the caches before they are actually needed. In the *Linear Prefetcher* the opposite happens. Lines are prefetched less targeted and the performance decreases if there is more than one cache in the system because each cache prefetches data after a cache hit. Also, the performance of the *Linear Prefetcher* decreases when the line size of the L2 cache is 16 words because now more unnecessary data is loaded (see second half of all configuration for each CGRA).

The *Linear Prefetcher* also loads much more cache lines into the caches as shown in Figure 16.2.[3] CGRA (c) only contains one cache. This means that the load on the *Coherence Controller* is lower, so more prefetches can be performed. Every cache prefetches data when an array was accessed which results in lower miss rates (see Figure 16.3) but also in more shared cache lines when the *Linear Prefetcher* is used. More shared cache lines result in more traffic on the coherence infrastructure which leads to delays even though the miss rate is lower. This can be seen in Figure 16.4 which shows the average delay for a memory access relative to the execution without prefetcher.

It can be seen, that for CGRA (c) the average delay is better for the *Linear Prefetcher*. No traffic on the coherence structure is generated because there is only one cache in the CGRA. Thus, the better miss rate can take effect.

Conclusion One can conclude that the *Lookahead Prefetcher* has the same performance as the *Linear Prefetcher* if there is more than one cache in the CGRA. This is

[3] Figure 16.2 shows an alternating behavior. This is due to the configuration ordering. They are ordered in a way that every second configuration uses MOESI protocol and every other configuration uses DRAGON. In MOESI the relative increase in cache line fills is lower because in the baseline without prefetcher already a high amount of cache lines are filled. This is because MOESI invalidates shared cache lines when they are written in another cache and afterwards the cache line has to be filled again when needed.

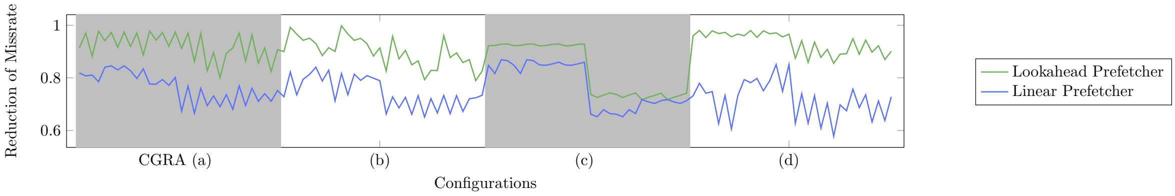


Figure 16.3.: Reduced miss rate due to prefetching relative to execution without prefetcher

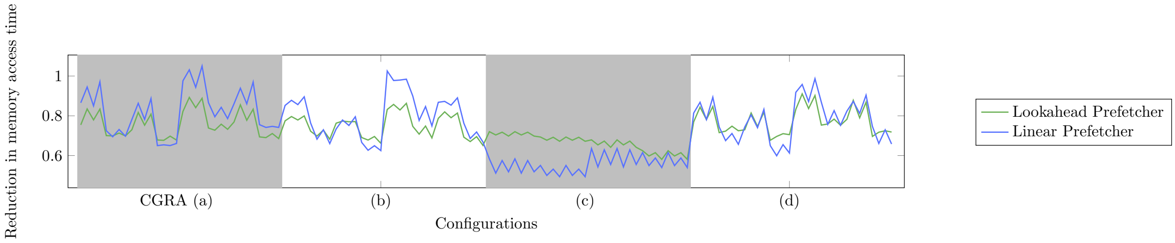


Figure 16.4.: Reduced average memory access time due to prefetching relative to execution without prefetcher

achieved through more purposeful prefetches because it looks ahead in the execution. It loads far less cache lines which will result in less energy consumption. Additionally, it requires less hardware overhead than the *Linear Prefetcher*, as most of the work is done in the mapping algorithm. The mapping time increases by 49.6 % on average. The average mapping time for all kernels in one benchmark increases from 1.9 s to 2.8 s. So the *Lookahead Prefetcher* will be considered the best option.

16.2. Discussion of Coherence Mechanisms

The choice of the Coherence protocol is closely linked with the ACD as shown in Figure 16.5. Here the speedup relative to the execution with MOESI without ACD is shown.

Again CGRA (c) is a special case. It has only one PE with DMA which means the coherence has only to be ensured between the Heap cache and the cache in the CGRA. In this case MOESI is the better Coherence protocol, because the Heap does only need the updated value when the execution on the CGRA is finished. So it is better to invalidate shared cache lines when they are written on the CGRA cache. Then the

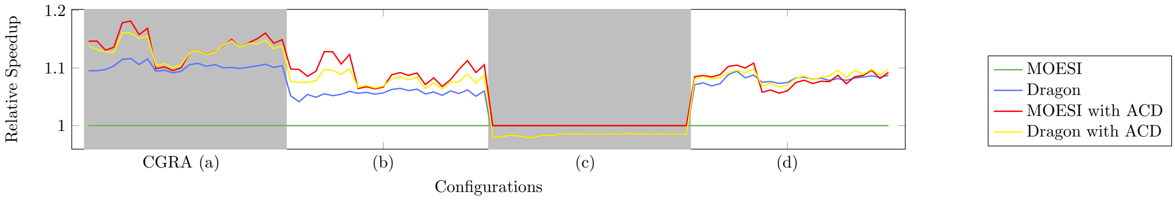


Figure 16.5.: Speedup for different coherence mechanisms relative to the execution with MOESI without ACD

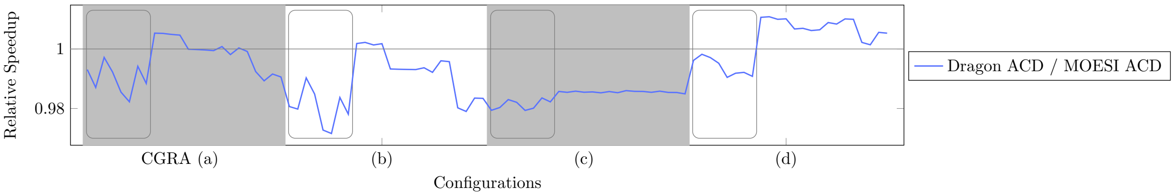


Figure 16.6.: Speedup of Dragon with ACD relative to MOESI with ACD. Gray boxes denote configurations with *Linear Prefetcher*.

CGRA holds the value exclusively and no further communication is needed with other caches. In Dragon the delay event 3b) in Table 11.2 occurs. So in a CGRA with a single cache MOESI outperforms Dragon. As mentioned before, ACD has no effect in this case.

In all other CGRAs Dragon always outperforms MOESI when ACD is deactivated, because (as mentioned in the Problem description in Section 4.1) the caches in a reconfigurable accelerator often access similar memory and share many cache lines. When two caches write to the same cache line alternately the cache line will be invalidated and fetched again which leads to high miss rates [2] and long delays (see also delay event 3a) in Table 11.2). In Dragon the cache lines are not invalidated but updated directly.

The performance of Dragon can even be improved when ACD used. Then cache line sharing is minimized so that the load on the *Coherence Controller* is decreased. The same effect can be seen when MOESI is used with ACD. A detailed comparison of Dragon with ACD and MOESI with ACD is shown in Figure 16.6. Here the speedup of Dragon with ACD in relation to MOESI with ACD is shown. For values smaller 1 MOESI with ACD is better.

It is noticeable that in the first third of all configurations of each CGRA (see gray boxes) MOESI with ACD outperforms Dragon with ACD clearly. Those configurations are the ones with *Linear Prefetcher*. This means that the *Linear Prefetcher* performs well with MOESI with ACD. As mentioned in the previous section, one disadvantage of the *Linear Prefetcher* is that it does not prefetch targeted. It always prefetches data from the memory region it just accessed and data will be prefetched in all caches and possibly the same data will be prefetched multiple times. Using ACD ensures that the data of one memory region is concentrated on one cache. Thus, the *Linear Prefetcher* can work more targeted because now only that cache will prefetch data from that region.

It can also be seen that the performance of MOESI with ACD gets better the less caches are in the CGRA (The CGRAs (a) and (d) have four caches, CGRA (b) has two caches and CGRA (c) only has one).

Still, both protocols have a similar performance. The reason for that is that ACD can be implemented efficiently and the advantage of Dragon is canceled by ACD. ACD can be implemented efficiently when working on Java bytecode because accesses to the same memory regions can be detected easily as shown in Chapter 10.2. When other executable formats are used, this is not as easy due to complex pointer arithmetic.

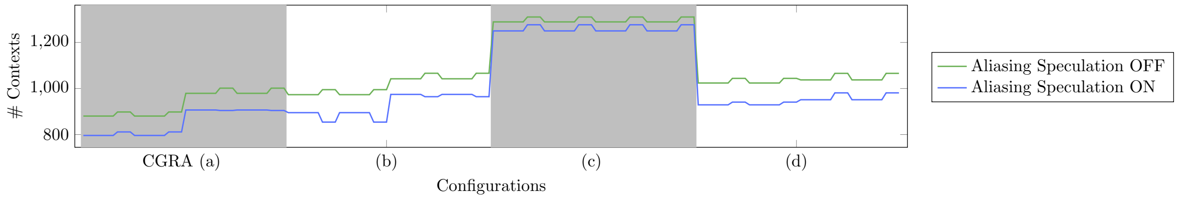


Figure 16.7.: Number of contexts per benchmark.

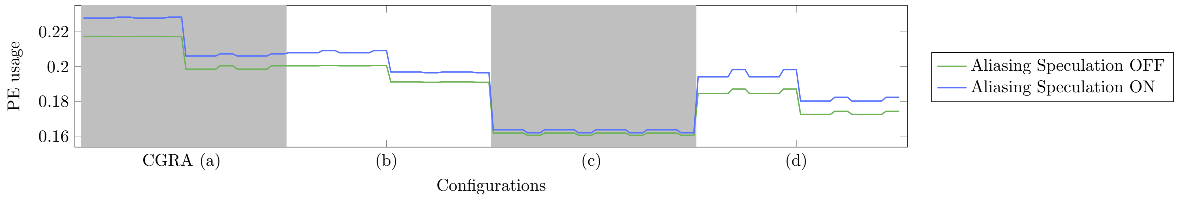


Figure 16.8.: PE usage

Conclusion For the coherence mechanisms no option is clearly superior to the others. The following statements can help to find a good solution for the selection of a mechanism.

- If no ACD is possible (because the executable format does not allow this or the cache designer has no influence on the mapping tool) it is always better to implement Dragon if there is more than one cache in the CGRA.
- If possible, ACD should always be enabled.
- If ACD is enabled, Dragon increases performance with increasing number of caches in the CGRA.
- If a *Linear Prefetcher* and ACD are used, MOESI performs better than Dragon.

In the setup described in this work ACD is possible and the *Lookahead Prefetcher* is used (see previous section). Thus, ACD should be switched on and for CGRAs with many caches Dragon should be used. For CGRAs with few caches MOESI should be used. Both Dragon and MOESI have a similar performance. Therefore, an energy evaluation (not part of this work) is necessary. The result might be that MOESI is better because updating data upon every write may result in a higher energy consumption for Dragon.

16.3. Discussion of Aliasing Speculation

Figure 16.7 and 16.8 show the number of contexts and the PE usage for the execution with and without *Aliasing Speculation*. As expected in all cases the number of contexts is decreased and the PE usage increased because now more parallelism can be exploited. Against expectation this does not lead to a speedup in every case as shown in Figure 16.9. This has two reasons. First, the reduced number of contexts does not necessarily mean that there are less executed contexts in one kernel. The inner loops have to be considered. When for example the number of contexts in the outer loop is reduced by

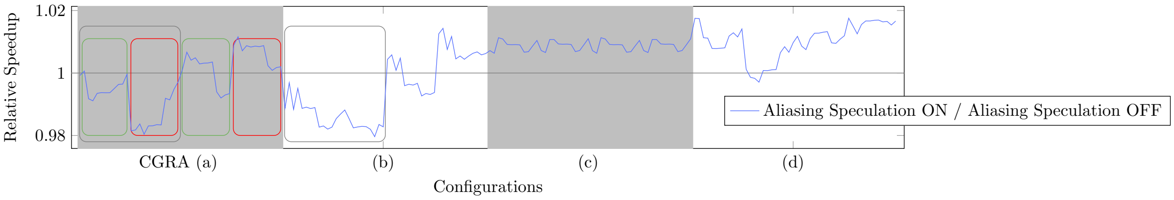


Figure 16.9.: Relative speedup of *Aliasing Speculation*

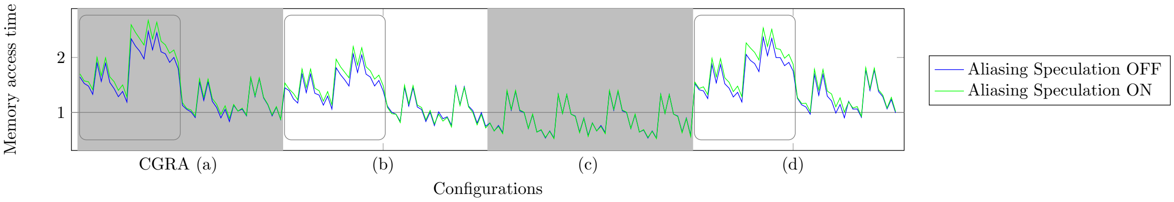


Figure 16.10.: Average memory access time

4 and the number of contexts in the inner loop is increased by 3, the total number of contexts is reduced by 1. In the likely case that the inner loop is executed n times where $n \geq 2$, the number of executed contexts is increased by $n \cdot 3 - 4$. The reason for the increased number of contexts in the inner loops is that the increased parallelism can only be exploited with some effort. Before different PEs can load values from the same array in parallel, the base address of the array has to be copied to each of these PEs. Creating these copies takes some time and in the worst case this takes place in the inner loop. The scheduler could be improved when inner loops are favored. For example the copies should be made in an outer loop and not in an inner loop.

The second reason for the unexpected behavior is, that the *Coherence Controller* gets congested. As discussed before, the average memory access time is higher for configurations without ACD because more coherence messages have to be exchanged. When *Aliasing Speculation* is switched on, the PE usage and the load on the *Coherence Controller* increase which results in an even higher average memory access time. The configurations in which this effect occurs are marked with gray boxes in Figure 16.10. For the CGRAs (a) and (b) this results in a slow down as shown in the gray boxes in Figure 16.9. For the CGRA (d) this effect is canceled because of the better inner loop scheduling.

From Table 13.1 it is known that consecutive updates lead to a delay of two cycles. When this number is decreased, the effect would be weaker. In an FPGA this could be done by using Distributed RAM instead of BRAM to store cache line information. BRAMs deliver the desired data only after one clock cycle whereas distributed RAM can deliver the data already in the same cycle. Thus, it would be possible to handle updates on the *Coherence Controller* earlier. The implications of this approach on the critical path of the cache system are not clear, because it was not tested in this work as this would broaden the scope of this work too much.

Also, an interesting fact is shown exemplary for CGRA (a) in Figure 16.9. The red boxes contain configurations with MOESI and the green boxes contain configurations with Dragon. When ACD is switched off (gray box) *Aliasing Speculation* performs

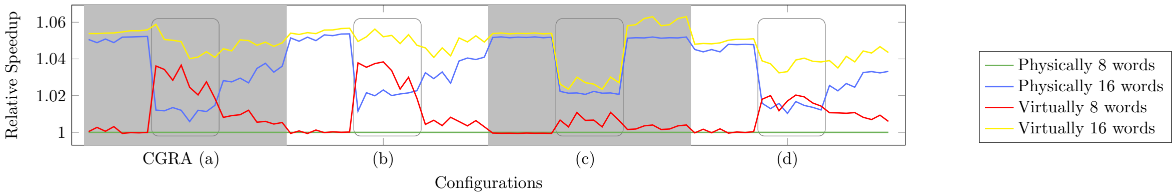


Figure 16.11.: Speedup of different L2 cache designs relative to a physically addressed L2 cache with 8 words per line

better with Dragon but when it is switched on, *Aliasing Speculation* performs better with MOESI. The reason for that lies in the load the different coherence strategies create on the *Coherence Controller*. *Aliasing Speculation* works better when there is little load and when no ACD is used, Dragon creates less load, whereas with ACD MOESI creates less load because no unnecessary message are exchanged with the heap cache.

Aliasing Speculation has practically no impact on the mapping time. The increased effort for the speculation and the handle comparison is canceled by the eased scheduling effort because of less dependencies in the graph.

Conclusion *Aliasing Speculation* allows to exploit parallelism more efficiently. Unfortunately, in the current implementation the *Coherence Controller* can not handle the load fast enough so that the system is slowed down in some cases. In addition to that, improvements can be made when the scheduler favors inner loops over outer loops.

16.4. Discussion of L2 Cache Design

Figure 16.11 shows the speedup for different L2 Cache configurations. It is obvious that a virtually addressed cache is always better than a physically addressed cache, when the caches have 16 words per line. With 8 words per line this also holds true for most cases. Only when no prefetcher is used (first third of all configurations of all CGRAs) the performance is slightly worse for some cases.

For both virtually and physically addressed caches 16 words per line give a better performance than 8 words per line.

It is interesting to note that the *Linear Prefetcher* performs better when the words per line are smaller as shown in the gray boxes. As mentioned before the *Linear Prefetcher* fetches cache lines untargeted. Thus, if the words per line is higher, a prefetch takes longer and more unnecessary data is fetched.

As already expected, the memory access times are smaller for a virtually addressed cache as shown in Figure 16.12 because the *Handle Table* has only to be accessed in case of a L2 cache miss.

Interestingly the L2 miss rate is lower for the virtually addressed caches in most cases. The reason for that is the mis-alignment of the cache lines in L1 and L2 caches as

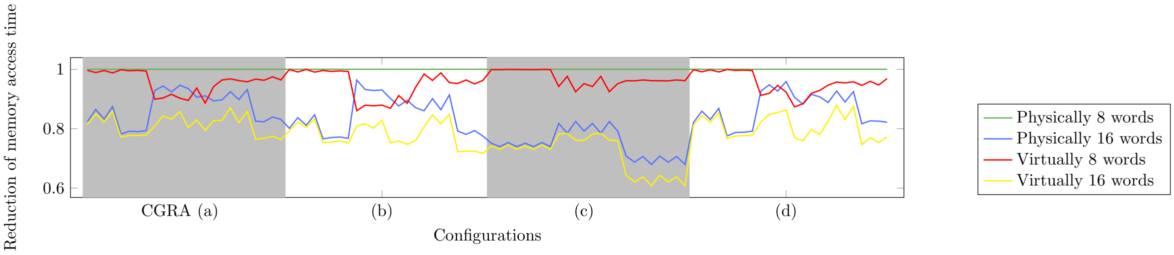


Figure 16.12.: Improvement of memory access time relative to a physically addressed L2 cache with 8 words per line

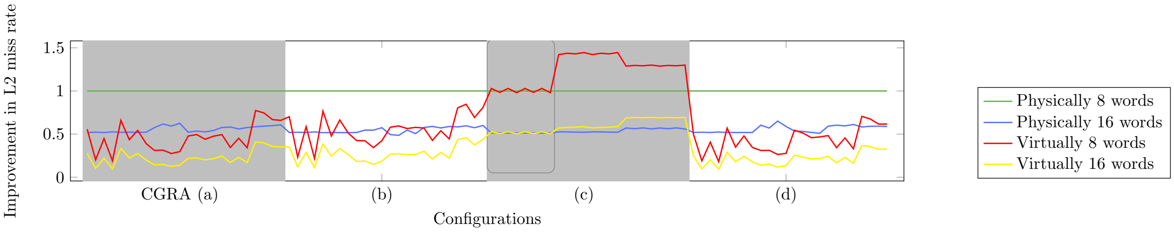


Figure 16.13.: Improvement of the read hit miss rate relative to a physically addressed L2 cache with 8 words per line

described above. When a small object is accessed it might happen that two L2 cache lines have to be loaded. This case was counted as two cache misses in this work. As a result, a virtually addressed L2 cache is better in almost all cases. In CGRA (c) where no prefetcher is used (gray box in Figure 16.13) both virtually and physically addressed cache have almost the same hit rate, because the L2 has to be accessed less frequently as the single L1 cache in this CGRA is larger than the caches in the other CGRAs. In CGRA (c) the *Coherence Controller* can handle more prefetch requests, as there is not much load on the *Coherence Controller*. Thus, when prefetchers are used, cache lines are prefetched so that small objects mostly can be accessed with one L2 cache line access in CGRA (c) and the physically addressed caches have a lower miss rate.

The number of words per line has only a minor impact on the clock frequency of the cache system. On one hand the critical path lies within in the L1 Caches as shown in Table 15.1 and on the other hand, only L1 cache lines are transferred in parallel. L2 cache lines are transferred sequentially via AXI. The index generation scheme (virtually or physically) also has only a minor impact, as it is just bit reordering. The exact influence of both factors is still to be measured.

In the future energy measurements have to be made. The physical address is loaded from the *Handle Table* speculatively and often unnecessarily. It is possible that this increases the energy consumption to much and the lookup can only be performed when it is actually necessary. In that case the performance especially of the virtually addressed cache decreases.

Conclusion If possible the words per line should be high in oder to minimize the miss rate on the L2 cache. A virtually addressed L2 cache is better as it has a higher

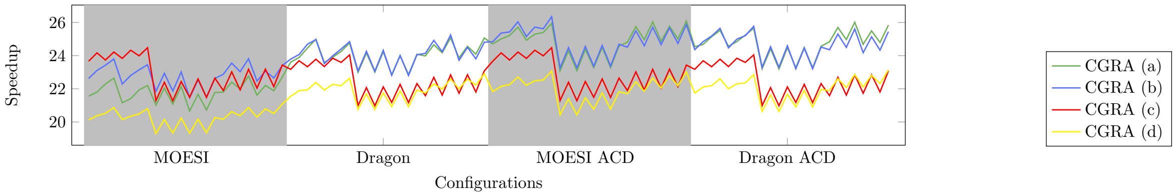


Figure 16.14.: Speedup for all four CGRAs in terms of clock cycles

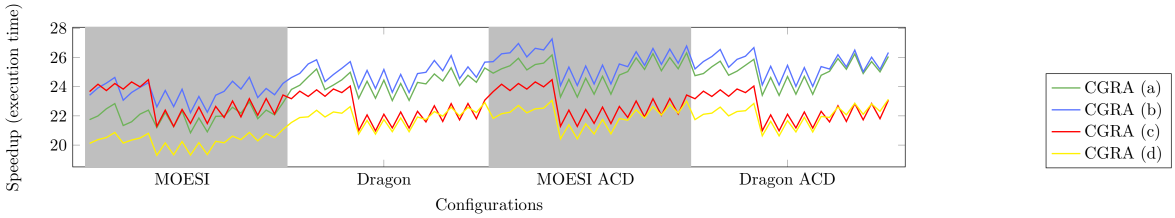


Figure 16.15.: Speedup for all four CGRAs in terms of execution time

performance and it eases the Garbage Collection in Java.

16.5. Discussion of CGRA Design

Finding the optimal CGRA design is an complex task which can only be covered briefly here. This section will concentrate only on the number of caches in the CGRA and the influence of interconnect. As it can be seen in Figures 16.8 and 16.7 that the PE usage is better and there are less contexts when there is more interconnect (compare CGRA (a) and (d)) and if there are more PEs with DMA (compare CGRA (a), (b) and (c)). In contrast to that the average memory access time increases with the number of PEs with DMA as shown in Figure 16.10 due to the delay events 3a) and b) from Table 11.2.

Figure 16.14 shows the speedup in terms of clock cycles for all CGRAs. As discussed before, CGRA (c) has only one cache. Thus, its performance is mostly independent of the coherence strategy. All other CGRAs benefit from better coherence strategies and the performance increases on average with the configurations on the x-axis. When ACD is on, the CGRAs (a) and (b) have almost the same performance in terms of clock cycles. In Figure 16.15 the clock frequencies are taken into account and the speedup in terms of execution time is shown. Here it becomes obvious that CGRA (b) gives the best performance.

A more complex CGRA structure eases the mapping effort. CGRA (a) has high interconnect and four caches which results in 1.7 s mapping time on average. CGRA (d) has also four caches but only a mesh interconnect. Thus, more copy nodes have to be inserted in the schedule and the mapping time increases to 2.0 s. The reduced number of caches in CGRA (b) and (c) also increases the scheduling effort and the mapping time is increased to 1.8 s and 2.6 s respectively.

Table 16.7.: Best memory subsystem configuration for the setup described in this work

Speedup	25.88
Prefetching	*Lookahead*
Coherence protocol	*MOESI*
ACD	*on*
Aliasing Speculation	*on*
L2 cache address	*virtually*
Words per L2 cache line	16
CGRA instance	*(b)*

Conclusion The interconnect of the CGRA has a high impact on the performance. The more connections are available, the better the performance. Allowing parallel accesses to the memory increases the PE usage but to many parallel accesses congest the *Coherence Controller* and slow the system down. In this work two caches seem to be optimal but with an improved *Coherence Controller* design, more PEs with DMA might be beneficial.

16.6. Summary

General statements can be made for ACD, L2 cache design and the prefetcher. Ideally ACD is switched on, the L2 cache is virtually addressed with 16 words per line and the *Lookahead Prefetcher* is used as it is more energy efficient.

For all other parameters no general statements can be made, as the performance of the memory subsystem strongly depends on many factors. The best configuration in terms of speedup in the setup described in this work is the fastest solution of CGRA (b) with a speedup of 26.35 on average. It uses MOESI, *Linear Prefetcher*, ACD, *Aliasing Speculation* and a virtually addressed L2 cache with 16 words per line. As mentioned earlier, the *Linear Prefetcher* achieves this by loading much more cache lines which results in a higher energy consumption. Thus, the best solution using a *Lookahead Prefetcher* will be investigated further. It can be seen in Table 16.4 that using CGRA (b) with a *Lookahead Prefetcher* also gives a very good speedup of 25.88 on average but the energy consumption will be lower.[4] Thus, this configuration will be considered the best for the current setup with 16 kB L1 caches with 4 ways and 8 words per line. The parameters are again listed in Table 16.7.

As mentioned above, this can not be generalized. If for example a specific application domain is considered or the cache implementation is improved so that updates in other caches can be done faster, the optimal setup uses the Dragon protocol. With the help of the AMIDAR simulator this was evaluated quickly[5], by reducing the time needed for a cache line update in the simulator by 2. Then, the design space exploration was

[4] Note that this configuration is better than the best configuration of CGRA (a) because this CGRA can only operate at 115 MHz

[5] This measurement was done in less than two hours including the simulation time of 80 Minutes.

Table 16.8.: Speedup for different configurations on CGRA (b) when the time needed to update cache lines is reduced by 2

Coherence Protocol	Aliasing Speculation	Speedup
Dragon	*off*	26.15
MOESI	*off*	25.72
Dragon	*on*	26.04
MOESI	*on*	25.88

repeated with ACD switched on, a virtually addressed L2 cache with 16 words per line and a *Lookahead Prefetcher*. In terms of clock cycle CGRA (a) now outperforms CGRA (b) but when the maximum clock frequencies are considered, CGRA (b) is still better. The results are shown in Table 16.8 for CGRA (b).

It can be seen that in this setup Dragon outperforms MOESI while it is better not to use *Aliasing Speculation*.

17. Results

To show that the design space exploration brings significant benefits, the solution found in the previous chapter will be compared to the worst solution using the same interconnect on the CGRA and also 16 words per line in the L2 cache (in order to have a fair comparison). This suboptimal solution uses a *Linear Prefetcher*, the MOESI protocol, no ACD, a physically addressed L2 cache and CGRA (a). During the design space exploration all benchmarks were used in two different sizes (benchmark scale 5 and 12). Now the benchmark scale 6 will be used to show that the best solution also performs well with other application sizes and no overfitting occurred during optimization. Figure 17.1 shows the speedup for all applications and both configurations. For all benchmarks but Twofish the proposed solution is better than the suboptimal solution. In Twofish not all kernels could be mapped onto the CGRA when ACD was activated because the *Condition Memory* in the C-Box was to small to hold all conditions (140 slots were required but the C-Box only contained 128). On average the best solution achieves a speedup of 25.50 while the suboptimal solution only achieves a speedup of 21.34. Thus, one can say that the optimization of the memory subsystem using the design space exploration increased the performance of the system by 22.6 % in terms of execution time (when the maximum frequencies of the CGRAs are taken into account). Figure 17.2 shows that for all benchmark the average delay (in clock cycles) caused by cache misses is reduced in all cases when the best configuration is used.

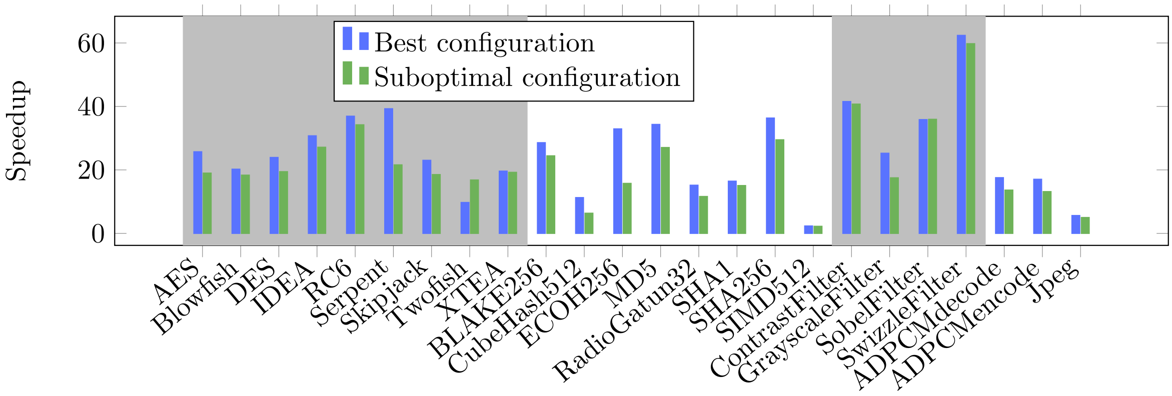


Figure 17.1.: Speedup for all benchmarks with benchmark scale 6 for the best and a suboptimal memory subsystem configuration.

In order to get an idea how the overall performance of the system improved during this work, another comparison is shown in Figure 17.3. Here the best solution is shown in comparison to a system without all the optimizations shown in in Chapter 10 and just one single L1 cache and no L2 cache. It can be seen that especially the filter applications benefit from the optimizations made during this work. On average, the

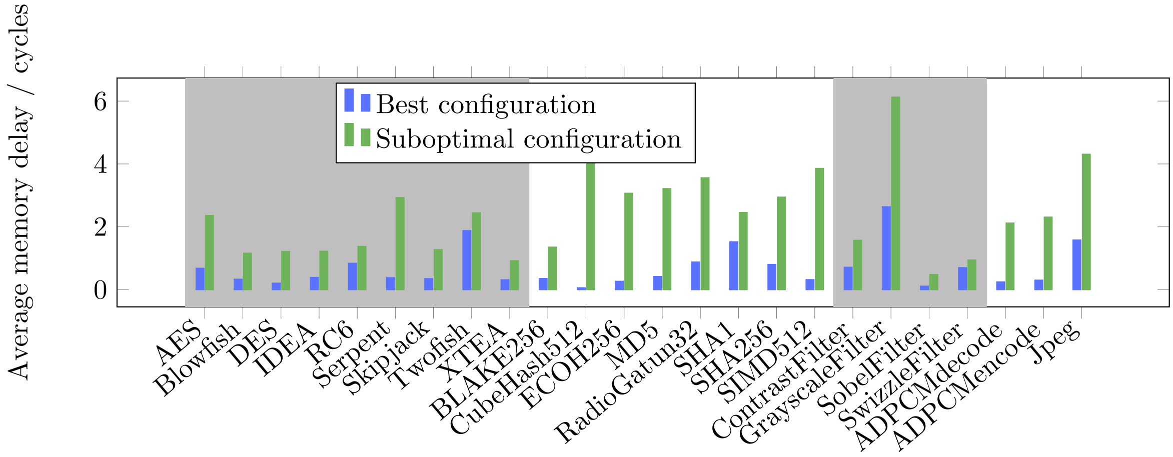


Figure 17.2.: Average delay when accessing the memory

speedup is 14.41 which means that the performance of the whole system was improved by 83.2 % in terms of execution time.[1]

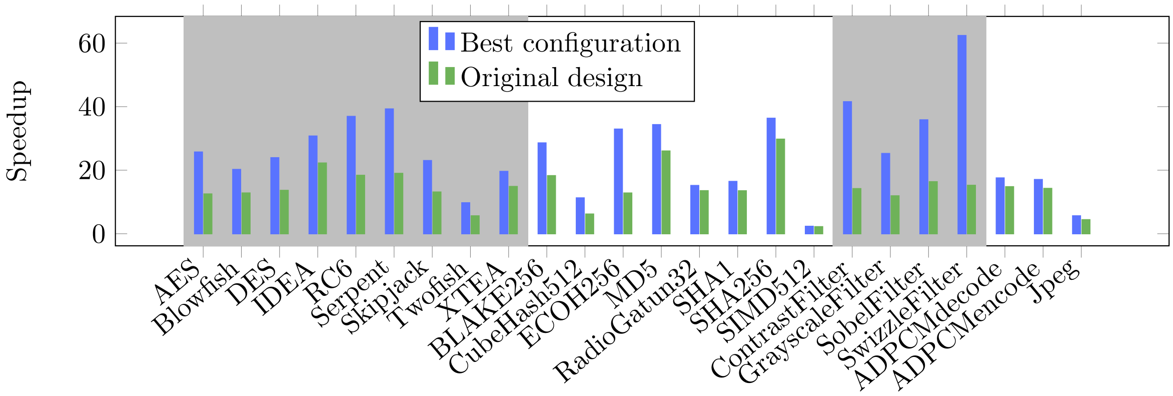


Figure 17.3.: Speedup for all benchmarks with benchmark scale 6 for the best configuration and the original design that was the starting point of this work.

17.1. Comparison With Other Approaches

Comparing the CGRA based accelerator presented in this work with other approaches is no trivial task. First, none of the other approaches can execute Java natively. Thus, all used benchmarks have to be ported and the results will also reflect language specific features (like overhead for initialization) and not only features of the system on which the software was executed. Second, many approaches like Plasticine [56] or the Layers CGRA [59] have a different design goal and concentrate on regular streaming

[1] To be exact, the original design of Döbrich [16] did not support method inlining. Thus, all method calls were inlined manually. If method inlining is also switched off and no manual inlining is done, the average speedup is only 3.61. Taking this into account, means that during this work the performance was increased by 731,2 %.

Table 17.1.: System configuration in Brandalero and Becks approach

Memory	L1 cache:	32 kB, 8-ways,	4 cycles hit latency
Subsystem	L2 cache:	256 kB, 8-ways,	12 cycles hit latency
	L3 cache:	2 MB, 16-ways,	36 cycles hit latency
CGRA	Adders	8 per level	$\frac{1}{2}$ cycle latency
	Multipliers	1 row per level	3 cycles latency
	Load Units	2 rows per level	2 cycles latency
	Store Units	1 row per level	1 cycle latency

applications. The approach that is most comparable is the work by Brandalero and Beck [7][2] which was described in Section 3.1.7. Yet, it is important to note that their work is not based on a hardware implementation but only on a simulation in Gem5 [6] in system-call emulation mode on a high level of abstraction. Numbers from a real implementation might be different. So the following comparison has to be treated with caution. A real comparison is only possible if both approaches were implemented on the same technology. Then not only the clock cycles but also the maximum clock frequency (and thus wall clock time), energy consumption and chip area could be compared.

Our system as described in Table 16.7 with the configuration found in Chapter 16 will be compared to Brandalero and Becks CGRA coupled with a dual issue superscalar processor. This processor is roughly equivalent to an ARM Cortex-A9 processor. The design parameters are given in Table 17.1. Kernels with up to five basic blocks can be mapped to the CGRA. The number of Layers was unconstrained, but 90 % of all kernels use less than 64 PEs.

Figure 17.4 shows the relative runtime for selected benchmarks[3]. Brandalero and Becks approach was executed with binaries without gcc optimizations (O0).

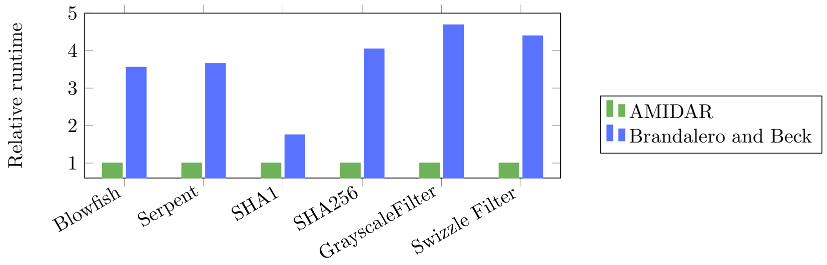


Figure 17.4.: Relative runtime for AMIDAR and Brandalero and Becks approach

When O0 is used, AMIDAR accelerated with a CGRA outperforms Brandalero and Becks approach by a factor of 3.7 on average in terms of clock cycles. This gives a good indication of the high performance of the AMIDAR system. AMIDARs advantage is that due to the C-Box and inlining, significant code sections can be mapped to

[2] Both authors offered great help and provided the measurement results reported in this section.

[3] Due to a small time frame during which this comparison could be made not all benchmarks could be ported from Java to C. From each benchmark group (with the exception of the whole applications) two benchmarks were chosen randomly.

the CGRA, while Brandaleros approach supports only five basic blocks. Yet, tests have shown that using gcc optimizations (O2 or O3) the performance of Brandaleros approach can be improved by a factor more than 3. This means that both approaches have a similar performance in that case.

This backs the results from the previous chapters: Optimizations made during compilation or the mapping process respectively (like ACD) have a great impact on the performance of the memory subsystem. Also, this gives an indication that the performance of AMIDAR can possibly be improved when there is a preprocessing done on the Java bytecode before the execution. This step could be included in the AXT conversion and would also accelerate the execution on a pure AMIDAR system without CGRA.

17.2. Prototype Implementation

Note: Parts of this section have already been published in [75]. The marking of self-citation is omitted in order to improve the reading flow.

A prototype of the whole system was implemented with the parameters shown in Table 17.2. The parameters were found using the simulator which is described in Chapter 14.

Table 17.2.: AMIDAR prototype implementation

Parameter	**Value**
L1 Cache(s) in CGRA	64 kB (sum of all caches), 8 words per line, 4 ways
L1 Cache in Heap	64 kB, 8 words per line, 4 ways
L2 Cache	256 kB, 8 words per line 4 ways, physically addressed
Context memory	1024 Contexts
C-Box Condition Memory	32 Bit
PE register file	256 Entries for gray PEs 64 for all others
Prefetching	*Lookahead*
Coherence protocol	*Dragon*
ACD	*on*
Aliasing Speculation	*off*
CGRA instance	Figure 15.1 (a)

This prototype operates at 81 MHz and is fully functional and shows the feasibility of this accelerator approach. The prototype was also used to improve the simulator further as described in Chapter 14.

17.2.1. Test Application

An HDMI controller was included in the prototype in order to show the correct execution and acceleration vividly. The test application is the calculation of the graphical representation of different Julia sets with 80x60 pixels. Afterwards, a sobel filter is applied. The resulting picture is shown on a display via HDMI.

The Julia set calculates a complex valued series of numbers in the form $Z_i = Z_{i-1}^2 + 0.7885 \cdot c$. For each pixel a different starting point Z_0 is chosen. The calculation of Z_i is finished when either $|Z_i|$ or i exceeds a threshold. The pixel is now associated with a color, dependent on the value of i. This value is read from an array. This means that the code contains data dependent loop boundaries and irregular memory accesses. When complete frame is fished, the whole calculation is restarted with a different parameter c. The value c is moved along the unit circle on the complex plane. Figure 17.5 shows the time needed per frame over the runtime.

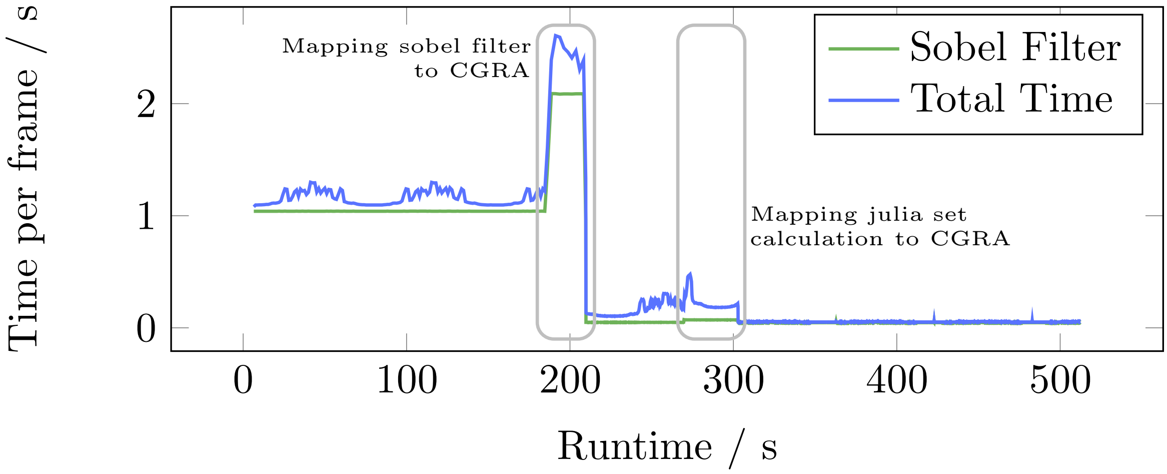


Figure 17.5.: Time for Julia set calculation and sobel filter per frame over runtime with unroll factor 1 [75]

In the first 180 seconds the whole benchmark is executed on AMIDAR. During this interval, the total time per frame (Julia set calculation + sobel filter) is 1155.2 ms on average and is shown in blue. It changes periodically because different Julia sets take different time to calculate and the parameter c is changed periodically. The time for the sobel filter (green) is independent of the content of the current frame. After 180 seconds a system thread is started. This thread reads the profiler data and the sobel filter is identified as the most time consuming kernel. Afterwards the mapping algorithm is started. The mapping time is about 21.6 seconds. In this interval the time per frame doubles, because now two threads have to be executed on the single core AMIDAR processor (first gray box). After the mapping thread is finished, the sobel filter is executed on the CGRA and the time per frame drops to 148.7 ms on average. Sixty seconds later the mapping thread is started again. Now the Julia set calculation is the most time consuming kernel. For this kernel, the mapping process takes 30.0 seconds and again the time per frame doubles (second gray box). When the mapping is finished the time per frame drops to 53.705 ms on average. This corresponds to an overall speedup of 21.5. Partially unrolling innermost loops can even increase the speedup [31]. When the unroll factor is set to 4 in this benchmark, the time needed to map the kernels to the CGRA increases to 43.1 s and 129.7 s respectively. At the same time the speedup increases to 29.0.

Table 17.3.: Results summary

Kernel	Unroll factor	Number of Nodes	Mapping Time	Speed-up	Total speedup
Sobel Filter	1	172	21.6 s	7.8	
Julia-Set	1	209	30.0 s	2.8	21.5
Sobel Filter	4	287	43.1 s	8.3	
Julia-Set	4	545	129.7 s	3.5	29,0

Table 17.3 summarizes all measurements. These results show that the presented system is able to accelerate kernels with different code structures and can lead to a speedup of almost 30. Note that the mapping software is not yet optimized for minimum runtime. It is rather programmed in a easily maintainable and extensible object oriented way, as it is still work in progress. Still, when the mapping time is evaluated it has to be taken into account that the mapping algorithm is executed on AMIDAR which runs only at 81 MHz on an FPGA. Assuming that an ASIC implementation which operates at 1 GHz is possible, the mapping time reduces by a factor of more than 12.

18. Conclusion

In Chapter 1 three goals for this work were stated.

The first goal was to implement a prototype of a processor coupled with a CGRA which fulfills the requirements shown in Table 1.1. Here it will be discussed whether these requirements were met.

- **Support of Arbitrary Applications** - This requirement was fully met. During this work a flexible interface between the host processor and the CGRA was designed. It allows the exchange of arbitrary local variables. The implemented memory subsystem is flexible and allows irregular accesses patterns.
- **Reconfiguration During Runtime** - This goal was also reached. During this work the support of Java 8 in AMIDAR was realized [49] so that the mapping process can be executed on AMIDAR without backporting. A simple interface was implemented so that the generated contexts can be transferred into the CGRA from the AMIDAR processor [66]. With the help of the profiler it is now possible to identify kernels, create a CGRA configuration and transfer this configuration to the CGRA. Then the instructions are patched so that the kernel will now be executed on the CGRA. The only open point is that the runtime of the mapping process has to be improved.
- **High performance** - Previous sections have shown that during this work the performance of the accelerator was vastly improved. This is done by an efficient memory interface with prefetching. Also, high level compiler optimizations like partial loop unrolling and speculative method inlining were applied to improve the performance. The C-Box was developed during this work with the help of [74] and [64]. This C-Box allows to execute complex control flow on the CGRA so that more kernels can be accelerated.
- **Low Programming Effort** - The accelerator works totally transparent for the programmer. No knowledge about the hardware is needed and arbitrary Java code can be accelerated.

In summary this means that the prototype fulfills all requirements.

The second goal was to implement an accurate and fast simulator. In Chapter 14 it is shown that the simulator is highly accurate. It has a high simulation speed which is two orders of magnitude faster than the ModelSim simulator which simulates an RTL description of the processor. Still, the simulation speed is not optimal and performance degrades when a CGRA with more PEs is simulated. Yet, it is possible to execute several simulations in parallel on a remote machine automatically, in order to speed up parameter sweeps.

The third and last goal was to use the simulator for a design space exploration. Chapter 16 discusses the results thoroughly. The optimal configuration was found for the given setup. Also, it was shown that this procedure is repeatable and it is possible to evaluate a different setup quickly by adapting the key parameters in the simulator. For several parameters the optimal value was found whereas for some parameters only guidelines could be given depending on the boundary conditions of the system.

All goals were achieved but still there is much room for improvement. These points will be discussed in the next section.

18.1. Open Points and Future Work

Energy Consumption Model The first and most interesting point is to evaluate the energy consumption. The design space exploration was mainly done only with regard to the execution time. It is possible that some design choices made during this work bring a slight speedup while increasing the energy consumption enormously. For example Dragon and MOESI give a similar performance when ACD is activated but it is not clear whether updating all shared cache lines in Dragon might result in a higher energy consumption. The same holds for *Aliasing Speculation*. Thus, energy measurements are required to tune and extend the existing energy consumption model in the AMIDAR simulator in order to find the best configuration not only in terms of speedup but also in terms of energy efficiency.

Improved Cache Design and ASIC Implementation As mentioned previously, the time to update cache lines is to high and leads to congestion when the Dragon protocol is used. Thus, the caches should be improved so that they use distributed RAM instead of BRAMs on the FPGA. Then is is possible to get information about the cache line one cycle earlier and the other caches can be notified quicker. As mentioned before, the CGRA is meant to be implemented on an ASIC. Thus, this problem will be solved when using the appropriate memory implementation in an ASIC design of AMIDAR coupled with a CGRA.

Potential in the Mapping Process The mapping process can still be optimized in several ways. First, the execution time can be improved as the code was written in an object oriented way in order to be readable and well maintainable. Furthermore, there are several open point that will increase the performance of the whole system further:

1. It should be possible for the scheduler to decide whether a write access to a shared cache line is an exclusive write which will invalidate the cache line in all other caches or a shared write which will provide the updated value to the other caches. This could combine the benefits from both coherence protocols Dragon and MOESI. Also, a solution of mixed caches is thinkable, where for example the heap cache uses MOESI and invalidates all shared cache lines when another cache writes that line, while all caches in the CGRA use Dragon and update the

cache line. This way the CGRA benefits from quick updates while unnecessary communication with the heap cache is blocked.

2. The unrolling process described in this work is very lightweight and efficient but still some improvements are possible. For example it is possible to unroll also outer loops. This will result in multiple versions of the inner loops which can be merged into a single loop in some cases. This would result in a light version of polyhedral memory access optimization.

   Additionally, the profiler of the AMIDAR processor could be extended so that the number executed loop iterations is recorded. With this information it would be possible to find an optimal unroll factor for each loop individually.

   Also other optimizations like loop tiling are thinkable.

3. ACD gives good performance but takes not into account what memory regions were accessed in the kernel previously executed. It is possible that two kernels access the same memory regions but in each kernel the accesses to that region are mapped to different caches. By implementing a global scheme the performance of ACD might be improved.

General Improvements Future work also includes the optimization of the CGRA structure. Implementations with pipelined ALUs and overlapping operations are thinkable. Also the scheduler itself can be improved with techniques like modulo scheduling or prioritization of inner loops.

Including more than one CGRA in one processor gives the possibility to accelerate different threads in parallel on different CGRAs.

Finally, efficient hand optimized libraries for common problems like FFT should be provided. Early tests have shown that this leads to substantial speedups [36].

18.2. Summary

During this work the worlds first prototype of a reconfigurable processor-CGRA system was implemented. It accelerates itself autonomously by reconfiguring the CGRA and using it as a general purpose hardware accelerator. The theoretical basics were described in Döbrichs work [12], but especially the interfaces between the host processor and the CGRA were only covered on a very high abstraction level. Thus, much effort was put in the design of the interface used to exchange *Live-In/Out* variables. Also, the interface to send the configuration (contexts) from the host processor to the CGRA was designed during this work.

The original concept of this accelerator could only handle simple memory accesses to one dimensional arrays which were completely loaded into the CGRA prior to the execution. This proved to be very inefficient for some kernels that work only on a few elements of a large array. Thus, the focus of this work was to implement and optimize an efficient and flexible memory interface for the CGRA based accelerator. The result was a multi cache system that allows the CGRA to load arbitrary data in parallel from the memory without prior knowledge of any access patterns. Several strategies to improve the performance of the memory subsystem were presented. Those strategies are not limited to the hardware level but also strategies on the software level were presented. This means that the algorithm that maps a kernel to the CGRA optimizes the scheduling and binding of memory accesses to increase the performance.

It is not intuitively clear whether combining all the presented strategies deliver a good performance. Thus, the fast and accurate simulator for the whole system was implemented to do a design space exploration in order to find the best set of strategies. The solution that was found exceeds the performance of a naive configuration of the multi cache system by 22.6 % in the current setup. General statements cannot be made about all presented strategies because to many factors have an influence on the performance of the design. Yet, this work shows a way how to find a very good configuration for a given system quickly and provides powerful tools like an accurate and fast simulator.

The implementation and optimization of the multi cache system itself and all optimizations that were developed during this work (like partial loop unrolling) led to a speedup of more than 83.2 % percent compared to the work originally presented in [12].

Bibliography

[1] A. Aiken and A. Nicolau. “Optimal Loop Parallelization”. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation.* PLDI ’88. Atlanta, Georgia, USA: ACM, 1988, pp. 308–317. ISBN: 0-89791-269-1. DOI: `10.1145/53990.54021`.

[2] Patrick Appenheimer. “Cache Management für CGRAs mit DMA”. Bachelors Thesis. Computer Systems Group: TU Darmstadt, 2016.

[3] James Archibald and Jean-Loup Baer. “Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model”. In: *ACM Trans. Comput. Syst.* 4.4 (Sept. 1986), pp. 273–298. ISSN: 0734-2071.

[4] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. “PACT XPP — A Self-Reconfigurable Data Processing Architecture”. In: *The Journal of Supercomputing* 26.2 (Sept. 2003), pp. 167–184.

[5] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro. “Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications”. In: *2008 Design, Automation and Test in Europe.* Mar. 2008, pp. 1208–1213. DOI: `10.1109/DATE.2008.4484843`.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. “The Gem5 Simulator”. In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: `10.1145/2024716.2024718`.

[7] M. Brandalero and A. C. S. Beck. “A Mechanism for energy-efficient reuse of decoding and scheduling of x86 instruction streams”. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017.* Mar. 2017, pp. 1468–1473. DOI: `10.23919/DATE.2017.7927223`.

[8] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou. “A Fully Pipelined and Dynamically Composable Architecture of CGRA”. In: *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines.* May 2014, pp. 9–16. DOI: `10.1109/FCCM.2014.12`.

[9] F. Dahlgren and P. Stenstrom. “Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors”. In: *TPDS* 7.4 (Apr. 1996), pp. 385–398. ISSN: 1045-9219.

[10] G. Dimitroulakos, M. D. Galanis, and C. E. Goutis. “Alleviating the data memory bandwidth bottleneck in coarse-grained reconfigurable arrays”. In: *2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP’05).* July 2005, pp. 161–168. DOI: `10.1109/ASAP.2005.12`.

[11] Stefan Döbrich and Christian Hochberger. “Effects of Simplistic Online Synthesis in AMIDAR Processors”. In: *ReConFig*. 2009, pp. 433–438.

[12] Stefan Döbrich and Christian Hochberger. “Exploring online synthesis for CGRAs with specialized operator sets”. In: *International Journal of Reconfigurable Computing* 2011 (2011), p. 10.

[13] Stefan Döbrich and Christian Hochberger. “Low-Complexity Online Synthesis for AMIDAR Processors”. In: *International Journal of Reconfigurable Computing - Selected Papers from ReconFig 2009 International Conference on Reconfigurable Computing and FPGAs (ReconFig 2009)* 2010 (2010).

[14] Stefan Döbrich and Christian Hochberger. “Low-Complexity Online Synthesis for AMIDAR Processors”. In: *International Journal of Reconfigurable Computing* 2010 (2010), p. 15.

[15] Stefan Döbrich and Christian Hochberger. “Towards Dynamic Software/Hardware Transformation in AMIDAR Processors”. In: *it - Information Technology* 50.5 (2008), pp. 311–316.

[16] Stefean Döbrich. “Performance Improvement of Adaptive Processors - Hardware Synthesis, Instruction Folding and Microcode Assembly”. PhD thesis. Dresden University of Technology, 2012.

[17] A. Fuchs, S. Mannor, U. Weiser, and Y. Etsion. “Loop-Aware Memory Prefetching Using Code Block Working Sets”. In: *2014 MICRO*. Dec. 2014, pp. 533–544.

[18] S. Gatzka and C. Hochberger. “Hardware Based Online Profiling in AMIDAR Processors”. In: *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. Apr. 2005, 144b–144b. DOI: 10.1109/IPDPS.2005.239.

[19] Stephan Gatzka and Christian Hochberger. “On the Scope of Hardware Acceleration of Reconfigurable Processors in Mobile Devices”. In: *HICSS*. 2005, p. 299.

[20] Stephan Gatzka and Christian Hochberger. “The AMIDAR Class of Reconfigurable Processors”. In: *The Journal of Supercomputing* 32.2 (2005), pp. 163–181.

[21] James Gosling. *The Java Language Specification, Java SE 8 Edition (Java Series)*. Addison Wesley Professional, 2014.

[22] V. Govindaraju, C. H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. “DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing”. In: *IEEE Micro* 32.5 (Sept. 2012), pp. 38–51. ISSN: 0272-1732. DOI: 10.1109/MM.2012.51.

[23] V. Govindaraju, C. H. Ho, and K. Sankaralingam. “Dynamically Specialized Datapaths for energy efficient computing”. In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. Feb. 2011, pp. 503–514. DOI: 10.1109/HPCA.2011.5749755.

[24] M. Hashemi, O. Mutlu, and Y. N. Patt. “Continuous runahead: Transparent hardware acceleration for memory intensive workloads”. In: *2016 MICRO*. Oct. 2016, pp. 1–12.

[25] Chen-Han Ho. “Mechanisms Towards Energy-Efficient Dynamic Hardware Specialization”. PhD thesis. University of Wisconsin-Madison, 2014.

[26] C. Hochberger, L. J. Jung, A. Engel, and A. Koch. “Synthilation: JIT-compilation of microinstruction sequences in AMIDAR processors”. In: *Proceedings of the 2014 Conference on Design and Architectures for Signal and Image Processing.* Oct. 2014, pp. 1–6. DOI: 10.1109/DASIP.2014.7115634.

[27] C. H. Hoy, V. Govindarajuz, T. Nowatzki, R. Nagaraju, Z. Marzec, P. Agarwal, C. Frericks, R. Cofell, and K. Sankaralingam. “Performance evaluation of a DySER FPGA prototype system spanning the compiler, microarchitecture, and hardware implementation”. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* Mar. 2015, pp. 203–214. DOI: 10.1109/ISPASS.2015.7095806.

[28] B. Janßen, P. Zimprich, and M. Hübner. “A dynamic partial reconfigurable overlay concept for PYNQ”. In: *2017 27th International Conference on Field Programmable Logic and Applications (FPL).* Sept. 2017, pp. 1–4. DOI: 10.23919/FPL.2017.8056786.

[29] Amit D. Joshi, Satyanarayana Vollala, B. Shameedha Begum, and N. Ramasubramanian. “Performance Analysis of Cache Coherence Protocols for Multi-core Architectures: A System Attribute Perspective”. In: *Proceedings of the International Conference on Advances in Information Communication Technology & Computing.* AICTC ’16. Bikaner, India: ACM, 2016, 22:1–22:7. ISBN: 978-1-4503-4213-1. DOI: 10.1145/2979779.2979801.

[30] L. J. Jung and C. Hochberger. “Optimal processor interface for CGRA-based accelerators implemented on FPGAs”. In: *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig).* Nov. 2016, pp. 1–7. DOI: 10.1109/ReConFig.2016.7857178.

[31] Lukas Johannes Jung and Christian Hochberger. “Feasibility of High Level Compiler Optimizations in Online Synthesis”. In: *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on.* Dec. 2015, pp. 1–7.

[32] Lukas Johannes Jung and Christian Hochberger. “Lookahead Memory Prefetching for CGRAs Using Partial Loop Unrolling”. In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications.* Ed. by Nikolaos Voros, Michael Huebner, Georgios Keramidas, Diana Goehringer, Christos Antonopoulos, and Pedro C. Diniz. Cham: Springer International Publishing, 2018, pp. 93–104. ISBN: 978-3-319-78890-6.

[33] Matthias Jung and Norbert Wehn. “Driving Against the Memory Wall: The Role of Memory for Autonomous Driving”. In: *Workshop 23.03. 2018: New Platforms for Future Cars: Current and Emerging Trends at IEEE Conference Design, Automation and Test in Europe (DATE).* 2018.

[34] Yongjoo Kim, Jongeun Lee, Toan X. Mai, and Yunheung Paek. “Improving Performance of Nested Loops on Reconfigurable Array Processors”. In: *ACM Trans. Archit. Code Optim.* 8.4 (Jan. 2012), 32:1–32:23. ISSN: 1544-3566. DOI: 10.1145/2086696.2086711.

[35] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, Jonghee Yoon, and Yunheung Paek. “Memory-Aware Application Mapping on Coarse-Grained Reconfigurable Arrays”. In: *High Performance Embedded Architectures and Compilers.* Ed. by Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 171–185. ISBN: 978-3-642-11515-8.

[36] Stefan Knipp and Ramon Wirsch. “Handoptimized DSP-Kernel for the Amidar CGRA”. Project Seminar. Computer Systems Group: TU Darmstadt, 2017.

[37] T. Kojima, N. Ando, H. Okuhara, and H. Amano. “Glitch-aware variable pipeline optimization for CGRAs”. In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig).* Dec. 2017, pp. 1–6. DOI: 10.1109/RECONFIG.2017.8279797.

[38] Hongsik Lee, Dong Nguyen, and Jongeun Lee. “Optimizing Stream Program Performance on CGRA-based Systems”. In: *Proceedings of the 52Nd DAC.* DAC ’15. San Francisco, California: ACM, 2015, 110:1–110:6. ISBN: 978-1-4503-3520-1.

[39] Ming-Hau Lee, Hartej Singh, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, Eliseu M.C. Filho, and Vladimir Castro Alves. “Design and Implementation of the MorphoSys Reconfigurable Computing Processor”. In: *Journal of VLSI signal processing systems for signal, image and video technology* 24.2 (Mar. 2000), pp. 147–164. ISSN: 0922-5773. DOI: 10.1023/A:1008189221436.

[40] Changgong Li. “Implementation of an AMIDAR based Java Processor”. PhD thesis. Technische Universität Darmstadt, 2019.

[41] Roman Lysecky, Greg Stitt, and Frank Vahid. “Warp Processors”. In: *Proceedings of the 41st Annual Design Automation Conference.* DAC ’04. San Diego, CA, USA: ACM, 2004, pp. 659–681. ISBN: 1-58113-828-8. DOI: 10.1145/996566.1142986.

[42] Roman Lysecky and Frank Vahid. “Design and Implementation of a MicroBlaze-based Warp Processor”. In: *ACM Trans. Embed. Comput. Syst.* 8.3 (Apr. 2009), 22:1–22:22. ISSN: 1539-9087. DOI: 10.1145/1509288.1509294.

[43] Roman Lysecky, Frank Vahid, and Sheldon X.-D. Tan. “Dynamic FPGA Routing for Just-in-time FPGA Compilation”. In: *Proceedings of the 41st Annual Design Automation Conference.* DAC ’04. San Diego, CA, USA: ACM, 2004, pp. 954–959. ISBN: 1-58113-828-8. DOI: 10.1145/996566.996819.

[44] B. Mei, M. Berekovic, and J-Y. Mignolet. “ADRES & DRESC: Architecture and Compiler for Coarse-GrainReconfigurable Processors”. In: *Fine- and Coarse-Grain Reconfigurable Computing.* Ed. by Stamatis Vassiliadis and Dimitrios Soudris. Dordrecht: Springer Netherlands, 2007, pp. 255–297. ISBN: 978-1-4020-6505-7.

[45] B. Mei, B. De Sutter, T. Vander Aa, M. Wouters, A. Kanstein, and S. Dupont. “Implementation of a Coarse-Grained Reconfigurable Media Processor for AVC Decoder”. In: *Journal of Signal Processing Systems* 51.3 (June 2008), pp. 225–243. ISSN: 1939-8115. DOI: 10.1007/s11265-007-0152-8.

[46] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling". In: *IEE Proceedings - Computers and Digital Techniques* 150.5 (Sept. 2003), pp. 255-61-. ISSN: 1350-2387. DOI: 10.1049/ip-cdt:20030833.

[47] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. "DRESC: a retargetable compiler for coarse-grained reconfigurable architectures". In: *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.* Dec. 2002, pp. 166–173. DOI: 10.1109/FPT.2002.1188678.

[48] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix". In: *Field Programmable Logic and Application.* Ed. by Peter Y. K. Cheung and George A. Constantinides. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 61–70. ISBN: 978-3-540-45234-8.

[49] Michael Meister. "Erweiterung des AMIDAR-Java-Prozessors von Java 1.4 auf Java 8". Bachelors Thesis. Computer Systems Group: TU Darmstadt, 2017.

[50] Karsten Müller. "Anpassung des AMIDAR Simulators an die aktuelle Hardware Implementierung". Bachelors Thesis. Computer Systems Group: TU Darmstadt, 2016.

[51] *NVIDIA TESLA V100 GPU ARCHITECTURE, THE WORLD'S MOST ADVANCED DATA CENTER GPU.* Whitepaper. WP-08608-001 v1.1. 2017.

[52] *NVIDIA's Next Generation CUDATM Compute Architecture: FermiTM.* Whitepaper. 2009.

[53] N. M. C. Paulino, J. C. Ferreira, and J. M. P. Cardoso. "Trace-Based Reconfigurable Acceleration with Data Cache and External Memory Support". In: *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications.* Aug. 2014, pp. 158–165. DOI: 10.1109/ISPA.2014.29.

[54] Nuno Paulino, João Canas Ferreira, João Bispo, and João M. P. Cardoso. "Transparent Acceleration of Program Execution Using Reconfigurable Hardware". In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition.* DATE '15. Grenoble, France: EDA Consortium, 2015, pp. 1066–1071.

[55] Stefan Pees. "Modeling Embedded Processors and Generating Fast Simulators Using the Machine Description Language LISA". PhD thesis. Rheinisch–Westfälischen Technischen Hochschule Aachen, 2003.

[56] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. "Plasticine: A Reconfigurable Architecture For Parallel Paterns". In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 389–402. ISSN: 0163-5964. DOI: 10.1145/3140659.3080256.

[57] Z. E. Rákossy, F. Merchant, A. Acosta-Aponte, S. K. Nandy, and A. Chattopadhyay. "Scalable and energy-efficient reconfigurable accelerator for column-wise givens rotation". In: *2014 22nd International Conference on Very Large Scale Integration (VLSI-SoC).* Oct. 2014, pp. 1–6. DOI: 10.1109/VLSI-SoC.2014.7004166.

[58] Z. E. Rákossy, T. Naphade, and A. Chattopadhyay. “Design and analysis of layered coarse-grained reconfigurable architecture”. In: *2012 International Conference on Reconfigurable Computing and FPGAs*. Dec. 2012, pp. 1–6. DOI: `10.1109/ReConFig.2012.6416736`.

[59] Zoltán Endre Rákossy, Dominik Stengele, Axel Acosta-Aponte, Saumitra Chafekar, Paolo Bientinesi, and Anupam Chattopadhyay. “Scalable and Efficient Linear Algebra Kernel Mapping for Low Energy Consumption on the Layers CGRA”. In: *Applied Reconfigurable Computing*. Ed. by Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz. Cham: Springer International Publishing, 2015, pp. 301–310. ISBN: 978-3-319-16214-0.

[60] M. Reichenbach, T. Lieske, S. Vaas, K. Haublein, and D. Fey. “FAUPU - A design framework for the development of programmable image processing architectures”. In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Dec. 2015, pp. 1–8. DOI: `10.1109/ReConFig.2015.7393309`.

[61] Johanna Rohde, L. J. Jung, and C. Hochberger. “Update or Invalidate: Influence of Coherence Protocols on Configurable HW Accelerators”. In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Apr. 2019, pp. 305–316.

[62] T. Ruschke, L. J. Jung, and C. Hochberger. “A Near Optimal Integrated Solution for Resource Constrained Scheduling, Binding and Routing on CGRAs”. In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2017, pp. 213–218. DOI: `10.1109/IPDPSW.2017.99`.

[63] T. Ruschke, L. J. Jung, D. Wolf, and C. Hochberger. “Scheduler for Inhomogeneous and Irregular CGRAs with Support for Complex Control Flow”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2016, pp. 198–207. DOI: `10.1109/IPDPSW.2016.72`.

[64] Tajas Ruschke. “Design and Implementation of a Scheduling Algorithm for a CGRA with regard to Routing Constraints”. Masters Thesis. Computer Systems Group: TU Darmstadt, 2015.

[65] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. Reed Taylor. “PipeRench: A virtualized programmable datapath in 0.18 micron technology”. In: *Proceedings of the IEEE 2002 Custom Integrated Circuits Conference (Cat. No.02CH37285)*. May 2002, pp. 63–66. DOI: `10.1109/CICC.2002.1012767`.

[66] Hendrik Schöffmann. “Implementierung eines Verfahrens zum nachfragegetriebenen Laden von CGRA-Kontexten”. Project Seminar. Computer Systems Group: TU Darmstadt, 2018.

[67] Yan Solihin. *Fundamentals of Parallel Multicore Architecture*. Chapman and Hall/CRC, 2015.

[68] Jeckson Dellagostin Souza, Anderson L. Sartor, Luigi Carro, Mateus Beck Rutzig, Stephan Wong, and Antonio C. S. Beck. “DIM-VEX: Exploiting Design Time Configurability and Runtime Reconfigurability”. In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Ed. by Nikolaos Voros, Michael Huebner, Georgios Keramidas, Diana Goehringer, Christos Antonopoulos, and

Pedro C. Diniz. Cham: Springer International Publishing, 2018, pp. 367–378. ISBN: 978-3-319-78890-6.

[69] Sven Ströher. "Implementierung eines verbesserten Simulators für AMIDAR Prozessoren". Bachelors Thesis. Computer Systems Group: TU Darmstadt, 2016.

[70] C. -. Su, C. -. Tsui, and A. M. Despain. "Saving power in the control path of embedded processors". In: *IEEE Design Test of Computers* 11.4 (Winter 1994), pp. 24–31. ISSN: 0740-7475. DOI: 10.1109/54.329448.

[71] Anita Tino and Kaamran Raahmifar. "Assessing Multi-Task Placement Algorithms in RCUs". In: *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016 IEEE International.* May 2016, pp. 1–6.

[72] Francisco-Javier Veredas, M. Scheppler, W. Moffat, and Bingfeng Mei. "Custom implementation of the coarse-grained reconfigurable ADRES architecture for multimedia purposes". In: *International Conference on Field Programmable Logic and Applications, 2005.* Aug. 2005, pp. 106–111. DOI: 10.1109/FPL.2005.1515707.

[73] Adrian Weber. "Implementierung eines Hardware-Profilers für AMIDAR Prozessoren". Bachelors Thesis. Computer Systems Group: TU Darmstadt, 2017.

[74] Dennis Leander Wolf. "Implementation of a Coarse Grained Reconfigurable Array". Masters Thesis. Computer Systems Group: TU Darmstadt, 2015.

[75] Dennis Leander Wolf, Lukas Johannes Jung, Tajas Ruschke, Changgong Li, and Christian Hochberger. "AMIDAR Project: Lessons Learned in 15 Years of Researching Adaptive Processors". In: *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC).* July 2018, pp. 1–8.

[76] S. Wong, T. van As, and G. Brown. "ρ-VEX: A reconfigurable and extensible softcore VLIW processor". In: *2008 International Conference on Field-Programmable Technology.* Dec. 2008, pp. 369–372. DOI: 10.1109/FPT.2008.4762420.

[77] Wm. A. Wulf and Sally A. McKee. "Hitting the Memory Wall: Implications of the Obvious". In: *SIGARCH Comput. Archit. News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964. DOI: 10.1145/216585.216588.

[78] C. Yang, L. Liu, S. Yin, and S. Wei. "Data cache prefetching via context directed pattern matching for coarse-grained reconfigurable arrays". In: *2016 53nd DAC.* June 2016, pp. 1–6.

A. AMIDAR Simulator Accuracy

The AMIDAR simulator accuracy was evaluated by executing the following sobel filter.

Listing A.1: Sobel filter used to test the AMIDAR simulator accuracy

```
public void sobelEdgeDetection(int[] img,
     int imgWidth, int imgHeight) {

 int[] rgbX = new int[3];
 int[] rgbY = new int[3];

 int[] finalPicture = this.finalPicture;
 int[] hx = this.hx;
 int[] hy = this.hy;

 for(int y = 3; y < imgHeight - 3; y++) {
  for(int x = 3; x < imgWidth - 3; x++){

   convolvePixel(hx, img, imgWidth, imgHeight, x, y, rgbX);
   convolvePixel(hy, img, imgWidth, imgHeight, x, y, rgbY);

   int r = abs(rgbX[0]) + abs(rgbY[0]);
   int g = abs(rgbX[1]) + abs(rgbY[1]);
   int b = abs(rgbX[2]) + abs(rgbY[2]);

   if(r > 255)
    r = 255;
   if(g > 255)
    g = 255;
   if(b > 255)
    b = 255;
   setRGB(finalPicture, imgWidth, x, y, (r<<16)|(g<<8)|b);
  }
 }

}
```

```
private void convolvePixel(int[] kernel, int[] img,
        int imgWidth, int imgHeight, int x, int y, int[] rgb) {

 int halfWidth = 1;
 int halfHeight = 1;


 for(int component = 0; component < 3; component++) {
  int sum = 0;
  int i = 0;
  for(int row = -halfHeight; row <= halfWidth; row ++) {
   for(int column = -halfHeight;
           column<= halfHeight; column++) {
    int imgRGB = getRGB(img, imgWidth, x-row, y-column);
    sum = sum + kernel[i++]*
          ((imgRGB >> (16 - 8 * component)) & 0xff);
   }
  }
  rgb[component] =  sum;
 }
}


private int getRGB(int[] img, int imgWidth, int x, int y) {
 return img[y * imgWidth + x];
}

private void setRGB(int[] img, int imgWidth,
          int x, int y, int rgb) {
 img[y * imgWidth + x] = rgb;
}

private int abs(int v) {
 return v < 0 ? -v : v;
}
```

B. CGRA Description

In the following a CGRA description in JSON format is shown exemplarily.

Listing B.1: CGRA descriptrion

```
{
"PEs" :
{
        "0" : "PE_no_mem.json",
        "1" : "PE_mem.json",
        "2" : "PE_no_mem.json",
        "3" : "PE_no_mem.json",
        "4" : "PE_no_mem.json",
        "5" : "PE_no_mem.json",
        "6" : "PE_no_mem.json",
        "7" : "PE_mem.json",
        "8" : "PE_mem.json",
        "9" : "PE_no_mem.json",
        "10" : "PE_no_mem.json",
        "11" : "PE_no_mem.json",
        "12" : "PE_no_mem.json",
        "13" : "PE_no_mem.json",
        "14" : "PE_mem.json",
        "15" : "PE_no_mem.json"
},
"Context_memory_size" : 4096,
"Interconnect" : "Interconnect_16pe_mesh.json",
"CBox_slots" : 128,
"CBox_evaluation_blocks" : 4,
"CBOX_output_ports_per_evaluation_blocks" : 1,
"Pipelined" : false,
"SecondRFOutput2ALU" : false,
"Branch_selection_mode" : "default",
"RomSize" : 32,
}
```

Listing B.2: CGRA mesh interconnect `Interconnect_16pe_mesh.json`

```
{
"name" : "Interconnect",
"Interconnection":
{
        "0" : [1,4],
        "1" : [0,2,5],
        "2" : [1,3,6],
        "3" : [2,7],
        "4" : [0,5,8],
        "5" : [1,4,6,9],
        "6" : [2,5,7,10],
        "7" : [3,6,11],
        "8" : [4,9,12],
        "9" : [5,8,10,13],
        "10" : [6,9,11,14],
        "11" : [7,10,15],
        "12" : [8,13],
        "13" : [9,12,14],
        "14" : [10,13,15],
        "15" : [11,14]
},
"live_out" :
{
        "0" : false,
        "1" : true,
        "2" : false,
        "3" : false,
        "4" : false,
        "5" : false,
        "6" : false,
        "7" : true,
        "8" : true,
        "9" : false,
        "10" : false,
        "11" : false,
        "12" : false,
        "13" : false,
        "14" : true,
        "15" : false
}
}
```

Listing B.3: PE description `PE_mem.json`

```
{
        "name"           : "PE_TYPE_mem",
        "IADD"           : {"energy": 1.100000,"duration": 1},
        "ISUB"           : {"energy": 1.100000,"duration": 1},
        "IDIV"           : {"energy": 7.700000,"duration": 7},
        "IMUL"           : {"energy": 4.400000,"duration": 2},
        "IREM"           : {"energy": 1.100000,"duration": 7},
        "IOR"            : {"energy": 1.100000,"duration": 1},
        "IAND"           : {"energy": 1.100000,"duration": 1},
        "IXOR"           : {"energy": 1.100000,"duration": 1},
        "ISHL"           : {"energy": 1.100000,"duration": 1},
        "ISHR"           : {"energy": 1.100000,"duration": 1},
        "IUSHR"          : {"energy": 1.100000,"duration": 1},
        "FADD"          : {"energy": 1.100000,"duration": 6},
        "FSUB"          : {"energy": 1.100000,"duration": 6},
        "FMUL"          : {"energy": 4.400000,"duration": 5},
        "FDIV"          : {"energy": 7.700000,"duration": 10},
        "LOAD"          : {"energy": 2.200000,"duration": 1},
        "STORE"         : {"energy": 1.100000,"duration": 1},
        "LOAD64"        : {"energy": 2.200000,"duration": 2},
        "STORE64"       : {"energy": 1.100000,"duration": 2},
        "DMA_LOAD"      : {"energy": 2.200000,"duration": 2},
        "DMA_STORE"     : {"energy": 2.200000,"duration": 2},
        "DMA_LOAD64"    : {"energy": 2.200000,"duration": 4},
        "DMA_STORE64"   : {"energy": 2.200000,"duration": 4},
        "CACHE_FETCH"   : {"energy": 2.200000,"duration": 2},
        "IFEQ"          : {"energy": 1.100000,"duration": 1},
        "IFNE"          : {"energy": 1.100000,"duration": 1},
        "IFGE"          : {"energy": 1.100000,"duration": 1},
        "IFGT"          : {"energy": 1.100000,"duration": 1},
        "IFLE"          : {"energy": 1.100000,"duration": 1},
        "IFLT"          : {"energy": 1.100000,"duration": 1},
        "CI_CMP"        : {"energy": 1.100000,"duration": 1},
        "HANDLE_CMP"    : {"energy": 1.100000,"duration": 1},
        "CONST"         : {"energy": 1.100000,"duration": 1},
        "I2F"           : {"energy": 1.100000,"duration": 1},
        "F2I"           : {"energy": 1.100000,"duration": 1},
        "I2B"           : {"energy": 1.100000,"duration": 1},
        "I2C"           : {"energy": 1.100000,"duration": 1},
        "I2S"           : {"energy": 1.100000,"duration": 1},
        "INEG"          : {"energy": 1.100000,"duration": 1},
        "FNEG"          : {"energy": 1.100000,"duration": 1},
        "FSIN"          : {"energy": 7.77770,"duration": 7},
        "FCOS"          : {"energy": 7.77770,"duration": 7},
        "NOP"           : {"energy": 1.100000,"duration": 1},
        "Regfile_size"  : 256,
        "rom_size"      : 16
}
```

C. Nested Loops in The Schedule

Figure C.1 shows an excerpt of the schedule of the ADPCM decoder mapped on a 2x2 CGRA with unroll factor 2. It can be seen that a nested loop starts at $t = 178$ and ends at $t = 188$. The loop condition is checked at $t = 178$ on PE2. If it is evaluated to false, the CCU performs a relative jump of distance 11 to $t = 189$. Note that the instruction that checks the loop condition (`1261:IFLE`) is marked green which means that it will be executed conditionally. The result of the instruction can only be true when the result of the instruction `466:IFGE` was also true (see C-Box column on the right).

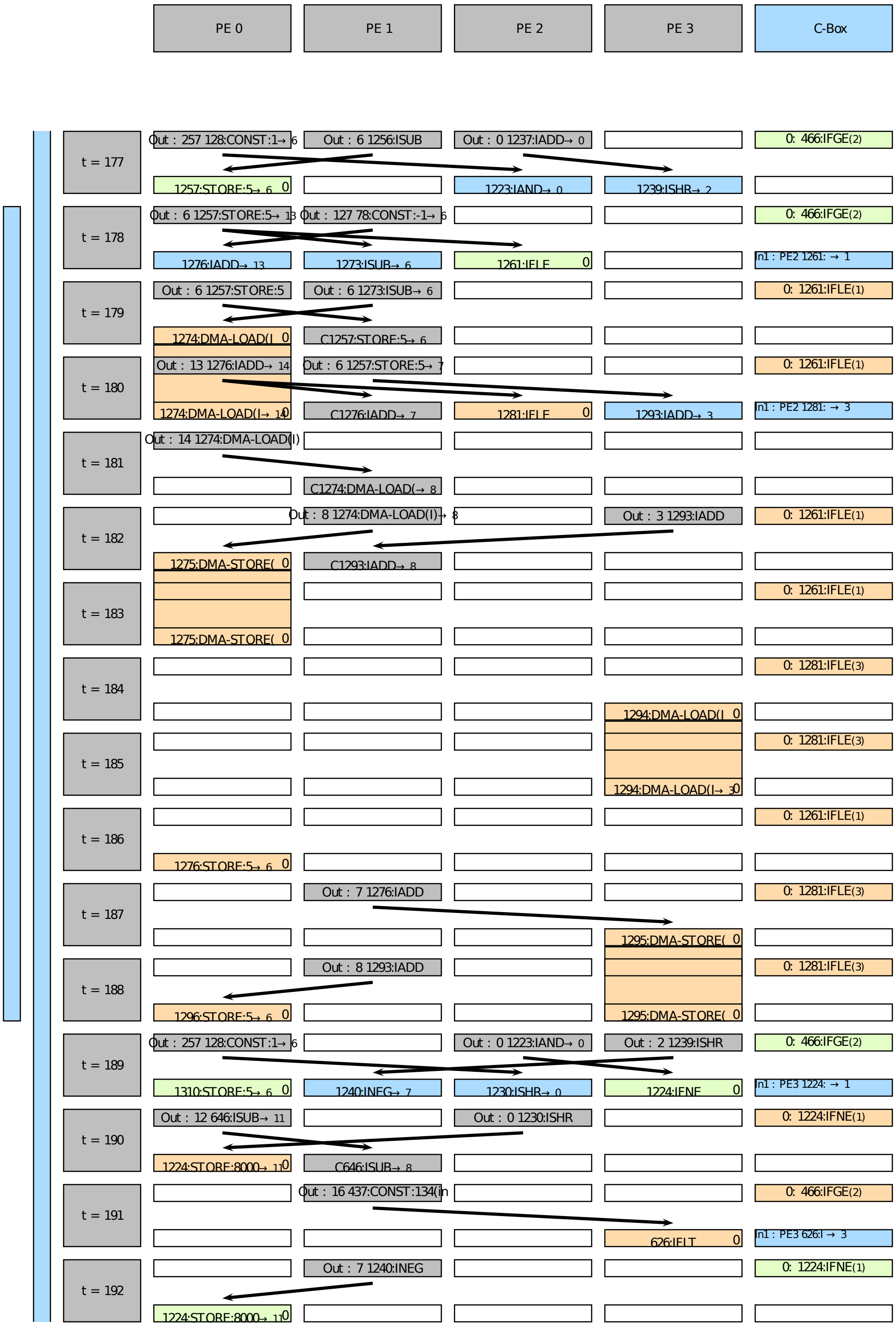


Figure C.1.: Excerpt of the schedule of the ADPCM decoder on a 2x2 CGRA

D. Design Space Exploration

Listing D.1 shows the sweep configuration of the design space exploration. In total this results in 18432 different simulations.

Listing D.1: Design space exploration sweep configuration

```
{
"parameter" :
{

"benchmarkScale" : [5,12],
"COHERENCE_PROTOCOL"             : ["DRAGON", "MOESI"],
"PREFETCHING"    : ["NONE","LINEAR", "UNROLL"]
"SW_COHERENCE_SUPPORT"        : [false, true],
"ALIASING_SPECULATION"     : ["OFF","PESSIMISTIC_CHECK"]
},
"fu" :
{
"CGRA" : [      "config/FU/CGRA/CGRA_165.json",
                        "config/FU/CGRA/CGRA_165_2.json",
                        "config/FU/CGRA/CGRA_165_3.json",
                        "config/FU/CGRA/CGRA_16.json"],
}
"application" :
[
"de/amidar/cacheBench/jpeg/Jpeg"

"de/amidar/cacheBench/adpcm/ADPCMencode",
"de/amidar/cacheBench/adpcm/ADPCMdecode",

"de/amidar/cacheBench/crypto/AES",
"de/amidar/cacheBench/crypto/DES",
"de/amidar/cacheBench/crypto/Blowfish",
"de/amidar/cacheBench/crypto/IDEA",
"de/amidar/cacheBench/crypto/RC6",
"de/amidar/cacheBench/crypto/Serpent",
"de/amidar/cacheBench/crypto/Skipjack",
"de/amidar/cacheBench/crypto/Twofish",
"de/amidar/cacheBench/crypto/XTEA",

"de/amidar/cacheBench/digests/BLAKE256",
"de/amidar/cacheBench/digests/CubeHash512",
"de/amidar/cacheBench/digests/ECOH256",
"de/amidar/cacheBench/digests/MD5",
"de/amidar/cacheBench/digests/RadioGatun32",
"de/amidar/cacheBench/digests/SHA1",
"de/amidar/cacheBench/digests/SHA256",
"de/amidar/cacheBench/digests/SIMD512",

"de/amidar/cacheBench/filter/ContrastFilter",
"de/amidar/cacheBench/filter/GrayscaleFilter",
"de/amidar/cacheBench/filter/SobelFilter",
"de/amidar/cacheBench/filter/SwizzleFilter",



],
"cache" :
{
"totalSizeCGRA" : [16],
},
"l2cache" :
{
"virtual" : [false, true],
"wordsPerLine" : [8,16]
}

}
```

D.1. Results

The following table shows the results of the design space exploration

Speedup	CGRA	ACD	Prefetcher	Aliasing Speculation	Coherence Protocol	L2 Cache address scheme	Words per L2 cache line
23.6	(a)	off	Linear	off	Dragon	physical	8
23.89	(a)	off	Linear	off	Dragon	physical	16
24.46	(a)	off	Linear	off	Dragon	virtual	8
24.99	(a)	off	Linear	off	Dragon	virtual	16
21.55	(a)	off	Linear	off	MOESI	physical	8
21.8	(a)	off	Linear	off	MOESI	physical	16
22.29	(a)	off	Linear	off	MOESI	virtual	8
22.64	(a)	off	Linear	off	MOESI	virtual	16
23.59	(a)	off	Linear	on	Dragon	physical	8
23.9	(a)	off	Linear	on	Dragon	physical	16
24.25	(a)	off	Linear	on	Dragon	virtual	8
24.77	(a)	off	Linear	on	Dragon	virtual	16
21.15	(a)	off	Linear	on	MOESI	physical	8

21.41	(a)	off	Linear	on	MOESI	physical	16
21.93	(a)	off	Linear	on	MOESI	virtual	8
22.2	(a)	off	Linear	on	MOESI	virtual	16
23.0	(a)	off	none	off	Dragon	physical	8
24.16	(a)	off	none	off	Dragon	physical	16
23.01	(a)	off	none	off	Dragon	virtual	8
24.23	(a)	off	none	off	Dragon	virtual	16
21.01	(a)	off	none	off	MOESI	physical	8
22.04	(a)	off	none	off	MOESI	physical	16
21.07	(a)	off	none	off	MOESI	virtual	8
22.15	(a)	off	none	off	MOESI	virtual	16
22.85	(a)	off	none	on	Dragon	physical	8
24.01	(a)	off	none	on	Dragon	physical	16
22.86	(a)	off	none	on	Dragon	virtual	8
24.08	(a)	off	none	on	Dragon	virtual	16
20.66	(a)	off	none	on	MOESI	physical	8
21.67	(a)	off	none	on	MOESI	physical	16
20.72	(a)	off	none	on	MOESI	virtual	8
21.78	(a)	off	none	on	MOESI	virtual	16
23.98	(a)	off	Lookahead	off	Dragon	physical	8
24.66	(a)	off	Lookahead	off	Dragon	physical	16
24.18	(a)	off	Lookahead	off	Dragon	virtual	8
25.07	(a)	off	Lookahead	off	Dragon	virtual	16
21.79	(a)	off	Lookahead	off	MOESI	physical	8
22.39	(a)	off	Lookahead	off	MOESI	physical	16
21.99	(a)	off	Lookahead	off	MOESI	virtual	8
22.76	(a)	off	Lookahead	off	MOESI	virtual	16
23.86	(a)	off	Lookahead	on	Dragon	physical	8
24.57	(a)	off	Lookahead	on	Dragon	physical	16
24.09	(a)	off	Lookahead	on	Dragon	virtual	8
25.06	(a)	off	Lookahead	on	Dragon	virtual	16
21.62	(a)	off	Lookahead	on	MOESI	physical	8
22.2	(a)	off	Lookahead	on	MOESI	physical	16
21.88	(a)	off	Lookahead	on	MOESI	virtual	8
22.7	(a)	off	Lookahead	on	MOESI	virtual	16
24.54	(a)	on	Linear	off	Dragon	physical	8
24.68	(a)	on	Linear	off	Dragon	physical	16
25.14	(a)	on	Linear	off	Dragon	virtual	8
25.52	(a)	on	Linear	off	Dragon	virtual	16
24.71	(a)	on	Linear	off	MOESI	physical	8
25.01	(a)	on	Linear	off	MOESI	physical	16
25.21	(a)	on	Linear	off	MOESI	virtual	8
25.72	(a)	on	Linear	off	MOESI	virtual	16
24.57	(a)	on	Linear	on	Dragon	physical	8

24.85	(a)	on	Linear	on	Dragon	physical	16
25.24	(a)	on	Linear	on	Dragon	virtual	8
25.65	(a)	on	Linear	on	Dragon	virtual	16
24.93	(a)	on	Linear	on	MOESI	physical	8
25.29	(a)	on	Linear	on	MOESI	physical	16
25.39	(a)	on	Linear	on	MOESI	virtual	8
25.95	(a)	on	Linear	on	MOESI	virtual	16
23.21	(a)	on	none	off	Dragon	physical	8
24.41	(a)	on	none	off	Dragon	physical	16
23.2	(a)	on	none	off	Dragon	virtual	8
24.48	(a)	on	none	off	Dragon	virtual	16
23.09	(a)	on	none	off	MOESI	physical	8
24.29	(a)	on	none	off	MOESI	physical	16
23.08	(a)	on	none	off	MOESI	virtual	8
24.37	(a)	on	none	off	MOESI	virtual	16
23.28	(a)	on	none	on	Dragon	physical	8
24.49	(a)	on	none	on	Dragon	physical	16
23.27	(a)	on	none	on	Dragon	virtual	8
24.57	(a)	on	none	on	Dragon	virtual	16
23.28	(a)	on	none	on	MOESI	physical	8
24.5	(a)	on	none	on	MOESI	physical	16
23.28	(a)	on	none	on	MOESI	virtual	8
24.58	(a)	on	none	on	MOESI	virtual	16
24.83	(a)	on	Lookahead	off	Dragon	physical	8
25.7	(a)	on	Lookahead	off	Dragon	physical	16
24.97	(a)	on	Lookahead	off	Dragon	virtual	8
26.01	(a)	on	Lookahead	off	Dragon	virtual	16
24.81	(a)	on	Lookahead	off	MOESI	physical	8
25.75	(a)	on	Lookahead	off	MOESI	physical	16
24.96	(a)	on	Lookahead	off	MOESI	virtual	8
26.03	(a)	on	Lookahead	off	MOESI	virtual	16
24.68	(a)	on	Lookahead	on	Dragon	physical	8
25.49	(a)	on	Lookahead	on	Dragon	physical	16
24.8	(a)	on	Lookahead	on	Dragon	virtual	8
25.84	(a)	on	Lookahead	on	Dragon	virtual	16
24.87	(a)	on	Lookahead	on	MOESI	physical	8
25.77	(a)	on	Lookahead	on	MOESI	physical	16
25.01	(a)	on	Lookahead	on	MOESI	virtual	8
26.09	(a)	on	Lookahead	on	MOESI	virtual	16
23.79	(b)	off	Linear	off	Dragon	physical	8
24.06	(b)	off	Linear	off	Dragon	physical	16
24.69	(b)	off	Linear	off	Dragon	virtual	8
24.97	(b)	off	Linear	off	Dragon	virtual	16
22.62	(b)	off	Linear	off	MOESI	physical	8

23.11	(b)	off	Linear	off	MOESI	physical	16
23.42	(b)	off	Linear	off	MOESI	virtual	8
23.8	(b)	off	Linear	off	MOESI	virtual	16
23.52	(b)	off	Linear	on	Dragon	physical	8
23.99	(b)	off	Linear	on	Dragon	physical	16
24.4	(b)	off	Linear	on	Dragon	virtual	8
24.84	(b)	off	Linear	on	Dragon	virtual	16
22.29	(b)	off	Linear	on	MOESI	physical	8
22.8	(b)	off	Linear	on	MOESI	physical	16
23.14	(b)	off	Linear	on	MOESI	virtual	8
23.45	(b)	off	Linear	on	MOESI	virtual	16
23.08	(b)	off	none	off	Dragon	physical	8
24.26	(b)	off	none	off	Dragon	physical	16
23.06	(b)	off	none	off	Dragon	virtual	8
24.32	(b)	off	none	off	Dragon	virtual	16
21.85	(b)	off	none	off	MOESI	physical	8
22.94	(b)	off	none	off	MOESI	physical	16
21.87	(b)	off	none	off	MOESI	virtual	8
23.02	(b)	off	none	off	MOESI	virtual	16
22.81	(b)	off	none	on	Dragon	physical	8
24.0	(b)	off	none	on	Dragon	physical	16
22.8	(b)	off	none	on	Dragon	virtual	8
24.05	(b)	off	none	on	Dragon	virtual	16
21.47	(b)	off	none	on	MOESI	physical	8
22.54	(b)	off	none	on	MOESI	physical	16
21.49	(b)	off	none	on	MOESI	virtual	8
22.62	(b)	off	none	on	MOESI	virtual	16
24.14	(b)	off	Lookahead	off	Dragon	physical	8
24.92	(b)	off	Lookahead	off	Dragon	physical	16
24.24	(b)	off	Lookahead	off	Dragon	virtual	8
25.25	(b)	off	Lookahead	off	Dragon	virtual	16
22.88	(b)	off	Lookahead	off	MOESI	physical	8
23.55	(b)	off	Lookahead	off	MOESI	physical	16
23.03	(b)	off	Lookahead	off	MOESI	virtual	8
23.81	(b)	off	Lookahead	off	MOESI	virtual	16
23.72	(b)	off	Lookahead	on	Dragon	physical	8
24.5	(b)	off	Lookahead	on	Dragon	physical	16
23.81	(b)	off	Lookahead	on	Dragon	virtual	8
24.81	(b)	off	Lookahead	on	Dragon	virtual	16
22.47	(b)	off	Lookahead	on	MOESI	physical	8
23.07	(b)	off	Lookahead	on	MOESI	physical	16
22.65	(b)	off	Lookahead	on	MOESI	virtual	8
23.4	(b)	off	Lookahead	on	MOESI	virtual	16
24.36	(b)	on	Linear	off	Dragon	physical	8

24.85	(b)	on	Linear	off	Dragon	physical	16
25.18	(b)	on	Linear	off	Dragon	virtual	8
25.65	(b)	on	Linear	off	Dragon	virtual	16
24.84	(b)	on	Linear	off	MOESI	physical	8
25.36	(b)	on	Linear	off	MOESI	physical	16
25.43	(b)	on	Linear	off	MOESI	virtual	8
26.04	(b)	on	Linear	off	MOESI	virtual	16
24.47	(b)	on	Linear	on	Dragon	physical	8
24.99	(b)	on	Linear	on	Dragon	physical	16
25.2	(b)	on	Linear	on	Dragon	virtual	8
25.77	(b)	on	Linear	on	Dragon	virtual	16
25.15	(b)	on	Linear	on	MOESI	physical	8
25.72	(b)	on	Linear	on	MOESI	physical	16
25.62	(b)	on	Linear	on	MOESI	virtual	8
26.35	(b)	on	Linear	on	MOESI	virtual	16
23.3	(b)	on	none	off	Dragon	physical	8
24.54	(b)	on	none	off	Dragon	physical	16
23.3	(b)	on	none	off	Dragon	virtual	8
24.6	(b)	on	none	off	Dragon	virtual	16
23.26	(b)	on	none	off	MOESI	physical	8
24.48	(b)	on	none	off	MOESI	physical	16
23.26	(b)	on	none	off	MOESI	virtual	8
24.55	(b)	on	none	off	MOESI	virtual	16
23.2	(b)	on	none	on	Dragon	physical	8
24.45	(b)	on	none	on	Dragon	physical	16
23.21	(b)	on	none	on	Dragon	virtual	8
24.52	(b)	on	none	on	Dragon	virtual	16
23.36	(b)	on	none	on	MOESI	physical	8
24.62	(b)	on	none	on	MOESI	physical	16
23.37	(b)	on	none	on	MOESI	virtual	8
24.69	(b)	on	none	on	MOESI	virtual	16
24.35	(b)	on	Lookahead	off	Dragon	physical	8
25.3	(b)	on	Lookahead	off	Dragon	physical	16
24.5	(b)	on	Lookahead	off	Dragon	virtual	8
25.61	(b)	on	Lookahead	off	Dragon	virtual	16
24.51	(b)	on	Lookahead	off	MOESI	physical	8
25.5	(b)	on	Lookahead	off	MOESI	physical	16
24.59	(b)	on	Lookahead	off	MOESI	virtual	8
25.72	(b)	on	Lookahead	off	MOESI	virtual	16
24.17	(b)	on	Lookahead	on	Dragon	physical	8
25.13	(b)	on	Lookahead	on	Dragon	physical	16
24.33	(b)	on	Lookahead	on	Dragon	virtual	8
25.45	(b)	on	Lookahead	on	Dragon	virtual	16
24.66	(b)	on	Lookahead	on	MOESI	physical	8

25.67	(b)	on	Lookahead	on	MOESI	physical	16
24.74	(b)	on	Lookahead	on	MOESI	virtual	8
25.88	(b)	on	Lookahead	on	MOESI	virtual	16
23.17	(c)	off	Linear	off	Dragon	physical	8
23.69	(c)	off	Linear	off	Dragon	physical	16
23.33	(c)	off	Linear	off	Dragon	virtual	8
23.78	(c)	off	Linear	off	Dragon	virtual	16
23.66	(c)	off	Linear	off	MOESI	physical	8
24.17	(c)	off	Linear	off	MOESI	physical	16
23.73	(c)	off	Linear	off	MOESI	virtual	8
24.22	(c)	off	Linear	off	MOESI	virtual	16
23.34	(c)	off	Linear	on	Dragon	physical	8
23.84	(c)	off	Linear	on	Dragon	physical	16
23.59	(c)	off	Linear	on	Dragon	virtual	8
24.04	(c)	off	Linear	on	Dragon	virtual	16
23.83	(c)	off	Linear	on	MOESI	physical	8
24.33	(c)	off	Linear	on	MOESI	physical	16
23.99	(c)	off	Linear	on	MOESI	virtual	8
24.48	(c)	off	Linear	on	MOESI	virtual	16
20.98	(c)	off	none	off	Dragon	physical	8
22.07	(c)	off	none	off	Dragon	physical	16
20.98	(c)	off	none	off	Dragon	virtual	8
22.11	(c)	off	none	off	Dragon	virtual	16
21.29	(c)	off	none	off	MOESI	physical	8
22.39	(c)	off	none	off	MOESI	physical	16
21.28	(c)	off	none	off	MOESI	virtual	8
22.44	(c)	off	none	off	MOESI	virtual	16
21.18	(c)	off	none	on	Dragon	physical	8
22.27	(c)	off	none	on	Dragon	physical	16
21.17	(c)	off	none	on	Dragon	virtual	8
22.31	(c)	off	none	on	Dragon	virtual	16
21.49	(c)	off	none	on	MOESI	physical	8
22.6	(c)	off	none	on	MOESI	physical	16
21.47	(c)	off	none	on	MOESI	virtual	8
22.64	(c)	off	none	on	MOESI	virtual	16
21.59	(c)	off	Lookahead	off	Dragon	physical	8
22.69	(c)	off	Lookahead	off	Dragon	physical	16
21.62	(c)	off	Lookahead	off	Dragon	virtual	8
22.84	(c)	off	Lookahead	off	Dragon	virtual	16
21.89	(c)	off	Lookahead	off	MOESI	physical	8
23.02	(c)	off	Lookahead	off	MOESI	physical	16
21.93	(c)	off	Lookahead	off	MOESI	virtual	8
23.18	(c)	off	Lookahead	off	MOESI	virtual	16
21.73	(c)	off	Lookahead	on	Dragon	physical	8

22.85	(c)	off	Lookahead	on	Dragon	physical	16
21.81	(c)	off	Lookahead	on	Dragon	virtual	8
23.08	(c)	off	Lookahead	on	Dragon	virtual	16
22.04	(c)	off	Lookahead	on	MOESI	physical	8
23.19	(c)	off	Lookahead	on	MOESI	physical	16
22.13	(c)	off	Lookahead	on	MOESI	virtual	8
23.43	(c)	off	Lookahead	on	MOESI	virtual	16
23.17	(c)	on	Linear	off	Dragon	physical	8
23.69	(c)	on	Linear	off	Dragon	physical	16
23.33	(c)	on	Linear	off	Dragon	virtual	8
23.78	(c)	on	Linear	off	Dragon	virtual	16
23.66	(c)	on	Linear	off	MOESI	physical	8
24.17	(c)	on	Linear	off	MOESI	physical	16
23.73	(c)	on	Linear	off	MOESI	virtual	8
24.22	(c)	on	Linear	off	MOESI	virtual	16
23.34	(c)	on	Linear	on	Dragon	physical	8
23.84	(c)	on	Linear	on	Dragon	physical	16
23.59	(c)	on	Linear	on	Dragon	virtual	8
24.04	(c)	on	Linear	on	Dragon	virtual	16
23.83	(c)	on	Linear	on	MOESI	physical	8
24.33	(c)	on	Linear	on	MOESI	physical	16
23.99	(c)	on	Linear	on	MOESI	virtual	8
24.48	(c)	on	Linear	on	MOESI	virtual	16
20.98	(c)	on	none	off	Dragon	physical	8
22.07	(c)	on	none	off	Dragon	physical	16
20.98	(c)	on	none	off	Dragon	virtual	8
22.11	(c)	on	none	off	Dragon	virtual	16
21.29	(c)	on	none	off	MOESI	physical	8
22.39	(c)	on	none	off	MOESI	physical	16
21.28	(c)	on	none	off	MOESI	virtual	8
22.44	(c)	on	none	off	MOESI	virtual	16
21.18	(c)	on	none	on	Dragon	physical	8
22.27	(c)	on	none	on	Dragon	physical	16
21.17	(c)	on	none	on	Dragon	virtual	8
22.31	(c)	on	none	on	Dragon	virtual	16
21.49	(c)	on	none	on	MOESI	physical	8
22.6	(c)	on	none	on	MOESI	physical	16
21.47	(c)	on	none	on	MOESI	virtual	8
22.64	(c)	on	none	on	MOESI	virtual	16
21.59	(c)	on	Lookahead	off	Dragon	physical	8
22.69	(c)	on	Lookahead	off	Dragon	physical	16
21.62	(c)	on	Lookahead	off	Dragon	virtual	8
22.84	(c)	on	Lookahead	off	Dragon	virtual	16
21.89	(c)	on	Lookahead	off	MOESI	physical	8

23.02	(c)	on	Lookahead	off	MOESI	physical	16
21.93	(c)	on	Lookahead	off	MOESI	virtual	8
23.18	(c)	on	Lookahead	off	MOESI	virtual	16
21.73	(c)	on	Lookahead	on	Dragon	physical	8
22.85	(c)	on	Lookahead	on	Dragon	physical	16
21.81	(c)	on	Lookahead	on	Dragon	virtual	8
23.08	(c)	on	Lookahead	on	Dragon	virtual	16
22.04	(c)	on	Lookahead	on	MOESI	physical	8
23.19	(c)	on	Lookahead	on	MOESI	physical	16
22.13	(c)	on	Lookahead	on	MOESI	virtual	8
23.43	(c)	on	Lookahead	on	MOESI	virtual	16
21.55	(d)	off	Linear	off	Dragon	physical	8
21.89	(d)	off	Linear	off	Dragon	physical	16
21.93	(d)	off	Linear	off	Dragon	virtual	8
22.38	(d)	off	Linear	off	Dragon	virtual	16
20.11	(d)	off	Linear	off	MOESI	physical	8
20.37	(d)	off	Linear	off	MOESI	physical	16
20.52	(d)	off	Linear	off	MOESI	virtual	8
20.87	(d)	off	Linear	off	MOESI	virtual	16
21.92	(d)	off	Linear	on	Dragon	physical	8
22.27	(d)	off	Linear	on	Dragon	physical	16
22.18	(d)	off	Linear	on	Dragon	virtual	8
22.63	(d)	off	Linear	on	Dragon	virtual	16
20.14	(d)	off	Linear	on	MOESI	physical	8
20.35	(d)	off	Linear	on	MOESI	physical	16
20.48	(d)	off	Linear	on	MOESI	virtual	8
20.81	(d)	off	Linear	on	MOESI	virtual	16
20.76	(d)	off	none	off	Dragon	physical	8
21.69	(d)	off	none	off	Dragon	physical	16
20.75	(d)	off	none	off	Dragon	virtual	8
21.75	(d)	off	none	off	Dragon	virtual	16
19.3	(d)	off	none	off	MOESI	physical	8
20.15	(d)	off	none	off	MOESI	physical	16
19.33	(d)	off	none	off	MOESI	virtual	8
20.24	(d)	off	none	off	MOESI	virtual	16
20.92	(d)	off	none	on	Dragon	physical	8
21.86	(d)	off	none	on	Dragon	physical	16
20.91	(d)	off	none	on	Dragon	virtual	8
21.93	(d)	off	none	on	Dragon	virtual	16
19.32	(d)	off	none	on	MOESI	physical	8
20.16	(d)	off	none	on	MOESI	physical	16
19.35	(d)	off	none	on	MOESI	virtual	8
20.26	(d)	off	none	on	MOESI	virtual	16
21.74	(d)	off	Lookahead	off	Dragon	physical	8

22.3	(d)	off	Lookahead	off	Dragon	physical	16
21.98	(d)	off	Lookahead	off	Dragon	virtual	8
22.59	(d)	off	Lookahead	off	Dragon	virtual	16
20.16	(d)	off	Lookahead	off	MOESI	physical	8
20.62	(d)	off	Lookahead	off	MOESI	physical	16
20.37	(d)	off	Lookahead	off	MOESI	virtual	8
20.87	(d)	off	Lookahead	off	MOESI	virtual	16
22.0	(d)	off	Lookahead	on	Dragon	physical	8
22.59	(d)	off	Lookahead	on	Dragon	physical	16
22.23	(d)	off	Lookahead	on	Dragon	virtual	8
22.91	(d)	off	Lookahead	on	Dragon	virtual	16
20.29	(d)	off	Lookahead	on	MOESI	physical	8
20.79	(d)	off	Lookahead	on	MOESI	physical	16
20.51	(d)	off	Lookahead	on	MOESI	virtual	8
21.07	(d)	off	Lookahead	on	MOESI	virtual	16
21.74	(d)	on	Linear	off	Dragon	physical	8
22.11	(d)	on	Linear	off	Dragon	physical	16
22.19	(d)	on	Linear	off	Dragon	virtual	8
22.6	(d)	on	Linear	off	Dragon	virtual	16
21.83	(d)	on	Linear	off	MOESI	physical	8
22.15	(d)	on	Linear	off	MOESI	physical	16
22.25	(d)	on	Linear	off	MOESI	virtual	8
22.71	(d)	on	Linear	off	MOESI	virtual	16
22.0	(d)	on	Linear	on	Dragon	physical	8
22.3	(d)	on	Linear	on	Dragon	physical	16
22.35	(d)	on	Linear	on	Dragon	virtual	8
22.85	(d)	on	Linear	on	Dragon	virtual	16
22.21	(d)	on	Linear	on	MOESI	physical	8
22.48	(d)	on	Linear	on	MOESI	physical	16
22.53	(d)	on	Linear	on	MOESI	virtual	8
23.06	(d)	on	Linear	on	MOESI	virtual	16
20.64	(d)	on	none	off	Dragon	physical	8
21.63	(d)	on	none	off	Dragon	physical	16
20.63	(d)	on	none	off	Dragon	virtual	8
21.67	(d)	on	none	off	Dragon	virtual	16
20.42	(d)	on	none	off	MOESI	physical	8
21.4	(d)	on	none	off	MOESI	physical	16
20.42	(d)	on	none	off	MOESI	virtual	8
21.46	(d)	on	none	off	MOESI	virtual	16
20.9	(d)	on	none	on	Dragon	physical	8
21.9	(d)	on	none	on	Dragon	physical	16
20.9	(d)	on	none	on	Dragon	virtual	8
21.96	(d)	on	none	on	Dragon	virtual	16
20.76	(d)	on	none	on	MOESI	physical	8

21.75	(d)	on	none	on	MOESI	physical	16
20.77	(d)	on	none	on	MOESI	virtual	8
21.82	(d)	on	none	on	MOESI	virtual	16
21.89	(d)	on	Lookahead	off	Dragon	physical	8
22.61	(d)	on	Lookahead	off	Dragon	physical	16
22.07	(d)	on	Lookahead	off	Dragon	virtual	8
22.86	(d)	on	Lookahead	off	Dragon	virtual	16
21.7	(d)	on	Lookahead	off	MOESI	physical	8
22.42	(d)	on	Lookahead	off	MOESI	physical	16
21.85	(d)	on	Lookahead	off	MOESI	virtual	8
22.64	(d)	on	Lookahead	off	MOESI	virtual	16
22.1	(d)	on	Lookahead	on	Dragon	physical	8
22.82	(d)	on	Lookahead	on	Dragon	physical	16
22.31	(d)	on	Lookahead	on	Dragon	virtual	8
23.14	(d)	on	Lookahead	on	Dragon	virtual	16
22.06	(d)	on	Lookahead	on	MOESI	physical	8
22.79	(d)	on	Lookahead	on	MOESI	physical	16
22.19	(d)	on	Lookahead	on	MOESI	virtual	8
23.01	(d)	on	Lookahead	on	MOESI	virtual	16

E. Contributions of this Work in Part II

This section aims to make clear the contributions of this work. Part two describes the processor-accelerator system. The Chapters 5, 6 and 7 describe mostly work that was done prior to this work. The only exception are the hardware implementation of the profiler described in 6.4, which needed some changes[73] and the C-Box in Chapter 7.4. The C-Box was developed together with the student works [74] and [64]. The CGRA generator was also developed during this work in a supervised Masters Thesis [74]. The contents of Chapter 8 were completely developed during this work. Chapter 9 is mostly based on previous work. Exceptions are the speculative method inlining described in Section 9.1 which was completely developed during this work and the Sections 9.5 and 9.6. Also, the scheduler was re-implemented during this work in a supervised Masters Thesis [64].