An Execution Model and High-Level-Synthesis System for Generating SIMT Multi-Threaded Hardware from C Source Code

Ein Ausführungsmodell und High-Level-Synthese System zur Erzeugung von SIMT Multithread-Hardware aus C Quellcode

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von Dipl.-Inform. Jens Christoph Huthmann aus Darmstadt

Tag der Einreichung: 03.06.2017, Tag der Prüfung: 21.08.2017
Darmstadt 2017 — D 17

1. Gutachten: Prof. Dr.-Ing. Andreas Koch
2. Gutachten: Prof. Dr.-Ing. Mladen Berekovic

TECHNISCHE UNIVERSITÄT DARMSTADT

Embedded Systems & Applications
Fachbereich Informatik

An Execution Model and High-Level-Synthesis System for Generating SIMT Multi-Threaded Hardware from C Source Code
Ein Ausführungsmodell und High-Level-Synthese System zur Erzeugung von SIMT Multithread-Hardware aus C Quellcode

Genehmigte Dissertation von Dipl.-Inform. Jens Christoph Huthmann aus Darmstadt

1. Gutachten: Prof. Dr.-Ing. Andreas Koch
2. Gutachten: Prof. Dr.-Ing. Mladen Berekovic

Tag der Einreichung: 03.06.2017
Tag der Prüfung: 21.08.2017

Darmstadt 2017 — D 17

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-67767
URL: http://tuprints.ulb.tu-darmstadt.de/id/eprint/6776

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den

(Jens Christoph Huthmann)

Abstract

The performance improvement of conventional processor has begun to stagnate in recent years. Because of this, researchers are looking for new possibilities to improve the performance of computing systems. Heterogeneous systems turned out to be a powerful possibility. In the context of this thesis, a heterogeneous system consists of a software-programmable processor and a FPGA based configurable hardware accelerator. By using an accelerator specifically tailored to a particular application, heterogeneous system can achieve a higher performance that conventional processors.

Due to their increased complexity, it is more complicated to develop applications for heterogeneous systems than for conventional systems based on a software-programmable processor. For programming the software and hardware parts, different languages have to be used and additional specialised hardware-knowledge is required. Both factors increase the development cost.

This work presents the compiler framework *Nymble* which allows to program a heterogeneous system with only a single high-level language. In the high-level language the developer only has to select which parts of the application should be executed in hardware. Nymble then generates a program for the software-processor, the configuration of the hardware, and all interfaces between software and hardware.

All heterogeneous systems supported by Nymble have in common that the software and hardware parts of an application have access to a shared memory. As this memory is external RAM with high access latency, it is necessary to insert a cache between the memory and hardware. With this cache, memory accesses can vary between very short or long access latency depending on whether the data is available in the cache.

To hide long latencies, this thesis presents a novel execution model which allows the simultaneous execution of multiple threads in a single accelerator. Additionally, the model enables threads to be dynamically reordered at specific points in the common accelerator pipeline. This capability is used to let other (non-waiting) threads overtake a thread which is waiting for a memory access. Thus, these other threads can execute their calculations independently of the waiting thread to bridge the latency of memory accesses.

Previous works are using execution models which only allow a single thread to be active in the accelerator. In case of a memory access with long latency, the

thread is exchanged with another non-waiting thread. This design of the hardware often causes many resources to lie idle for a significant amount of time.

In contrast, the presented novel execution model dynamically spreads multiple threads over the pipeline. This results in a higher utilisation of the resources by using resources more effectively. Furthermore, the simultaneous execution of multiple threads can achieve similar throughput as multiple copies of a single-threaded accelerator running in parallel.

The new execution model makes it possible to combine the improved throughput of multiple copies with the increased efficiency of simultaneous threads in a single accelerator. Thread reordering allows the new model to be effectively used with a cached shared-memory.

In comparison, between four copies of a single-threaded accelerator and a multi-thread accelerator with four thread (both created by Nymble), a resource efficiency of up to factor 2.6x can be achieved. At the same time, four simultaneous threads can be up to 4x as fast as four threads executed consecutively on a single accelerator. Compared to other, more optimised compilers, Nymble can still achieve up to 2x faster runtime with 1.5x resource efficiency.

Kurzfassung

Da bei der Leistung von herkömmlichen Prozessoren in den vergangenen Jahren eine Stagnation bei der Verbesserung der Leistung zu verzeichnen war, wurde nach neuen Möglichkeiten gesucht die Leistung von Computersystemen zu steigern. Heterogene Systeme haben sich als eine leistungsfähige Alternative herausgestellt. Im Kontext dieser Arbeit besteht ein heterogenes System in der Regel aus einem mit Software programmierbaren Prozessor und einem konfigurierbarem Hardwarebeschleuniger. Durch den für jede Anwendung speziell konfigurierten Hardwarebeschleuniger können heterogene Systeme eine bessere Leistung als ein herkömmlicher Prozessor erreichen.

Wegen ihrer größeren Komplexität ist es schwieriger für heterogene Systeme Anwendungen zu entwickeln als für ein herkömmliches, rein auf einem Software-Prozessor basiertem System. Da zur Programmierung der Soft- und Hardwareanteile verschiedene Sprachen verwendet werden müssen und zusätzliche, spezialisierte Hardware-Kenntnisse erforderlich sind, erhöht dies die Kosten der Entwicklung.

Diese Arbeit stellt das Compilerframework *Nymble* vor, welches es ermöglicht, ein heterogenes System allein mit einer Hochsprache zu programmieren. Nachdem der Entwickler in der Hochsprache festgelegt hat welche Teile in Hardware ausgeführt werden sollen, erzeugt Nymble ein Programm für den Prozessor, die Hardwarekonfiguration und alle Schnittstellen zwischen der Soft- und Hardware.

Alle von Nymble unterstützen Systeme haben gemeinsam, dass sich der Software- und Hardwareteil einer Anwendung einen gemeinsamen Speicher teilen. Da es sich bei dem Speicher um externen RAM mit hoher Zugriffslatenz handelt, ist es notwendig ein Cachesystem zwischen Speicher und Hardware einzufügen. Dies sorgt dafür, dass Speicherzugriffe zwischen sehr kurzer oder langer Latenz variieren können, in Abhängigkeit davon ob die Daten bereits im Cache verfügbar sind.

Um potentielle lange Latenzen zu überbrücken, präsentiert diese Arbeit ein innovatives Ausführungsmodell, das die simultane Ausführung mehrerer Threads in einem gemeinsamen Hardwarebeschleuniger erlaubt. Zusätzlich ermöglicht das Modell, an bestimmten Punkten in der gemeinsamen Pipeline die Reihenfolge der Threads dynamisch zu ändern. Diese Fähigkeit wird dazu verwendet, dass ein Thread, welcher auf einen Speicherzugriff warten muss, durch andere nicht wartende Threads überholt werden kann. Dadurch können diese unabhängig vom wartenden Threads ihre Berechnungen ausführen und so Latenzen überbrücken.

Bisherige Arbeiten verwendeten ein Ausführungsmodell, in dem jeweils nur ein Thread im Hardwarebeschleuniger aktiv sein konnte und im Falle eines Speicherzugriffs durch einen anderen Thread ausgetauscht wurde. Durch den Aufbau der Hardware kam es hier oft dazu, dass viele der Ressourcen einen signifikanten Anteil der Laufzeit brach lagen.

Durch seinen innovativen Aufbau verteilt das neue Ausführungsmodell mehrere Threads dynamisch über die Pipeline. Dadurch werden mehr Ressourcen gleichzeitig sinnvoll genutzt und eine bessere Auslastung erreicht. Weiter erreicht das simultane Ausführen mehrerer Threads einen ähnliche Durchsatz wie mehrere simultan ausgeführte Kopien eines Hardwarebeschleunigers, welcher jeweils nur einen Thread unterstützt.

Das neue Ausführungsmodell macht es möglich, den erhöhten Durchsatz mit einer verbesserten Effizienz zu kombinieren. Durch das gegenseitige Überholen von Threads ist es möglich das neue Modell effektiv mit einem geteilten Speicher über einen Cache zu verwenden.

Im Vergleich, zwischen vier Kopien eines Beschleunigers für jeweils einen Thread und einem Beschleuniger für vier Threads (beide durch Nymble erzeugt), wird eine Ressourceneffizienz von bis zu Faktor 2,6x erreicht. Dabei sind vier simultane Threads bis annähernd 4x mal so schnell wie vier auf einem Beschleuniger sequentiell ausgeführte Threads. Verglichen mit anderen, besser optimierten Compilern, erreicht Nymble immer noch eine bis zu 2x bessere Laufzeit mit einer 1,5x Ressourceneffizienz.

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich bei der Anfertigung dieser Arbeit begleitet und unterstützt haben.

Zuerst möchte ich meinen Eltern Gisela und Klaus Huthmann danken, die es mir ermöglicht haben, mich auf meine Arbeit zu konzentrieren.

Auch meine Schwester Alexandra Huthmann gab mir moralische Unterstützung. Meine Familie stand immer hinter mir und dieser Beistand half mir sehr dabei diese Arbeit zu schreiben.

Ein besonderer Dank gilt auch Prof. Dr. Ing. Andreas Koch, der meine Arbeit betreut hat. Neben den Diskussionen und Vorschlägen zu meiner Arbeit, möchte ich mich besonders für das Lektorat meiner Manuskripte bedanken.

Weiterhin möchte ich mich auch bei meinen Kollegen aus der TU Darmstadt bedanken. Die abendlichen Diskussionen mit Benjamin Thielmann haben zur Inspiration meiner Arbeit geführt. Durch die Unterstützung von Julian Oppermann und Björn Liebig konnte ich mich auf die Entwicklung meines Ausführungsmodels konzentrieren. Auch Florian Stock stand immer bereit um Probleme mit der Infrastruktur schnell zu lösen und hatte für viele Fragen eine Antwort. Es war mir ein Vergnügen mit meinen Kollegen zusammenzuarbeiten.

Abschließend möchte ich mich auch bei all meinen Freunden bedanken, welche mich in dieser Zeit begleitet haben.

Contents

1 Introduction

For the last 40 to 50 years it seemed the improvement of computer performance was bound to increase without ever slowing down. Moore's Law [Moo06] was an early observation that the number of transistors inside a CPU is doubled every 18 months. This observation motivated many researchers to keep this rate going up until recently.

In the beginning, this rate could be kept by continuously decreasing the size of each transistor. It was noticed by Dennard [Den+74], that a reduction of a transistor's size by factor of two comes along with a reduction in power consumption by a factor of four. However, coming into the 2000s, this could no longer be achieved. The power usage reduction for transistors could not keep up with the reduction of the size. In turn, it becomes increasingly difficult to cool new chips, because more power is concentrated on a smaller area. This led to a stagnation in the increase of clock frequencies for new chips, which provided a major portion of the improvement of CPU performance.

To further increase performance, the goal changed to put multiple computation cores into a single chip to distribute the power across the chip and thus allowing better cooling. With this it was possible to stay within Moore's Law. However, it became obvious, that in the future, new ways are necessary for even further performance increases. It is assumed that it will not be possible to reach higher performance cost-efficiently by increasing the number of transistors.

Due to this situation, the goal is shifting to use transistors more effectively. This can be achieved by trading programmability for more efficient usage with specialised accelerators. Examples for this are chips for 3D graphics or video acceleration.

This reduced programmability limits the application of such fixed accelerators. To combine the advantages of general purpose CPUs and accelerators, so called heterogeneous systems are developed. In these systems computationally intensive parts are offloaded to dedicated accelerators while less intensive calculations stay on the CPU.

Different methods exist to implement such accelerators. The first method is to develop and implement an accelerator for only a single specific application. So called Application-Specific Integrated Circuits (ASICs) implement the accelerator as a fixed custom accelerator hardware in silicon. While providing the best performance and efficiency, ASICs are the most expensive way to implement a custom accelerator like coprocessors for encryption. Because of their high non-recurring

expenses for development, ASICs are not viable for small quantities. Another, more flexible method is to use Field Programmable Gate Arrays (FPGAs) which can be re-programmed. Though FPGAs have a lower clock frequency and thus provide less performance than ASICs, the re-programmability allows a flexible use for many applications.

One of the biggest challenges in the application development for heterogeneous systems is that a programmer not only has to develop the software but also has to program the hardware. This requires software and hardware development skills, increasing the cost of development and reducing the productivity compared to software only development. The different components of a heterogeneous system make it also more complex to develop for. To reduce the programming difficulty and improve the productivity it would be favourable if the whole system could be programmed using just a software language. To allow this, it is necessary to develop a compiler which automatically translates the software into hardware programming.

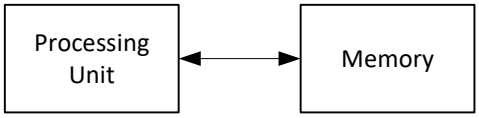
These challenges are acknowledged by different academic works (for more details, see Chapter 8). To understand the differences in target architectures a short overview of computing system architectures is presented.

The architecture of computing systems can be broadly grouped into one of the following classes. A typical computer (see Figure 1.1a) consists of a processing unit (the CPU) and an attached memory. As this memory is normally not directly in the CPU both communicate via a bus. As mentioned earlier, today it is common to put multiple cores into a single CPU, which can understood as multiple processing units connected to a single memory shared between the units (see Figure 1.1b).

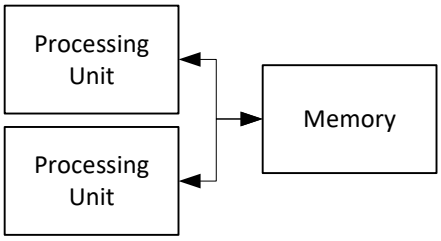
As the latency of memory could not be decreased by the same degree as the processing speed increased, it became necessary to use memory closer to or even on the processing unit (see Figure 1.1c) or else the memory would bottleneck the whole system [Bac78]. In CPUs this localised memory is typically used as cache for the external main memory. By having the data, which is currently worked on, closer to the processing unit allowed to reduce the latency of the memory accesses and thus improve the system performance. Of course, local memory can also be used together with multiple processing units (see Figure 1.1d).

Finally, there are the cases of having a processing unit without external memory and having such a processing unit connected to one of the other variants over an additional bus (see Figure 1.1e). In that case, the processing unit of the original system is used as a master which has to transfer all data to the slave accelerator before it can work on its local data. This leads back to the aforementioned heterogeneous systems.

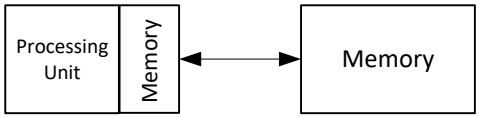
Many of other academic HLS compiler target the less complex model of an accelerator with its own memory and generally assume that all necessary data is already in that memory. With their focus on local memories, many issues of us-



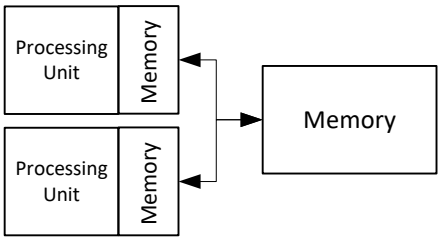
(a) Single processing unit with external memory



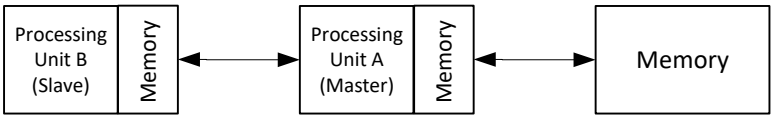
(b) Multiple processing units and shared external memory



(c) Single processing unit with local and external memory



(d) Multiple processing units with local and shared external memory



(e) Different processing units

Figure 1.1: Overview of computing system architectures

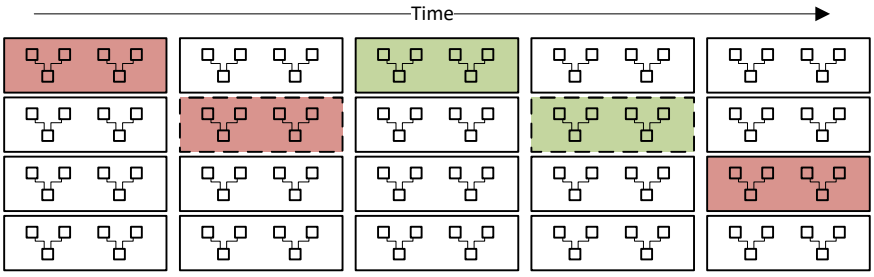


Figure 1.2: Temporal multi-threading by replacing waiting threads

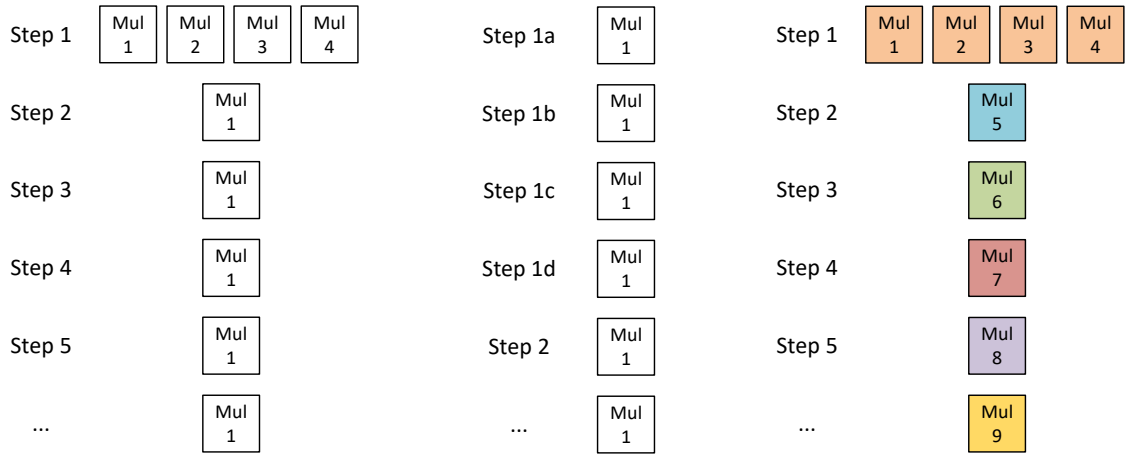
ing shared main memory (through a cache system) do not have to be handled. However, the focus of this work is improving the efficiency of accelerators in a heterogeneous system using shared main memory. Here, both the accelerator and CPU are sharing the same memory (with virtual addresses and pointer).

The fundamental problem is that it is impossible to predict the time required for accessing the shared memory. If the actual time is longer than the predicted, the accelerator has to wait for data. However, when the accelerator is waiting for an access, the resources of the accelerator are unused because there is no data to process. Some works aim to improve this efficiency problem by letting the accelerator work on a different set of data. For more information, see the related work in Chapter 8.

One method to create multiple data sets is to split a problem into tasks which can be computed independently. So if the accelerator waits for data of one task, it can switch to working on another task. This switching of the accelerator between multiple task is called multi-threading or more specifically temporal multi-threading.

Figure 1.2 shows the behaviour over time for two tasks or threads in an accelerator. The first thread (red) runs until it reaches the second stage of the accelerator where it has to wait for data (dashed outline). The waiting thread is then exchanged with another thread (green) which also runs until it has to wait for data at the same stage. However, now the data for the first thread is available and thus the thread is switched back into the accelerator.

Another fundamental problem is that some operators in the accelerator lie idle for a significant portion of the execution time as they are not required in each step. For example, in a pipeline (see Figure 1.3), one step might use four multiplication operations but all other steps only use a single multiplication. So most of the time, the remaining three operators lie idle (1.3a). By sharing a single real multiplication operator between the four logical operations in the first step, the efficiency can be improved, because this single operator is now always in use (1.3b). However, this comes with the cost that the performance is reduced as the first step takes more time to execute.



(a) Without operator sharing **(b)** With operator sharing **(c)** Simultaneous multi-threading

Figure 1.3: Operator sharing example

Extending this sharing to multi-threading means to increase the utilization of operations using multiple simultaneous threads. In the example, multiple threads should be organized in such a way that always one thread is utilizing the four multiplication operations (1.3c). The difference here is that operators are not shared in a single thread but are utilized by multiple threads. Until now, multi-threading in FPGA based accelerators was only used for simple-structured applications.

While this kind of simultaneous multi-threading can improve the performance [LRS83], most prior research (see Chapter 8) focused on creating multiple copies of the accelerator to execute multiple threads in parallel. This was done because there was no method to combine simultaneous and temporal multi-threading in a single accelerator instance with variable latency memory accesses.

With this work I want to show that it is possible to combine these three points:

1. Temporal switching between threads to hide memory latency
2. Simultaneous execution of multiple threads to improve performance
3. Resource sharing between multiple threads to improve efficiency

To achieve this, this work presents a pipeline model with a dynamic distance between iterations. In pipelines, the distance between two iterations is the most important indicator for performance. The smaller the distance, the higher the throughput of the pipeline will be. However, in typical pipeline implementations, the whole pipeline is controlled by a single controller. If the pipeline has to wait at any point, all iterations have to wait, resulting in a reduction of performance. This is increased even further with the integration of simultaneous multi-threading as now multiple threads are waiting.

By using a distributed controller, the presented pipeline model can dynamically change the distance between iterations, which in turn is one of the basics of the proposed simultaneous multi-threading. By integrating modules to store thread context into the pipeline, the enhanced model is able to dynamically reorder threads.

1.1 Research Contributions

This thesis provides the following two main contributions. First, the Nymble HLS compiler framework which is also used in a number of other research projects. Second, a novel multi-threaded execution model where simultaneously executed threads share the resources of a single accelerator instance.

1. Nymble

   Nymble is a high-level synthesis framework which can generate FPGA based accelerators for a variety of heterogeneous target systems. The accelerators can be implemented with one of many execution models and in the future the compiler will be able to mix and match multiple models according to the requirements of the application.

   While this work uses C as the input language for the high-level synthesis, Nymble was used as a back-end for a domain specific language compiler [Hut+10a; Hut+10c].

   In the source code, the programmer can select the hardware part of the application using simple pragmas. The compiler supports heterogeneous systems with shared memory using virtual addresses and pointers.

   Before Nymble was published in [Hut+13; HOK14], it was used in research on load-value prediction [Thi+11; THK11b; THK11a; Thi+12; THK12].

   In this research, Nymble was used to create accelerators where the memory accesses can be executed speculatively. When a memory access is executed, a heuristic (load value prediction) returns a speculated value for that access. This allows the pipeline to continue, even when a memory access cannot be finished immediately. When the actual access is finished, the speculated value is compared to the actual value. In case both are equal, a fast-path is used to commit the correct value in later pipeline stages and clear buffers. In case the values differ, these buffers are used to redo the calculation with the corrected value.

   Other works based on Nymble were published in [LK16; SOK16].

2. Multi-threaded execution model

   The main part of this work is the multi-threaded execution model. The aim of this model is to improve performance and efficiency of pipelined accelerators. By allowing other threads to overtake a stalled thread in the pipeline, the throughput is increased. At the same time the resource efficiency is improved because all threads share the same accelerator and its operators. Only at some pipeline stages, additional logic is introduced for the reordering of threads. By selectively placing these multi-threaded stages in a insofar single-threaded pipeline, the model achieves a higher efficiency than just using copies of the accelerator for each thread.

   The model was published in [HOK14; HK15].

Another contribution is the inclusion of hardware functions. Hardware functions are very similar to functions in software, as they can be used from multiple points in the pipeline without implementing them multiple times. This reduces the resource consumption with only a small impact on the execution time for many benchmarks.

Further, this works compares two of the most commonly used execution models for accelerators (basic block and pipelined, see Section 2.2). It will be shown that both have advantages and disadvantages for implementing applications with different characteristics. While the focus of this work lies on the pipelined model, this work shows a method how the presented multi-threading technique can be adapted to the basic block model. It will even demonstrate the possibility of arbitrarily mixing both models. In the future, this will allow the compiler to select the most suitable model for each part of the application. Table 1.1 shows an overview of all supported execution models and where they are presented and evaluated.

		Section	
Execution Model	Threading	Presented	Evaluated
Pipeline with Static II	Single	3.2	9.1.2.1
Pipeline with Dynamic II	Single	3.3	9.1.2.1, 9.7
Pipelined with Dynamic II	*Multi*	*4*	*9.2 - 9.3, 9.5 - 9.8*
Basic Block	Single	5	9.4
Basic Block	Multi	5.1	9.4

Table 1.1: Supported hardware execution models (main model is highlighted)

The presented multi-threading is extensively evaluated to show that it can achieve a 2x improvement in resource efficiency compared to simply duplicating a single-threaded pipeline. At the same time, multi-threading can improve the throughput by a factor of 3.5x compared to consecutively executing four threads. The multi-threaded model is especially efficient for applications with floating point

operations, as the multi-threading overhead is reduced with increased operator complexity.

In comparison to other academic HLS compilers, Nymble achieves comparable results in terms of area and runtime. However, as Nymble focuses on the pipelined model, there are some benchmarks (where a basic block models performs better) for which Nymble is slower. Also, as Nymble targets a scalable shared memory model instead of simpler local memories, Nymble uses more resources for its more complex memory handling and the resulting dynamic iteration intervals in the pipelines.

However, with the inclusion of multi-threading and complex operations (floating point) Nymble comes much closer to comparable implementations in terms of resource consumption. For the runtime, it even can execute some benchmarks faster than other sequential implementations.

1.2 Thesis Structure

This work begins with an explanation of compiler basics (Chapter 2). This is followed by explanation and comparison of two general execution models (Section 2.2). These models describe how the instructions of an application are executed.

Based on these models, the two different hardware executions models supported by this work (basic block and pipelined) are explained in detail (Chapter 3). Note that at first, these models only allow for single-threaded execution. The multi-threaded enhancement of the main execution model (pipeline) will be explained in the following chapter (Chapter 4).

It is followed by a method to integrate the multi-threading previously shown into the basic block execution model (Chapter 5), which is generally used by the other academic compilers. In the future, the compiler could be extended to select whichever execution model is more suitable for a given part of the application.

After an explanation how the compiler translates an application for a heterogeneous system to hardware (Chapter 6), the different target systems for heterogeneous execution are shown (Chapter 7). Before the evaluation of the multi-threaded execution model and a comparison with other academic compilers (Chapter 9), an overview of related work is given (Chapter 8). Here, some academic compilers are discussed in detail. Because of references to the presented hardware execution models and multi-threading, the related work was not placed as an earlier chapter.

After the comprehensive evaluation, the work ends with an explanation of future work (Chapter 10) and final conclusion (Chapter 11).

2 Basics

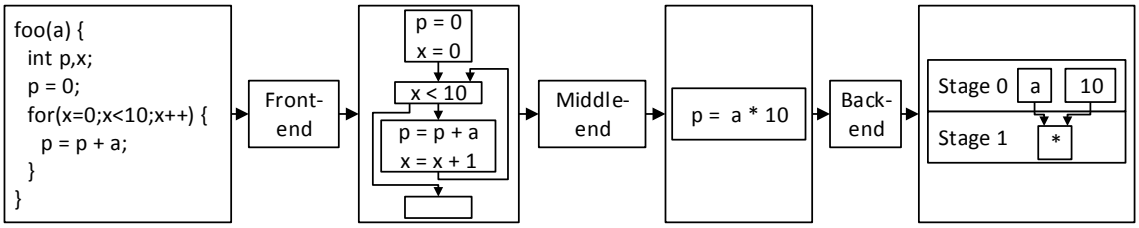


Figure 2.1: Typical flow from front- to middle- to back-end

A typical compiler consists of a front-, middle- and back-end (Figure 2.1). The front-end reads and translates the source code into a format easier processable. This format is called the Intermediate Representation (IR). The middle-end is responsible for applying target system independent optimizations on the IR. Finally, the back-end is responsible for applying target system-dependent optimizations and generating the programming of the target system.

The development of a complete set of front-, middle- and back-end is not an easy task. Additionally, all target system-specific steps are only in the back-end. So to focus the development on back-end, it is efficient to use existing front- and middle-ends. gcc [SE16] (Section 8.2.1) and LLVM [LA04] (Section 8.2.2) are the most well known open-source projects providing a set of front- and middle-ends (also including some back-ends).

But the IR of these projects is designed for compilation of applications for general purpose processors with a generally sequential execution. For the parallel execution targeting FPGAs, this generic IR has to be translated into a suitable hardware IR.

IRs are used to represent a program's semantic behaviour. How the application is then actually executed is described through the execution model. For understanding this process, the basic program representation and execution models are presented in the following sections.

2.1 Program Representations

This section describes the basic program representations used or referenced in this work.

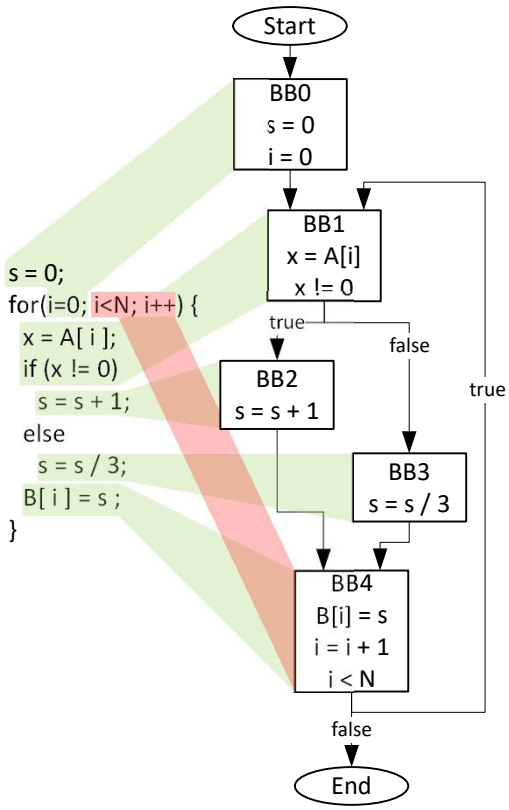


Figure 2.2: Conversion of source code to CFG

2.1.1 Control-Flow-Graph (CFG)

A commonly used IR in compilers is the Control-Flow-Graph (CFG) [All70]. CFGs were developed to represent all possible paths of the *control flow* through each function. A CFG is a directed graph $G = (V, E)$, with nodes V and edges E. Each node contains a list of instructions. Only the last operation in each node can be a (conditional) reference to another node, defining the next node the application executes. Such a node is called a Basic Block (BB). Within a BB, the control always flows from the first to the last instruction in that block. An edge (a, b) means that the control can flow from node a to node b. Each instruction consists of an operation and its operands. Additionally, the result of the operation can be stored in a variable for later usage.

Figure 2.2 shows the transformation from source code to CFG. The mapping of statements to IR instructions is marked by coloured areas. Edges marked with `true` or `false` indicate the control flow in accordance to the result of a conditional instruction at the node's end. The CFG has total ordering equal to the source code order. It is usually defined by the order given by the programmer through the source code.

2.1.2 Data-Flow-Graph (DFG)

Accompanying the CFG, another representation is often used for a number of optimizations and analyses. A Data-Flow-Graph (DFG) [Rod69; Den80] is used to show all paths *data* can take through the application. It is a directed graph $G = (V, E)$, with nodes V and edges E. Each node represents a single instruction. An edge (a, b) means that the result of instruction a is used in instruction b.

DFGs contain no informations about the control-conditions. Thus, it is not possible to represent the behaviour of multiple assignments to the same variable in BBs on different control-flow paths. In the example, BB2 and BB3 both contain an assignment to `s`. The following "store to array `B`"-operation has to select one of the previous assignments. It would be theoretically possible to create a DFG for each variant, but the number of DFGs would increase exponentially with the number of such assignments. A partial example for that tentative method is shown in Figure 2.3. To avoid this, the specialised representation form SSA (see Section 2.1.3) was developed. This form ensures that there is only a single static assignment to each variable.

Ignoring this and just creating a DFG results in a DFG as can be seen in Figure 2.4. This DFG has multiple merge-points of data-flows (denoted by zigzag arrows), where the behaviour of which one of the merged data should be used is undefined.

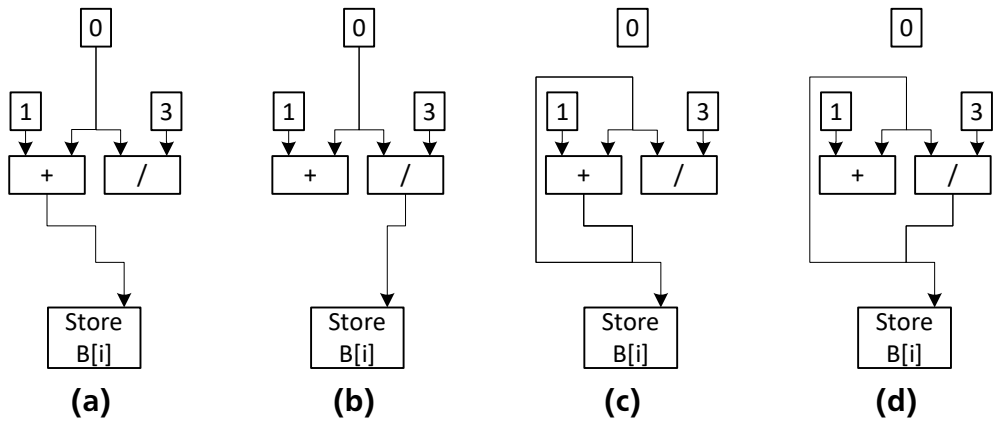


Figure 2.3: Tentative DFG variants for variable s and the store

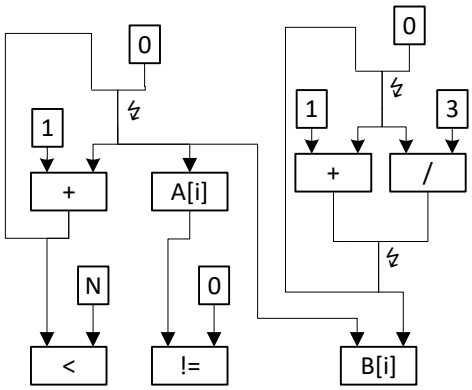


Figure 2.4: Data-Flow-Graph (DFG) for CFG in Figure 2.2
(zigzag arrows denote the merge of two data-flows, which results in undefined behaviour)

2.1.3 Static Single Assignment (SSA)

This form is used to allow easier representation of the data-flow while incorporating the control-conditions.

Each assignment to a variable V creates a new version V_i of the variable. The value is static in the loop iteration that created it. This way, for each use of V_i it can be immediately seen where the value is assigned. Because of this property, the form is called Static Single Assignment (SSA). At nodes with more than one incoming edge, where possible control-flows merge, different versions of the same variable have to be merged into a new version. The pseudo-instruction ϕ selects the version corresponding to the control-flow actually taken.

SSA is often used with but not limited to CFGs. For example, any source code can also be written in SSA form. The transformation from a normal CFG to SSA was made popular by [Cyt+91].

With the additional ϕ-instructions, it is now possible to solve the problem with merge of data-flow from multiple sources mentioned in Section 2.1.2. For each merge, the ϕ-instructions has all sources as its parameters and is integrated into the DFG in the same manner as all other instructions. Figure 2.5 shows the resulting SSA form for the CFG and DFG in Figure 2.2.

2.1.4 Memory-Dependency-Graph (MDG)

The MDG shows the dependencies between instructions that can access memory. Two memory instructions are dependant on each other when they can access the same memory location. The MDG is a directed graph $G = (V, E)$, with nodes V and edges E are either inter- or intra-iteration dependencies. Thus, E is split into

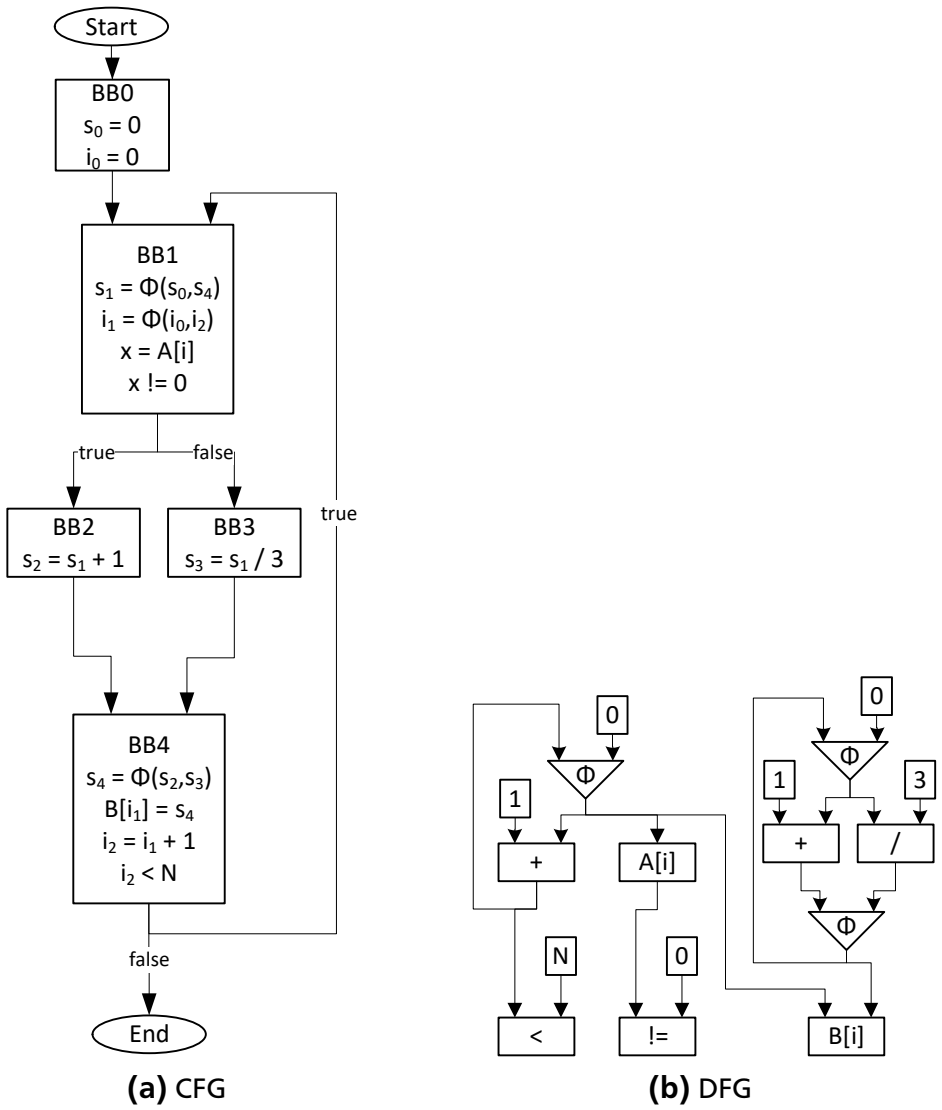
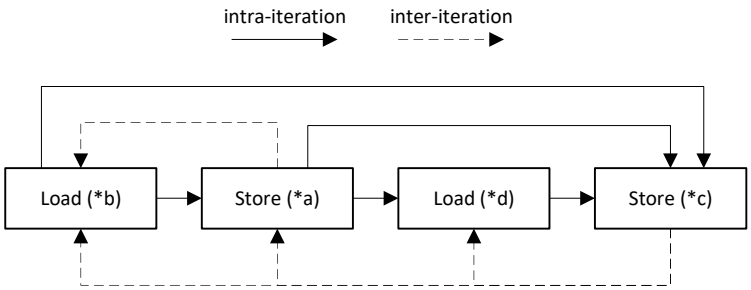


Figure 2.5: Static Single Assignment (SSA) form of CFG and DFG in Figure 2.2

```
x = 0;
while (x < 5) {
  tmp = *b;
  *a = tmp + 1;
  x = tmp / 2;
  *c = *d + 1;
}
```

(a) Code



(b) Memory-Dependency-Graph

Figure 2.6: Memory-Dependency-Graph example

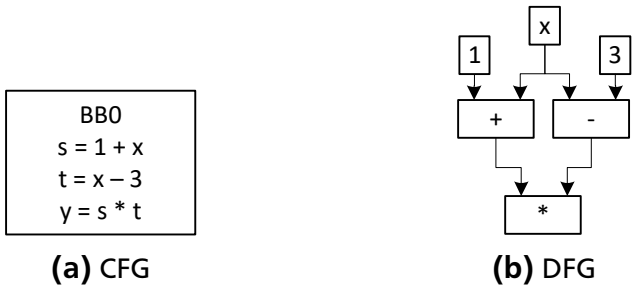


Figure 2.7: Sequential CFG / parallel DFG comparison (ignoring obvious optimizations)

the subsets E_{inter} and E_{intra}. Each node represents an instruction that can access memory. An edge (a, b) means that the instruction a and b can access the same memory location and that a is executed before b in CFG order. The edge is an intra-iteration dependency if a and b are executed in the same loop iteration, otherwise it is an inter-iteration dependency. An edge (a, b), where both a and b are reading accesses, is generally omitted because it does not cause any problems when omitted. Figure 2.6 shows an small code snippet and its MDG.

2.1.5 Control-Data-Flow-Graph (CDFG)

The CFG model is used to represent sequential applications, but it cannot be easily used to represent parallel executed instructions. On the other hand DFGs can easily express instructions that can be executed in parallel. Operations which have no (transitive) connection can be executed in parallel, as they have no data dependencies. Figure 2.7 shows such a case where the parallel execution of the addition and subtraction cannot be easily seen in the CFG (a), while it is obvious in the DFG (b).

But as a DFG has no information about the control-flow or control-conditions, it cannot model the impact of control-flow or control-conditions on the execution. Another form is required, which combines the representation of parallel execution with the control-flow information. An extended DFG is constructed out of a SSA CFG by integrating the ϕ-instructions into the DFG.

The resulting CDFG is a directed graph $G = (V, E)$ with nodes V represent instructions and edges E are either data- or control-dependencies. Thus, E is split into the subsets E_d for data-dependencies and E_c for control-dependencies.

The control-flow or control-conditions are integrated into the CDFG in two ways. First, control-dependant data is propagated similar to ϕ-instructions in the CFG. But as the CDFG contains no CFG like control-flow edges, ψ-instructions are used instead. These ψ-instructions use pairs of data- and control-dependencies (compared to the pairs of data-dependency and control-flow

edge in ϕ-instructions). If the control-dependency of an input pair is evaluating to `true`, the corresponding data is propagated. These control-dependencies are constructed in such a way that only one dependency per ψ-instruction is `true`.

Additionally, ϕ-instructions only require the *immediate* incoming CFG edges for the propagated data selection and rely on the sequential order of the CFG. If the condition of the last branch instructions before the edge is included, it is called an *immediate condition*. However, in the CDFG ψ-instructions require the *complete condition* under which data is selected. The *complete condition* is a combination of the conditions of *all* branch instructions which are evaluated before reaching the BB that contains the instruction.

To clarify this distinction and show why complete conditions are necessary, the examples in Figures 2.8 and 2.10 are used. The first example in Figure 2.8 shows a code segment, its SSA-CFG and the alternatives for evaluating actual values for the control conditions in the CDFG. The CDFG is constructed with only immediate control conditions to show why this is not enough. Note that the CFG in Figure 2.8b and CDFG in Figures 2.8c to 2.8e omit the normally present edge for the default case of the switch for a cleaner example. When a condition is evaluated to `true`, all edges from that condition are coloured in green. When a condition is `false`, the edges are red. This can be changed by logical operations.

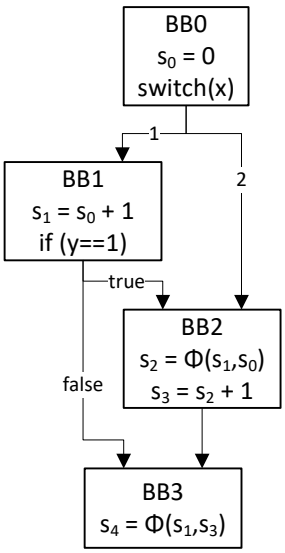
For the ψ-instruction at the bottom, marked by a, the selection of the appropriate value seems to work correctly for the cases shown in Figures 2.8c and 2.8d. But when examining Figure 2.8e, the ψ-instruction's control-dependencies both evaluate `true` at the same time which makes it impossible to select the correct value. The cause for this is that in CFG, the control flow would move from BB0 to BB2 and then to BB3. This does not evaluate the condition `y==1` at all, but in the CDFG, such non-executed instructions are generally only handled the correct value selection through ψ-instructions (Instructions with side effects need extra handling, shown shortly). A correctly constructed CDFG with *complete conditions* as shown in Figure 2.9 does not have this problem.

The second way of using control-dependencies is to control the execution of instructions with side-effects such as modification of data through memory accesses. Only if the control-dependency evaluates to `true` is the instruction executed. In the CFG, the position in the control-flow determines when the operation is executed. In the CDFG, the control-flow is transformed into a control-dependency which has to evaluate `true` to execute the operation.

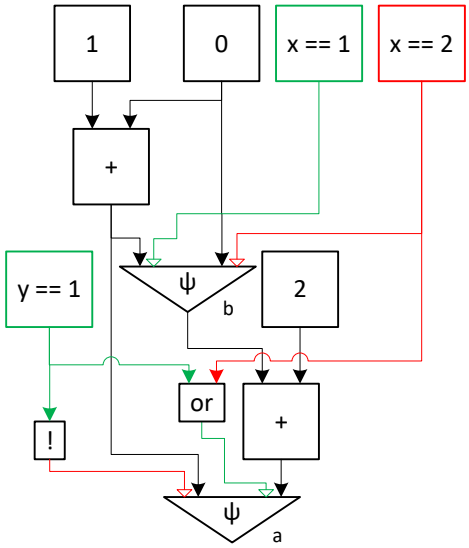
In the second example, these control-dependencies for operations with side-effects are shown. In Figure 2.10, three memory access are shown in different branches created by two branch instructions with the conditions `A` and `B`. The example contains no ϕ-instructions. Looking at the CDFG with only immediate conditions (Figure 2.10b), it can be easily seen that again immediate conditions are not sufficient. When `A` and `B` are used separately, it can happen that either the

```
s = 0;
switch(x) {}
  case 1:
    s = s + 1;
    if(y==1)
      break;
  case 2:
    s = s + 2;
}
```

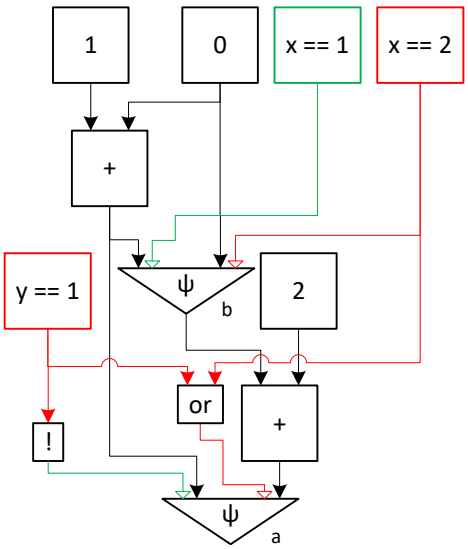
(a) Code



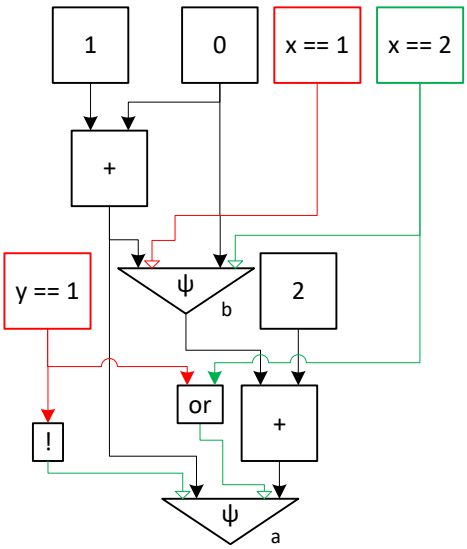
(b) SSA-CFG



(c) CDFG alternative 1



(d) CDFG alternative 2



(e) CDFG alternative 3

Figure 2.8: CDFG with only immediate control conditions

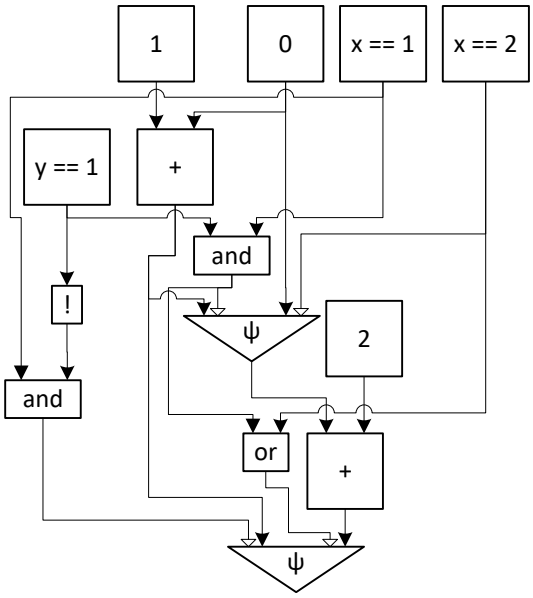
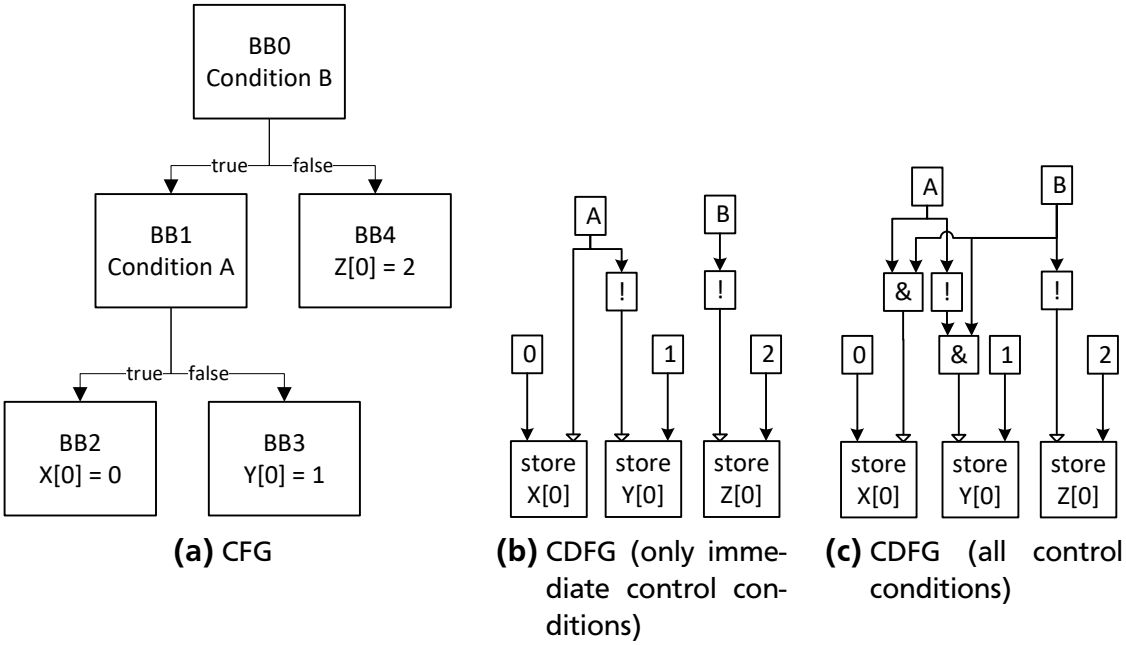


Figure 2.9: CDFG with complete condition for example in Figure 2.8

store to `X` or `Y` from BB2 or BB3 is executed together with `Z` in BB4. The solution again is to use the complete condition as shown in Figure 2.10c.

The algorithm to compute these control-conditions and the construction of the CDFG will be shown in Section 6.4.2.



(a) CFG

(b) CDFG (only immediate control conditions)

(c) CDFG (all control conditions)

Figure 2.10: Multiple memory accesses

2.2 Execution Models

This section will show two general models which describe how an application is executed. Execution models describe the order in which the work is executed. This order can be either decided statically at compile time or dynamically at runtime. Most models use a varying degree of both. However, both basic model shown here use a statically decided order to simplify their explanation.

Both models will start with a simple CPU model as an example for their theoretical base model. Then a simplified version of the model is applied to a hardware execution model. These simplified models are then compared. Finally, the hardware execution models that are implemented by this work are shown in Chapter 3.

The example CPU contains a number of execution phases: Instruction Fetch (IF), Instruction Decode (ID), Execute Instruction (EX), Memory Access (MA) and Register Write Back (WB). Each phase requires one time step or clock cycle to execute its function.

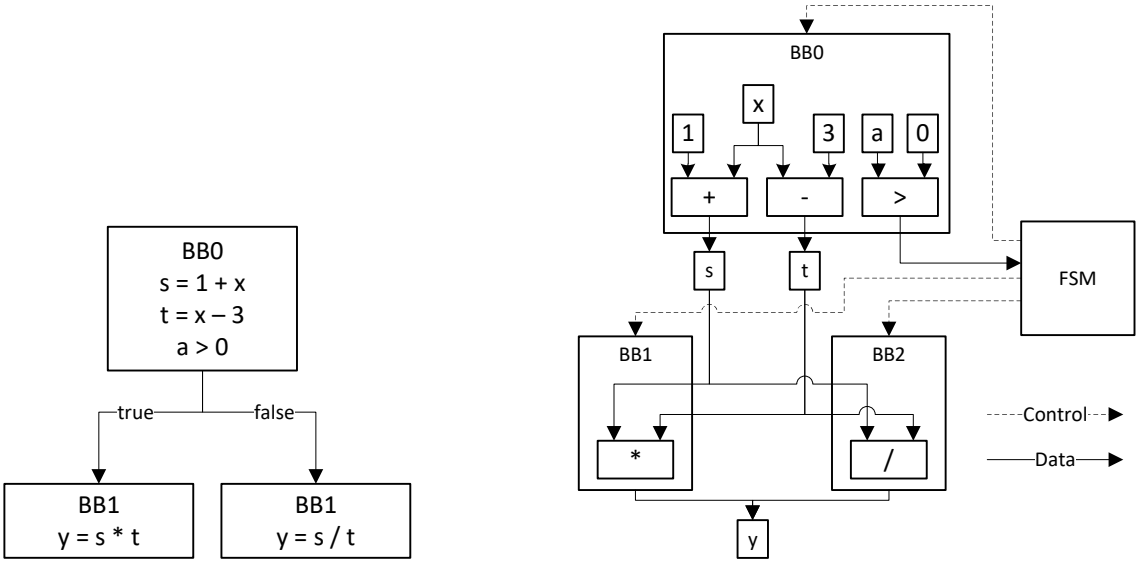
2.2.1 Basic Blocks FSM

For the first model, the CPU can execute only one instruction at time. This means that another instruction can be loaded by the IF unit from memory, only when the previous instruction has been completely executed in all necessary phases. During ID phase it can be decided to skip the execution of EX, MA or WB when it is not needed for a specific instruction. That way the latency of single instruction can vary between 2 and 5 cycles.

The theoretical adaptation of that example into a model is very closely linked to the CFG model. For an application in the form of an CFG, each Basic Block (BB) is seen as a single state in the FSM whose instructions are executed together. While the instructions in the BB can (depending on the implementation) be executed somewhat parallel (example is following shortly), the instructions of only one BB are executed at a time because the FSM has only a single state. Thus, the overall sequential nature CFGs is kept, resulting in simpler control conditions. The execution time in this model is the sum of all BB's execution time.

Figure 2.11b shows such a FSM, executing the operations of each BB parallel, while sequentially going through the BBs. BB0 calculates the values for s and t and then stores them in a register in-between the basic blocks. The result of the comparator goes to the FSM which controls which BB is executed. Internally, it uses the BBs directly as its state.

When not all operations in a BB can be executed in a single clock cycle, the BB is generally split into multiple BBs so that all operation can be assigned in such a way that the operations in the individual blocks can be executed in a single clock cycle. However, there are also operations that take more than one clock cycle by



(a) CFG **(b)** BB FSM with DFG

Figure 2.11: Executing CFG operations in parallel with a BB FSM (ignoring obvious optimizations)

itself. Here it is not possible to split the BB, as a single operation cannot be split. So instead of splitting the BBs into more BBs, the BBs are subdivided by Execution Basic Blocks (EBBs) (so called for a lack of common distinction to BBs).

Assuming for the example in Figure 2.11 that the division in BB2 takes ten clock cycles, BB2 is divided into ten EBB (see Figure 2.12). Also, both BB1 and BB2 is assigned to a single EBB each. Now, the FSM does not use BBs as states any more, but these EBBs. This assignment of operations to one or more EBB is done by a scheduler during compile time that creates a static schedule.

As each EBB takes exactly one clock cycle, the execution time of a system with this model can be easily calculated by the sum of all executed EBB, which in consequence means that it depends on which EBB are executed (more in Section 2.2.3).

This execution model is used by many HLS compilers (see Section 8.3).

2.2.2 Pipeline

For the second model, the CPU can execute *multiple* instructions simultaneously. This is done by ordering the execution phases into a pipeline with one stage per phase. In this pipeline, the data from one stage can only flow into the next stage. But as soon as the data from an instruction leaves a stage, the stage can work on the data of a new instruction. None of the stages can be skipped, so the time to execute a single instruction is the same for all instructions. Assuming no parallel execution of instructions, the number of cycles to execute multiple instructions

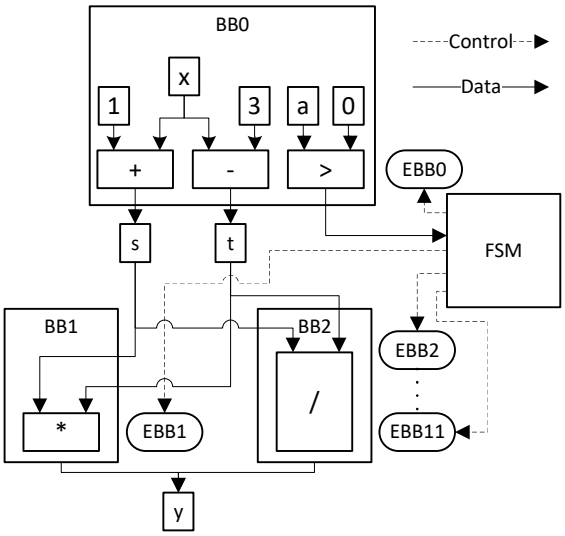


Figure 2.12: BB FSM with EBBs for example in Figure 2.11 (division takes ten clock cycles)

Inst. 1	IF	ID	EX	MA	WB		
Inst. 2		IF	ID	EX	MA	WB	
Inst. 3			IF	ID	EX	MA	WB
Cycle	1	2	3	4	5	6	7

Figure 2.13: Pipelined execution of 3 instructions

is the number of instructions minus one, plus the number of cycles for the last instruction. Figure 2.13 shows the execution of three pipelined instructions. For each cycle and instruction the figure shows which stage is used by each instruction.

The theoretical adaptation of that example into a model based on the CDFG program representation is not as straightforward as for the previous representation. Remembering that the CDFG models all control and data dependencies, it is possible to assign (or *schedule*) each instruction to a specific pipeline stage. These dependencies were either inter- or intra-iteration dependencies. While intra-iteration dependencies only affect the schedule of a single iteration, the inter-iteration dependencies and their schedules affect when the next iteration can be started. Only if the instructions at the source of the dependency have been executed, the next iteration can be started. The time interval between the re-execution of each instruction is the so called Initiation Interval (II). As the time is generally measured in clock cycles and each cycle the iterations can move one pipeline stage ahead, the II is also a metric for the distance between two data dependent iterations. It

can be used to determine the effectiveness of the pipeline, smaller II being better. Assuming that the whole pipeline generally has the same II, the II cannot be smaller than the length of the longest inter-iteration dependency.

A small II is important, because the execution time of the pipelined model depends mostly on it. As after II cycles a new iteration can be started, the overall execution time can be calculated by multiplying the number of executed iterations n_{iter} with the II (assuming that each stage takes one clock cycle). The length of the pipeline l_{pipe} is only really important for the last iteration as only then it is necessary to wait until it reaches the end of the pipeline (all other iteration obviously reach the end before the last iteration). However, as the last iteration already moved II stages through the pipeline before it is known that it is really the last iteration, these II number of stages can be removed from the runtime. Thus, the execution time of the pipelined model can be calculated as $II \times (n_{iter} - 1) + l_{pipe}$.

Figure 2.14 shows an example of such a pipelined CDFG. It is a very simple loop, adding one to a value read from memory and immediately writing it back to another memory location. The pipeline uses the *init* operation (which is true only once) to select the initial zero for the loop counter `i` in Stage 0. Stage 1 contains multiple operations, the increase of the loop counter by adding one, the loop of array `A[i]` from memory, and the test for loop termination (the *end* operation is, however, in the last stage). In Stage 2, the value read from memory is then increased by one and finally written back to array `B[i]` in Stage 3. The pipeline has an II=2 because the increase of `i` requires one clock cycle.

Figure 2.14a shows the behaviour of the pipeline by depicting which stages are active during the pipeline execution. It shows that the execution time calculated with II=2, $n_{iter} = 2$ amd $l_{pipe} = 4$ indeed is $2 \times (2 - 1) + 4 = 6$.

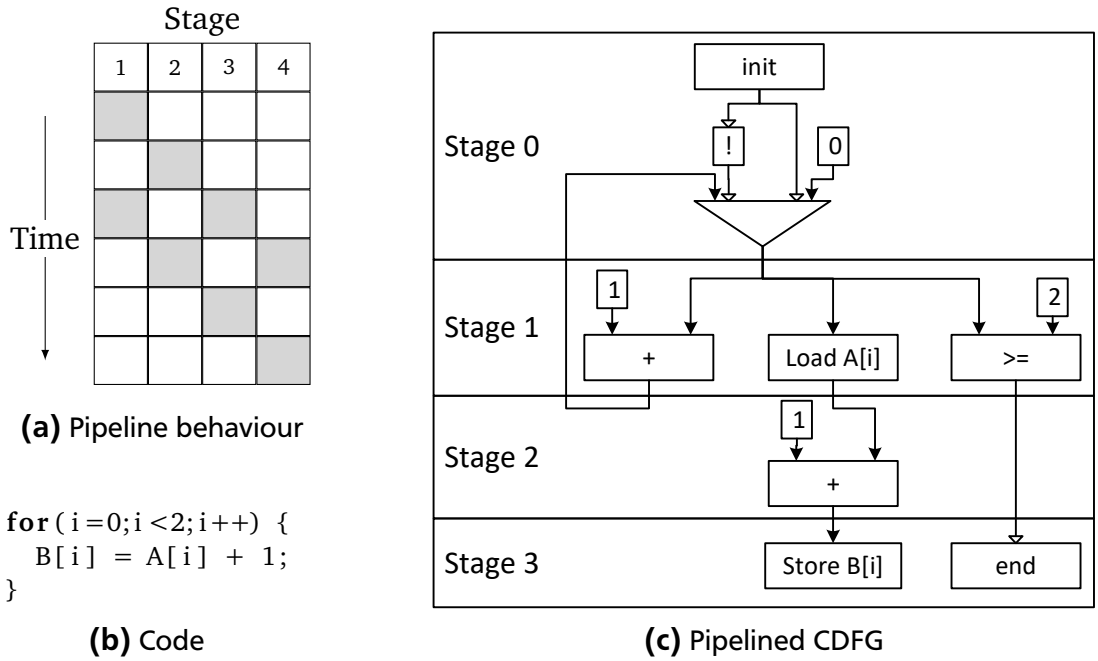


Figure 2.14: Example of a pipelined CDFG (II=2)

2.2.3 Comparison

Both of the previously presented models have their advantages and disadvantages for different applications. This section will discuss the general differences between both models. For that purpose two examples, one that favours each model, will be used for comparison.

The first example (Section 2.2.3.1) will show an application which favours the BB FSM execution model. Also, it will be shown that the favourable model can change depending on the input data. The second example (Section 2.2.3.2) then shows an application which favours the pipeline model.

The discussion will talk about two paths through the loop bodies of the example applications. For each application both paths will be marked by red circles for path P_A and a green dashed line for path P_B.

2.2.3.1 Example Favoring BB FSM Model

The first example application is defined by a small loop. At the start of each iteration a value `x` is loaded from the array `A`. This load uses the loop variable `i` as its offset. Depending on the value of `x`, the value `s` is increased by one or divided by 3. In each iteration the value `s` is then written to the array `B`, with `i` as its offset. The code of this can be seen in Figure 2.15a.

The CFG, shown in Figure 2.15b, is unbalanced in the loop body. This means that the execution time differs for different paths through the CFG. The path P_A (marked by red circles) through the loop body is very short compared to P_B (marked by a green dashed line). This is caused by different operations in BB2 on P_A and BB3 on P_B. While BB2 contains a fast addition (assumed to be executed in one clock cycle), the divider in BB3 is assumed to take ten clock cycles. In fact all operations besides the divider are assumed to take one clock cycle to execute. Note that in a typical system, the array accesses in BB1 and BB4 can take a variable amount of cycles through the cache systems. But for simplicity of the examples, this is omitted here.

As explained in Section 2.2.1, the BB FSM model's execution time depends on which EBBs are executed, which in this example depends on the input data. In Figure 2.16c, the EBB structure of the application is shown. During the compilation the contained operations in each BB were transformed into DFGs and then statically scheduled to EBBs. As written as in Section 2.2.1, the execution time for each path corresponds with the number executed EBBs on that path (assuming that each EBB takes one clock cycle). P_A executes EBBs 1, 2 and 13. P_B executes 1, 3 to 12 and 13. Note that both paths have the same EBB in the beginning and end. The difference in the middle is caused by the decision (BB1) between the addition (BB2) or division (BB3). As P_A executes only three EBBs and P_B executes twelve EBBs, the execution time for P_A is shorter than for P_B. From this follows that, depending on data read from `A[i]` in BB1, the execution time of an iteration and, in turn of the entire applications, differs.

For the pipeline model, the application is transformed into a CDFG which is then scheduled into pipeline stages, as can be seen in Figure 2.16b. Again, in the pipelined model the execution time largely depends on the II. And the II is defined by the longest inter-iteration dependency. In this example, this is the path P_B, as shown as in Figure 2.16b. P_B has a length of 12 stages (0 to 11), thus taking 12 cycles to execute one iteration. Thus resulting in an II of 12. The store to array `b` in stage 12 is executed in parallel to stage 0. The behaviour of the pipeline is shown in Figure 2.16a, where active stages are denoted by grey boxes.

As shown as in Section 2.2.2, the execution time of the pipeline model can be as $II \times (n_{iter} - 1) + l_{pipe}$. n_{iter} is the number of executed iterations and the pipeline has a length of l_{pipe} stages. As after each II number of cycles, a new iteration is started, it thus takes $II \times (n_{iter} - 1)$ cycles to start the last iteration. This last iteration then has to move through the whole pipeline, requiring l_{pipe} cycles.

Knowing both execution models, it is now possible to calculate the execution time for them. For the BB FSM, it is necessary to know execution for each possible path, which are P_A with an execution time of $t_A = 3$ and P_B with $t_B = 12$. In the pipelined model the execution time depends only on $II = 12$ and $l_{pipe} = 13$. For both models the number iterations will be varied in $n_{iter} = [1, 9]$.

To determine the number of executions (or execution count) EX_A and EX_B for both paths P_A and P_B in BB FSM model it is necessary to define the input data for decision in BB1. Because of the very simple decision between zero and non-zero, the input data is defined as the number of zeroes z_A contained in the array `A`. The overall number of executed iterations corresponds to the size N of array `A`. Thus, the execution time can be calculated with the function $t_A \times (N - z_A) + t_B \times z_A$. When executing 6 iterations, the time for the Deterministic Finite Automaton (DFA) model is between 15 and 73 cycles for 0 and 6 zeroes in array `A`, respectively. For the pipeline model, the execution time is always 73 cycles. All times can be seen in Table 2.1.

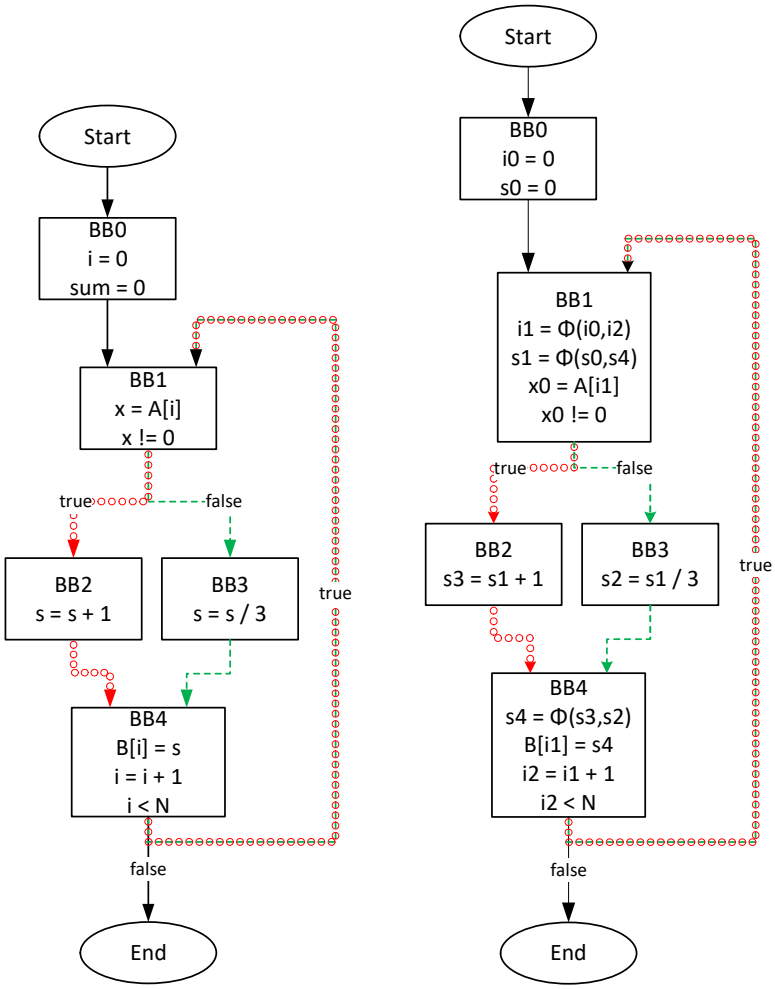
From these numbers it can be seen that the pipeline model generally performs worse than the BB FSM model in case of such an application structure. Only when all iterations in the BB FSM model have to use the longer path P_B, the execution time for both models become the same. This means that an application where long operations, which are not executed in each iteration, are influencing the length of the longest inter-iteration dependency and in turn the II, is more suitable for execution with the BB FSM model.

	BB FSM with z_A =										Pipeline
n_{iter}	0	1	2	3	4	5	6	7	8	9	
1	4	13	-	-	-	-	-	-	-	-	13
2	7	16	25	-	-	-	-	-	-	-	25
3	10	19	28	37	-	-	-	-	-	-	37
4	13	22	31	40	49	-	-	-	-	-	49
5	16	25	34	43	52	61	-	-	-	-	61
6	19	28	37	46	55	64	73	-	-	-	73
7	22	31	40	49	58	67	76	85		-	85
8	25	34	43	52	61	70	79	88	97	-	97
9	28	37	46	55	64	73	82	91	100	109	109

Table 2.1: Execution time (#clock cycles) depending on the number of zeros in the input data z_A and number of iterations n_{iter}

```
int A[N], B[N];
s = 0;
for(i=0;i<N;i++) {
  x = A[i];
  if(x != 0)
    s = s + 1;
  else
    s = s / 3;
  B[i] = s;
}
```

(a) Code



(b) Basic block structure of CFG

(c) SSA CFG

Figure 2.15: Comparison of basic block FSM and pipelined CDFG: advantage FSM (division takes ten clock cycles)

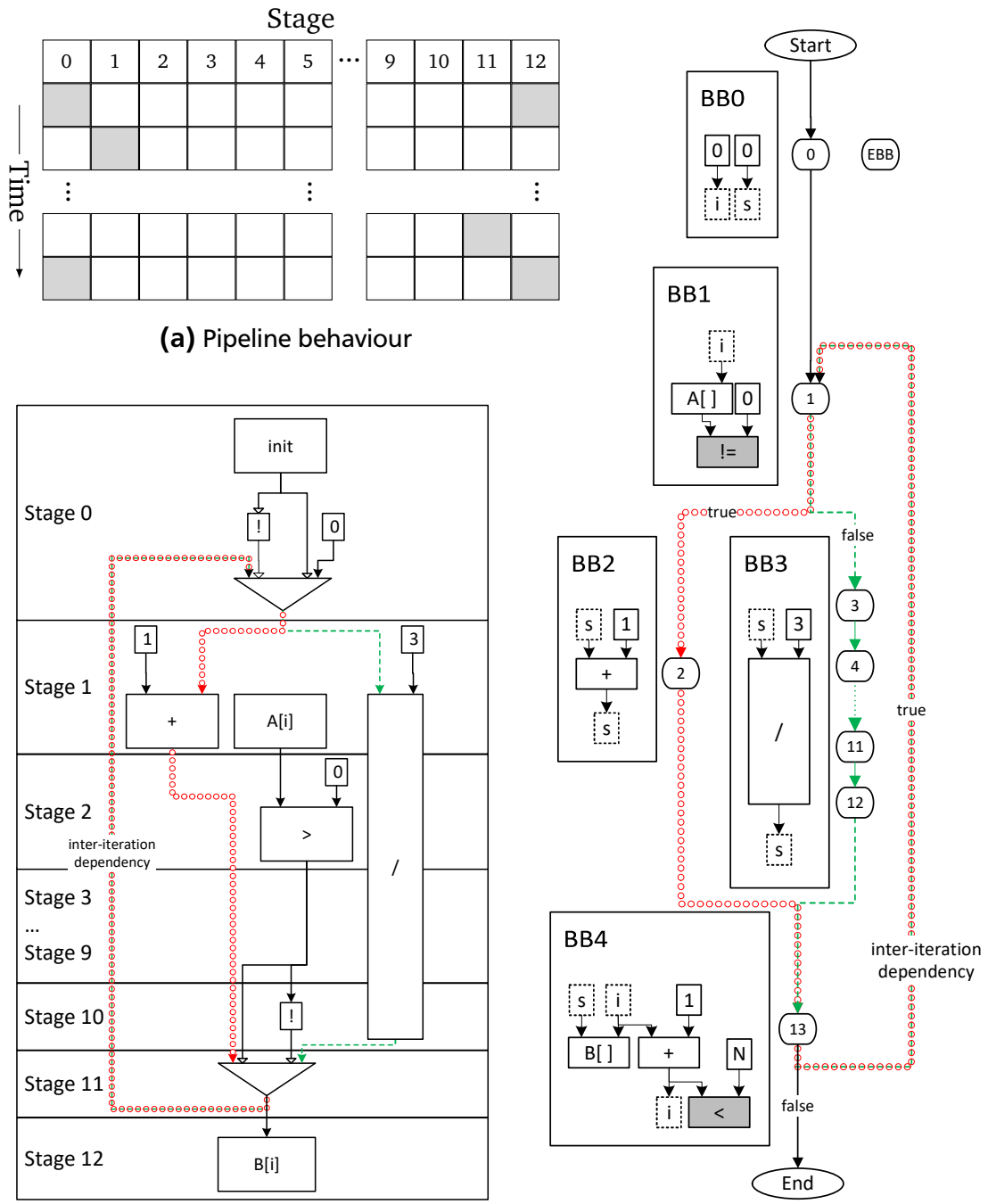


Figure 2.16: Comparison of basic block FSM and pipelined CDFG: advantage FSM (division takes ten clock cycles)

2.2.3.2 Example Favoring Pipeline Model

The second example application is, like the first, defined by a small loop. Initially, the value of `s` is initialized to 0, and then modified by each loop iteration. At the start of each iteration, a value `x` is loaded from the array `A`. This load uses the loop variable `i` as its offset. Depending on the value of `x`, the value `s` is increased or decreased by one. The value in `s` is then copied to `t` and divided by 3. Finally, `t` is written to the array `B`, with `i` as the offset.

Unlike the first application, the CFG (shown in Figure 2.17b) the execution time of the operation in each BB are equal for "parallel" BBs on different paths through the CFG. This means that CFG is balanced, unlike the previous unbalanced example. The addition in BB2 of path P_A (red circles) and the subtraction in BB3 of P_B (green dashed line) need the same number of cycles to execute. Both P_A and P_B then go through BB4 containing the division and write.

Figure 2.18c shows the EBB structure for the DFA model after the transformation into DFGs and scheduling to EBBs. P_A uses the EBBs 1, 2 and 4 to 14. P_B uses 1, 3 and 4 to 14. Note that both paths have the same number (14) of EBBs and thus have the same execution time. Multiplied by the number of iterations (again $N = 6$) and adding the cycle for EBB0 the execution time of the application in the DFA model is 85.

For the pipeline model the application is transformed into a CDFG which is then scheduled into pipeline stages, resulting in the pipeline shown in Figure 2.18b. Unlike the previous example, the longest (and only shown as the loop counter `i` is omitted) inter-iteration dependency encompasses only a small part of the CDFG.

This allows for the parallel execution of multiple iterations, indicated by an II smaller than the length of the pipeline. This is shown by pipeline behaviour in Figure 2.18a, where active stages are denoted by grey boxes.

The BB FSM model cannot execute multiple BBs at the same time and thus has to execute all iterations sequentially. On the other hand, the pipeline model can initiate the next iteration immediately after the value `s` has been increased or decreased.

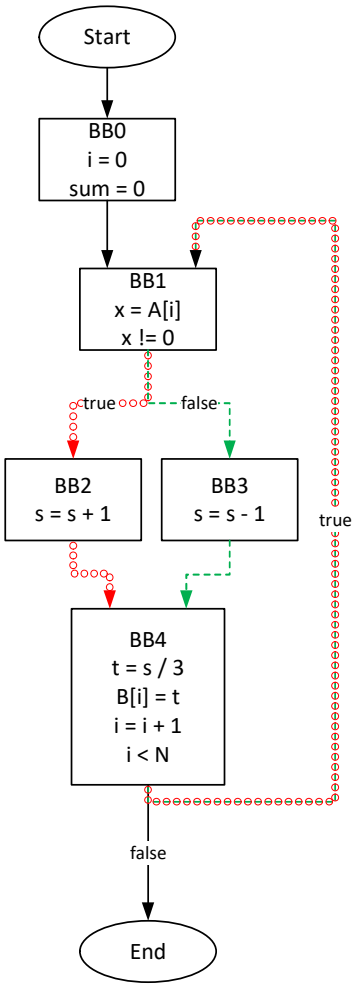
Again the execution of the pipeline model can be calculated with $II \times (n_{iter} - 1) + l_{pipe}$. With $II = 4$, $n_{iter} = 6$ and $l_{pipe} = 15$, the execution time of the pipeline model is 35 cycles. Table 2.2 shows a summary of executions times for different number of iterations n_{iter}.

n_{iter}	1	2	3	4	5	6	7	8	9
BB FSM	15	29	43	57	71	85	99	113	127
Pipeline	15	19	23	27	31	35	39	43	47

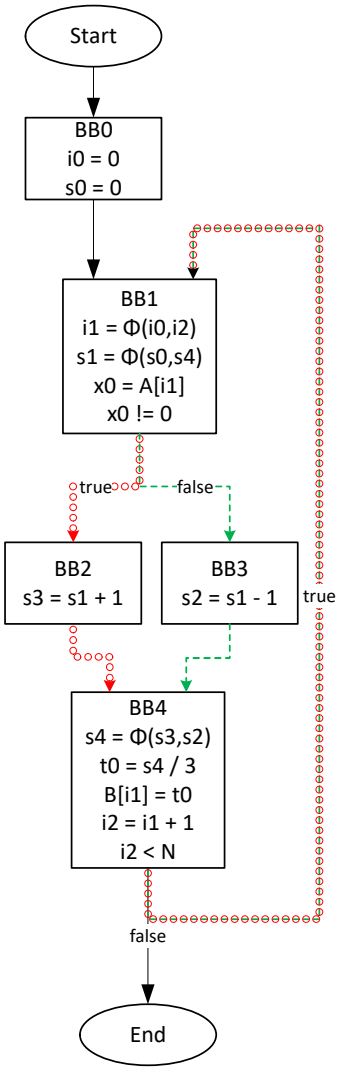
Table 2.2: Runtime (#clock cycles)

```
s = 0;
for(i=0;i<N;i++) {
  x = A[i];
  if(x != 0)
    s = s + 1;
  else
    s = s - 1;
  t = s / 3
  B[i] = t;
}
```

(a) Code



(b) Basic block structure of CFG



(c) SSA CFG

Figure 2.17: Comparison of basic block FSM and pipelined CDFG: advantage CDFG (division takes ten clock cycles)

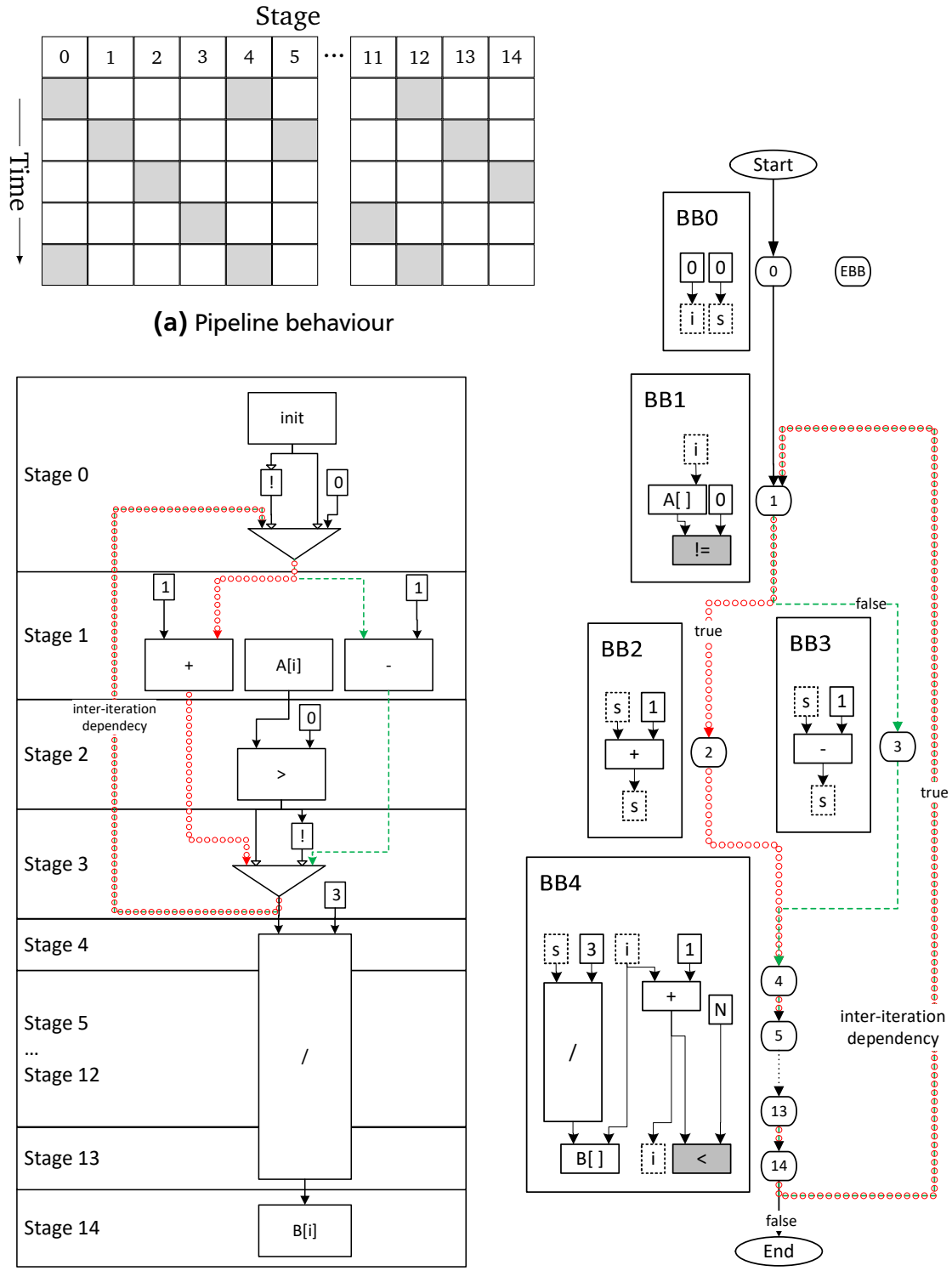


Figure 2.18: Comparison of basic block FSM and pipelined CDFG: advantage CDFG (division takes ten clock cycles)

2.2.3.3 Summary of Comparison

This section showed a comparison of the DFA and pipeline execution model for the use in application-specific hardware. It was shown that for both models applications exist, which execute faster on one of the models. In general, the pipeline model is better for applications which can be scheduled with a small II compared to the overall length. If an application is unbalanced and cannot be scheduled with a small II, the DFA model is better.

While most of the remainder of this work will focus on the pipeline model, later a method to mix both execution models will be shown in Chapter 5 and Section 10.1. This method will allow to chose the model which is the most appropriate for a given part of the application.

3 Hardware Execution Model

This chapter will describe how the applications are executed on the hardware. All presented execution models are variants of pipelined model (see Section 2.2.2) and use the CDFG representation of the application. In the execution model, the CDFG *instructions* are executed using hardware *operations*. In addition to the interaction between the operations (data and control dependencies), it will also be shown how the memory dependencies in the MDG are satisfied during the execution.

The main differences between the presented hardware execution models lie in the handling of the II. While the basic interaction between the instructions is generally the same for all variants, the way iterations are moved through the pipeline is vastly different between them. Nested loops, memory instructions and the handling of the MDG is also influenced by the handling of the II.

The discussion of the hardware execution models will be done on five different levels; HW-SW communication, Loop Hierarchy, Pipeline, Stage and Operation. The HW-SW communication level will show the interaction between the hardware and software. The Loop Hierarchy level will show how the application's loop hierarchy is handled in the execution model. The HW-SW communication and Loop Hierarchy levels are generally identical for all presented executions models. The pipeline level will show the interactions between the stages in the pipeline. Here it will be shown how the movement of loop iterations between the stages is controlled. The general concept of each execution model can be understood by only going as deep as the pipeline level. It also contains most of the differences between the execution models. Because of that, each model description will start with an example on the pipeline level. Then the stage level will show the interactions between operations. Finally, the operation level will show what is happening in each operation.

Additionally, the pipeline and subsequently each stage is split into data-path and controller. The data-path does all data evaluations (including logical conditions), while the controller decides which and when stages are executed.

3.1 General Concepts

This section will describe concepts common between all hardware execution models.

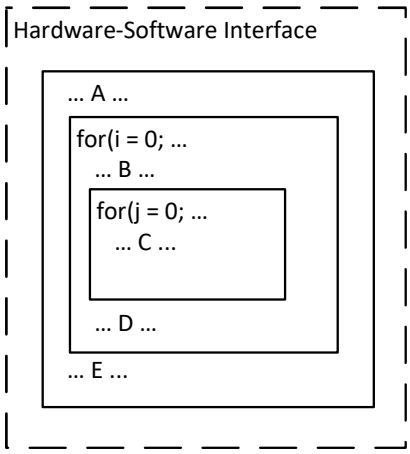
3.1.1 Loop Hierarchy

Each loop in the application is transformed into a separate CDFG, which is executed as a pipelined data-path. The loop hierarchy is modelled by placing sub-loops into their respective parent loops. In the parent, entire sub-loops are represented by a single operation that, if it is executed, starts the execution of the sub-loop.

Figure 3.1b shows an example of the nesting hierachy for the application in Figure 3.1a. Each box represents a CDFG for the corresponding loop. Note that the outermost code parts A and E are not really a loop. But in the execution model, they are treated as a loop that only executes a single iteration. The interface between the hardware and software parts is always around the outermost loop or CDFG of the hardware part.

```
int foo(int x) {
  int i;
  #pragma HARDWARE on
  ... A ...
  for (i = 0; i < N; i++) {
    int j;
    ... B ...
    for(j = 0; j < x; j++) {
      ... C ...
    }
    ... D ...
  }
  ... E ...
  #pragma HARDWARE off
  return i;
}
```

(a) Source



(b) Nesting Hierachy

Figure 3.1: Mapping of loops to nested CDFGs

3.1.2 Operation

Each instruction in the CDFG is implemented by an operation in the data-path of the pipeline. An Operation belongs to one the following three general types, which are classified by the number of clock cycles required to execute.

The basic type of operations can be executed in a single clock cycle. The second type, Multi-Cycle Operation (MCO) requires more than a single clock cycles but always the same number of cycles. The third type, Variable Latency Operation (VLO) requires an unknown number of clock cycles to execute.

Operations in the basic type often execute so quickly that multiple of these operations can be executed as a chain in a single clock cycle.

3.1.3 Multi-Cycle Operation (MCO)

As MCOs require more than one clock cycle, they are placed in multiple pipeline stages as each stage holds all operations that can be executed in a single cycle. So MCOs have an input, output and multiple intermediate stages. The input stage is the stage where they receive their input parameters and begin their calculation. The intermediate stages are the stages where the calculations proceed. The output stage is the stage where the calculation is finished and the output can be consumed by other operations.

Additionally, MCOs have to be completely pipelined in order to be integrated into the pipeline. If an MCO is not pipelined, it will be treated as a VLO, as they are not required to be pipelined.

3.1.4 Variable Latency Operation (VLO)

The pipelined model assumes that all operations have a fixed latency. This means, that at compile time, the number cycles necessary to execute the operation is known. Now, as a nested loop's execution time might be dependent on its parameters, an operation type is required that represents such operations with a variable latency.

These Variable Latency Operations (VLOs) are scheduled as single cycle operations, but when they are executed the pipeline is stalled until the VLO is finished. How the pipeline is stalled differs between the presented execution models. Currently, the only instructions which are represented by VLOs are nested loops and cached accesses to the shared memory.

3.1.5 Pipeline Hierarchy

Now that all general elements of the accelerator and the pipelines are known, the overall interaction can be explained. Figure 3.2 shows an overview of the connections and nesting of a complete accelerator. Starting on the left, the software and hardware communicate via the HW-SW Interface, which is also the top level of the nesting hierarchy shown in Figure 3.1b. This level contains buffers for input parameters and results of the accelerator which can be exchanged via direct communication. It also contains the control interface between the software and the accelerator.

Right next to the HW-SW interface is the top most loop of the application. This first loop is represented by a VLO in the interface. When this VLO signals the

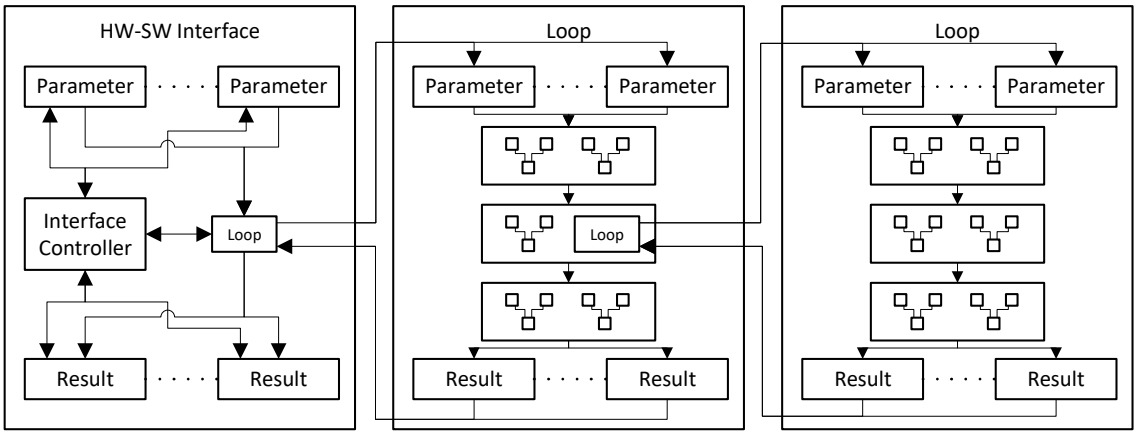


Figure 3.2: Hierarchy of nested loop pipelines and hardware-software Interface

completion of its execution, the control interface signals to the software that the accelerator is finished. The interface is also used to start the accelerator, when the software requests it. The nested loop itself contains additional buffers for its input parameters and results. The double buffering at this stage is acceptable to make sure that all loops are implemented in the same way. The loop itself is then executed with one of the execution models presented in this chapter.

Finally, next to the first loop is a second loop which is nested inside the first loop. This nested loop is represented as VLO which is integrated in the pipeline of the first loop. There is no limit to the loop nesting depth. Unless noted otherwise, all loops in an accelerator are executed with same execution model, as the first loop.

Later in this work, a method to re-use loop instances similar to functions in software will be shown.

3.1.6 HW-SW Communication

The communication between the software and the hardware accelerator is performed using registers and shared memory. Registers are used for controlling and exchanging small (in terms of storage size) amounts of data. The shared memory is used to exchange all other data values. The general invocation protocol of the hardware accelerator is shown in Section 3.1.6.4.

3.1.6.1 Registers

While the registers have a common memory-mapped interface to be accessed from the software, their purpose and usage on the hardware side are quite different. First there are the data registers to exchange scalar data values between the software and hardware. The number of data registers is application-specific. Second,

there are a fixed number of control registers. The interrupt register is used to signal when the hardware has a message for the software. The message is encoded in the value of the interrupt.

3.1.6.2 Shared Memory

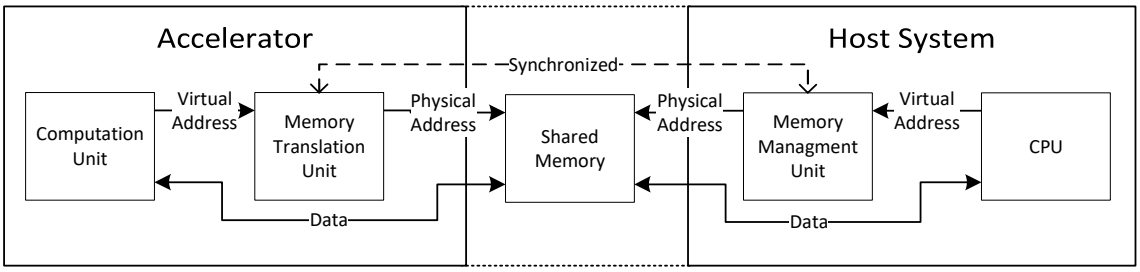


Figure 3.3: Shared virtually addressed memory between accelerator and host system (memory location is target system dependant)

To allow the use of pointers to virtual addresses in the application, the application runs in single continuous memory block which is shared between the hardware and software. Figure 3.3 shows this concept. For the software side on the host system, the mapping from virtual to real memory addresses is done with the normal memory mapping techniques of the operating system and MMU. On the hardware side, the mapping is done with a single entry MMU. This single entry MMU, which could better be called a Memory Translation Unit (MTU), just holds the base address of the shared memory block and is used with the offset encoded in the virtual addresses to calculate the real address.

The specifics of the shared memory depend on the target system architecture. For further details, see Chapter 7.

3.1.6.3 Hardware-to-Software Calls

The hardware-software communication model and the execution model include a way with which the hardware can call functions in the software part of the application. This differs from the normal model of executing an accelerator and waiting until it is finished, in that the accelerator can interrupt itself, let the software execute a function and continue its execution from the point where it was interrupted. In the source of the application, these hardware-to-software calls or Software Services (SWSs), as they will be called in this work, can be marked by pragmas similar to the hardware selection pragmas. A SWS is integrated into the pipeline hierarchy as a VLO in the loop containing the SWS, as shown as in Figure 3.4. A SWS is executed similar to a nested loop. It uses the same buffers for

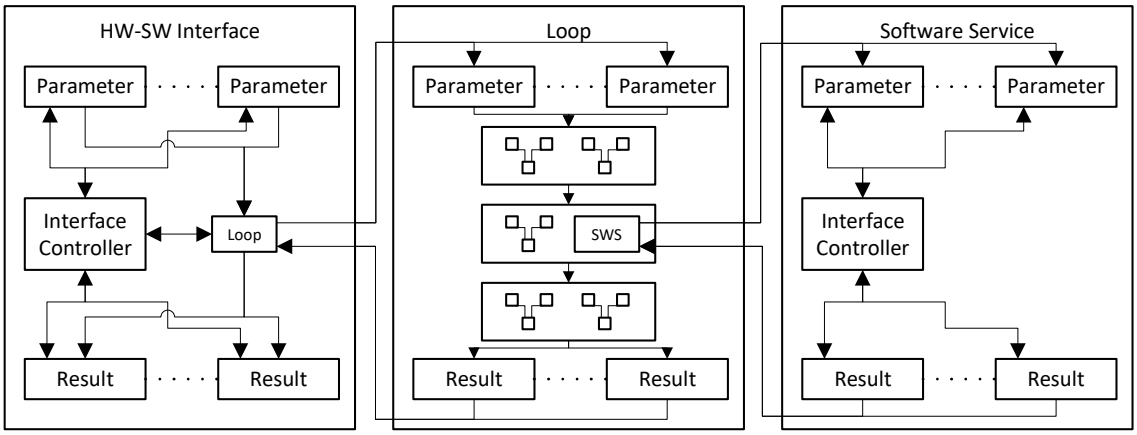


Figure 3.4: Hierarchy of loop pipelines, hardware-software interface and Software Service

the parameters and results of the SWS. But instead of executing a pipeline, the SWS uses the interface controller to communicate with the software and request the execution of a function. Because the same interface is used as for the normal communication, no additional communication channels are added.

As before, the details of the implementation are target system specific and are shown in Chapter 7.

3.1.6.4 Hardware Invocation Protocol

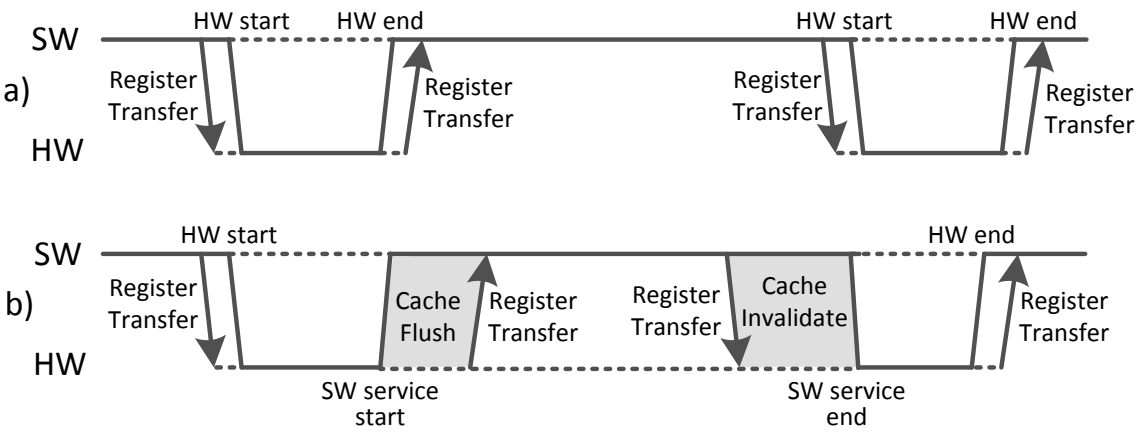


Figure 3.5: Accelerator invocation protocol (solid line signifies control, dotted line is stored state while suspended)

This section will describe the basic protocol which controls the exchange of data and context switches between the software and hardware parts of the application. The execution always begins with the start of the software part of the applica-

tion. This is the main executable, generated by the compiler. When started, the software executes all instructions up until the marked hardware part. When the hardware part is reached, the software writes the parameters of the hardware part to their respective input register. When this register transfer to the hardware is finished, the software starts the hardware calculation using the control registers. The hardware is then executed using on the executions models described here.

When the hardware is finished or requests a SWS, it sends a signal to the software indicating the cause and requested SWS. After reading the control registers, the software then can, depending on the application and the target system, instruct a flush of the shared memory cache to guarantee data integrity. After the optional flush, the results or parameters for the SWS are read by the software and the software continues or executes the SWS. When a SWS is finished by the software, it writes the SWS's results back into input registers of the hardware and signals that the hardware can continue. Depending on the application the shared memory cache is invalidated.

Figure 3.5.a shows the repeated execution of the hardware part of the application. In 3.5.b, a SWS is executed. The execution of a SWS is always inside the execution of the hardware part.

3.2 Pipeline with Static II

The first model is based on the pipeline model from Section 2.2.2 and is specified by the fixed II. All dependencies are resolved using a fixed schedule. This means that under no circumstances the distance (number of stages, see Section 2.2.2) between two iterations changes. The execution model will take all necessary steps to ensure this.

3.2.1 Example

Figure 3.6 shows an example for the execution of a pipeline with the static II model. This pipeline was constructed for an unspecified application which has a single memory access as its only VLO. Data dependencies required that this memory access is scheduled to Stage 3 and the resulting pipeline has an II of 3.

The first iteration enters the pipeline at Time $t = 1$ and continues into the next stage at $t = 2$. As the pipeline has an II of 3, at $t = 4$ the second iteration enters the pipeline. The memory access on Stage 3 is activated and assumed to stall. To keep the distance between the iterations constant, all iterations have to be stalled. At $t = n$ the memory access on Stage 3 finishes and the pipeline can continue. At $t = n + 2$ the third iteration enters the pipeline and the second iteration activates the memory access on Stage 3 again. The memory access again stalls, thus stalling

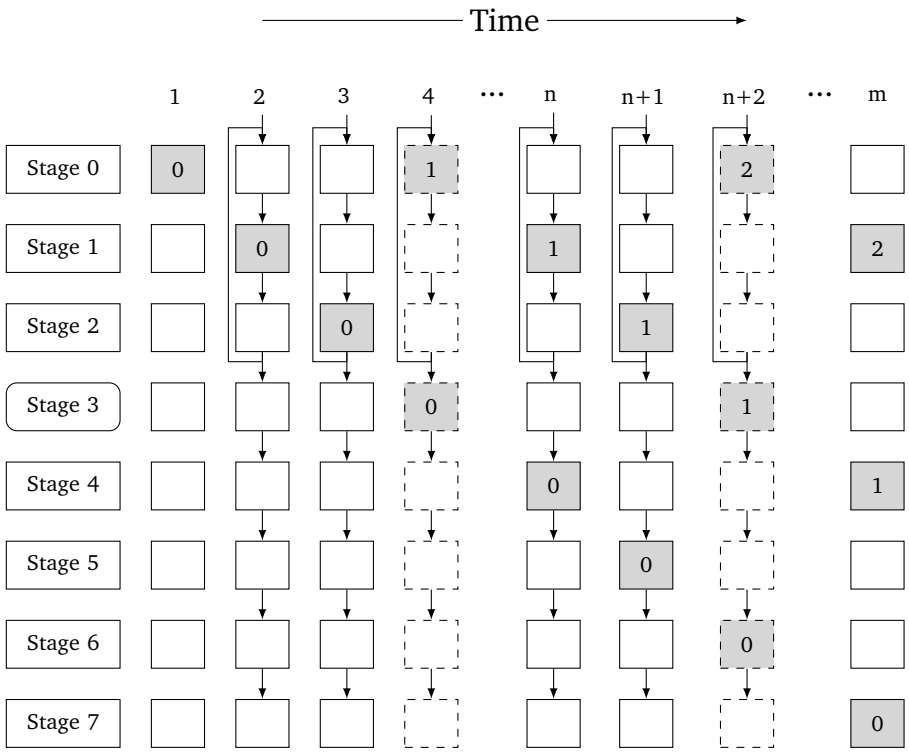


Figure 3.6: Static Initiation Interval behaviour (II=3)
Iterations are shown as numbered boxes
Dashed iterations are stalled, rounded stages contain VLOs

the pipeline. The memory access finishes again and the pipeline can continue at $t = m$.

This example shows the biggest disadvantage of the static II execution model. Stalling of one iteration immediately stalls all other iterations to ensure that dependencies are not violated.

3.2.2 Pipeline level

On the pipeline level of the static II model, the handling of all VLOs and memory dependencies is done globally. In the example it was shown that all stall decisions always have to affect all pipeline stages to keep the static II. In fact, the memory dependencies are resolved by creating a schedule (see section 6.6) which does not violate them. By guaranteeing that this schedule is always kept, which is automatically done by maintaining the fixed II, memory dependencies are resolved at compile time.

As shown in Figure 3.7, the pipeline of this model is split into controller and data-path. In the data-path, the iterations, meaning all data values belonging to

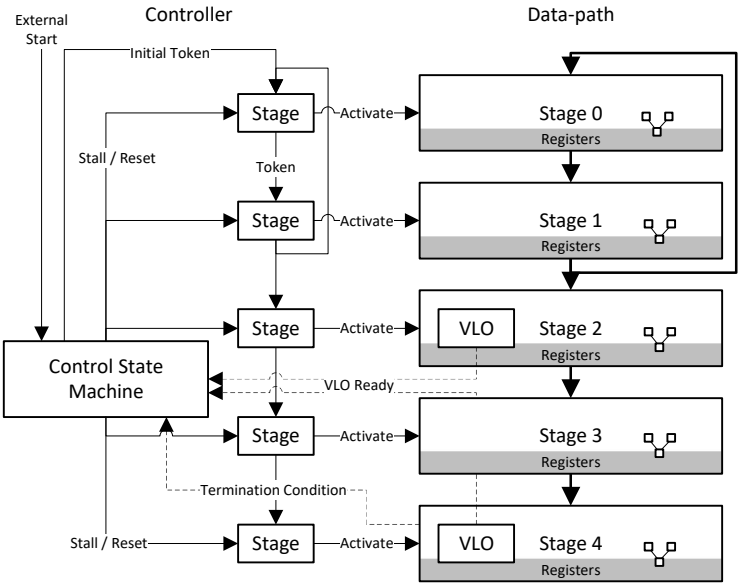


Figure 3.7: Static II model on the pipeline level ($II = 2$)

one iteration, move from one stage to the next as long as the controller does not stall the whole pipeline. While at least one VLO is executed, the complete pipeline is stalled by the control state machine. When all currently executed VLOs signal their completion the pipeline is allowed to continue.

As this stall behaviour is global to the whole pipeline, the controller decisions are all global. While there is no stall, the controller activates all successors of stages that currently contain an iteration to move these iterations into the next stage. An activated stage uses the data from the previous stage to calculate the result of its instructions. The results are stored in registers on their respective stage until they are consumed by the next stage.

For a pipeline with n-stages, the controller is comprised out of n registers, which hold the activation tokens of currently active stages. Each cycle where the pipeline is not stalled, these tokens are shifted into the next stage. The activation token from stage $II - 1$ is also looped back into the first stage register to create the next iteration. If the loop termination condition, which is calculated in last stage, is `true`, the pipeline is stopped. The control state machine inserts the initial token into the first stage upon an external start signal. This also resets all currently stored values from an earlier execution of the loop pipeline.

The pipeline is terminated when the termination condition, which is always located in the last stage, is evaluated `true`. The controller then signals the completion of the loop pipeline to its parent loop.

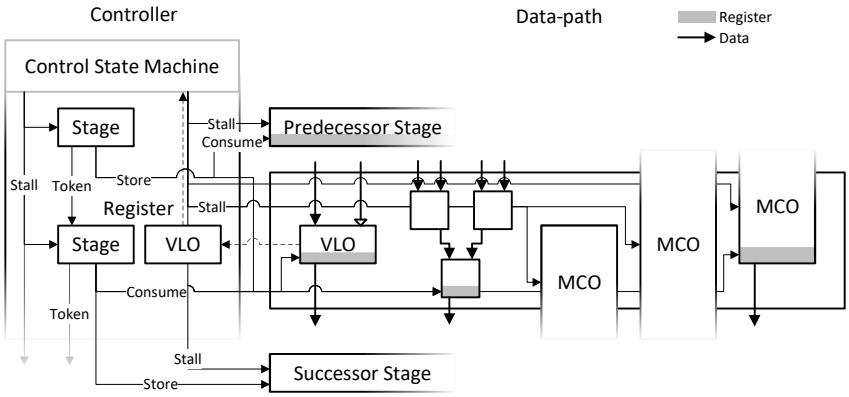


Figure 3.8: Controller excerpt and pipeline stage with chained basic operations, a VLO, input, intermediate and output stage of MCOs. Shaded parts of operations contain a register to store data until it is consumed by the successor stage

3.2.3 Stage level

As all the controller decisions are global, only the connection details of all signals which are used on a single stage are shown on the stage level. Figure 3.8 includes the signals in the controller, between the controller and data-path and in the data-path. It will also be shown how the different operation types (see Section 3.1.2 ff.) are handled.

Besides the stage register, the controller has a register for each VLO, which is globally evaluated, to store if that VLO has been started. This register is used to indicate whether the controller has to stall the pipeline until that VLO is finished. The register is controlled by the control-dependency of that VLO in the CDFG. At the same time, the VLO receives its parameters and the execution is started. The result is written into the output buffer of the VLO when it finishes its execution. The details for the VLO execution is described in Section 3.2.4.2 on the operation level.

All basic operations in a single stage are chained together. Only the last operation in a chain contains a register to store data until it is consumed by the successor stage.

MCOs can be a part of stage as either with their input, intermediate or output stage. The input stage is the stage where a MCO receives its parameters. The output stage is the stage where result of a MCO is available until consumed. All stages in between are the intermediate stages. For MCOs it is assumed that they can be halted, so that their internal pipeline is always kept in sync with the overall pipeline. If that assumption is false, they have to be handled as VLOs. Beyond this, MCOs need no additional logic to be supported.

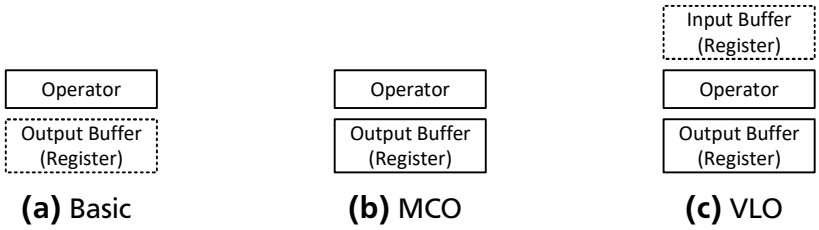


Figure 3.9: Operation type schemas

3.2.4 Operation level

The basic operation, as shown in figure 3.9a, consists of the operator with an optional output buffer. Operations are combinatorial if they are used without the output buffer.

3.2.4.1 Multi-Cycle Operation (MCO)

MCOs, as shown in Figure 3.9b, consist of the instruction with a mandatory output buffer. The operator is a pipeline that calculates the result in a constant number of cycles. When the operation is activated the input parameters are put into the pipeline, which immediately starts the calculation. When the pipeline signals completion, the result is written into the output buffer. The output buffer is a simple register. As this can hold only one value, it is obvious why the MCO pipeline must be able to be halted. If the application pipeline is halted by the static II model, and the MCO pipeline is not, then additional results would either be discarded or overwrite the previous result in the output register.

3.2.4.2 Variable Latency Operation (VLO)

VLOs, as shown in Figure 3.9c, consist of the operator with a mandatory output buffer. Optionally, they can have an input buffer for storing their parameters. The output buffer is mandatory because it is assumed that VLOs have such high delay that no combinatorial use is applicable. The input buffer is used for values that remain static during the execution of the VLO, like with nested loops where an input parameter is the termination condition that is checked each iteration. The number of input- or output buffers is determined by the actual instruction as they can have more than one parameter or, in the case of nested loops, multiple result values. All buffer of one type are kept synchronous. In the static II model, the buffers are always simple registers, similar to the basic operations.

When the operation is activated, the parameters are written into the input registers and the VLO's operator is executed. In case the VLO is a nested loop, the

nested graph is executed in the already described manner. When the VLO is finished the results are written to the output registers and the completion is signalled to the controller.

3.2.4.3 Nested Loop

Nested loops are executed as a variant of VLOs with unique operators. In fact, the pipeline of the nested loop is instanced inside the VLO. For that each input parameter is represented by an operation which implements the input buffer of VLOs. The controller of the nested loop is then signalled to start the calculation of the loop which is done with the shown execution model. The result values of the loop are represented by operations which implement the output buffer of VLOs. In the parent loop the nested loop is represented by just a VLO with its input parameters and output values connected to other operations in the parent loop.

3.2.4.4 Memory Access

Memory accesses are implemented as variant of VLOs with either unique or shared operators. But in most cases it can be assumed that memory accesses use shared operators as the number of memory ports is generally limited. More details to underlying sharing method is shown in Section 3.2.5. When the VLO representing a memory access is activated, the parameters of that access are either written into an input buffer or directly transmitted to the underlying memory system. As most memory systems have a limited number of access ports, the memory access has to go through an intermediary resource sharing layer (explained in the next Section 3.2.5). The memory system then checks its cache and if necessary accesses the shared memory. When the access is finally finished the result is written into the VLO's output buffer, waiting to be consumed by the pipeline.

Again, the details of the memory system's execution is described in the target system specifics in Chapter 7.

3.2.5 Resource Sharing

Representative for all resources that might be in limited availability, this section will use memory accesses to explain the resource sharing in the static II model. All memory operations have to be assigned to limited number of ports provided by the memory system. The static II model uses a technique that relies on the property that the static II guarantees a constant distance between all iterations. This means that it is always known which stages are active at the same time. Thus, all stages

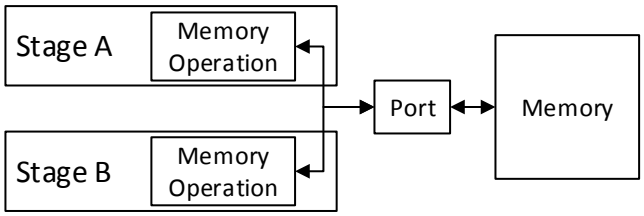


Figure 3.10: Simple port sharing in static II model

are grouped into sets of simultaneously active stages. All operations in one set are executed at the same time and cannot share their resources. On the other hand they can share their resources with operations in all other sets.

For the static model, this method is only used and developed for memory accesses which are then simply connected to their shared port of the memory system (Figure 3.10). This relies heavily on the actual implementation of the memory operators and the memory system. This work does not further refine the resource sharing in the static model, because the focus switched to the dynamic II model, presented in the next section. A more refined resource sharing system and its applications in the static model is an aspect of another work [LK16].

To assign the operations and resources, it is necessary to create an operation schedule which ensures that no limited resource is used more than it is available in a set of simultaneously active stages. Such a schedule is created by greedily moving a resource limited operation to the next stage until the conflict is resolved or using more sophisticated methods such as modulo scheduling. The scheduling algorithms are shown in Section 6.6.

3.3 Pipeline with dynamic II

The second model is also based on the pipeline model from Section 2.2.2 and is specified by a dynamic II. Depending on the actual of execution time in clock cycles of VLOs, this model can let iterations catch up to each other. This means that the II or the distance (number of stages, see Section 2.2.2) between two iterations can change during the execution. Some dependencies have to be resolved dynamically during the execution as they can not be guaranteed like in the static II model.

3.3.1 Example

In contrast to the static II model, this model allows for a dynamic II. For some of the advanced techniques described later, such a more sophisticated execution model was needed. Stalling the complete pipeline would render the improvements of these new techniques useless. But since stalls cannot be prevented, it was necessary to restrict their effect to only small parts of the pipeline.

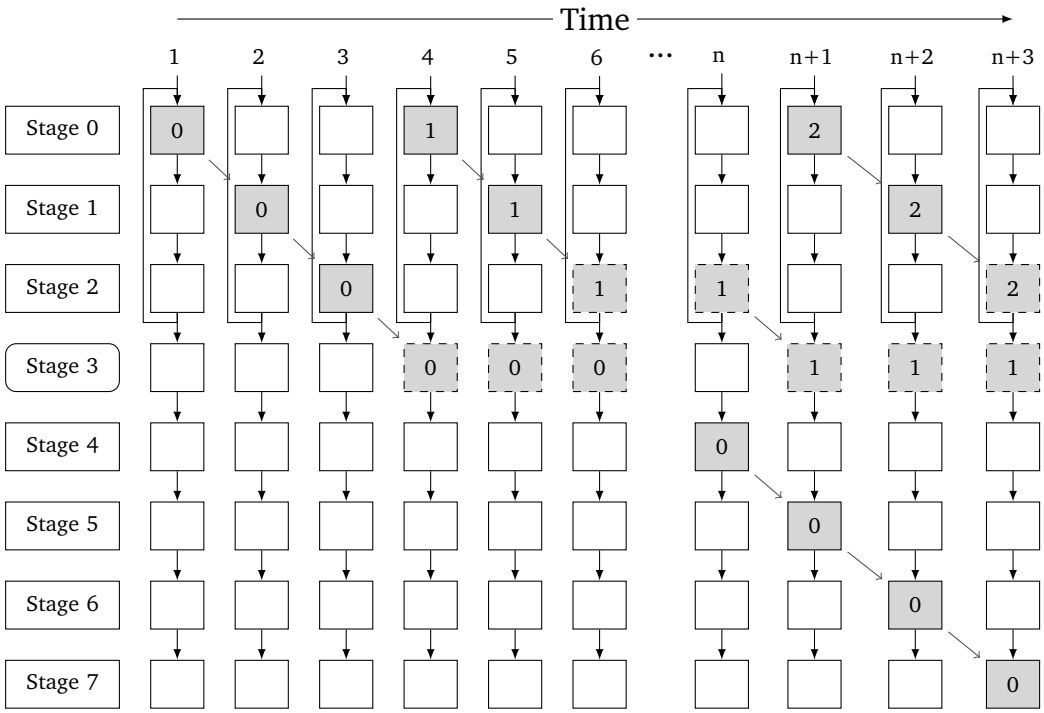


Figure 3.11: Dynamic Initiation Interval behaviour (II=3)
Iterations are shown as numbered boxes
Dashed iterations are stalled, rounded stages contain VLOs

In Figure 3.11, the behaviour of the dynamic II model can be seen. Up to time $t = 4$ it is exactly the same as the static II model. At $t = 5$ the first difference becomes visible. While Iteration 0 on Stage 3 is stalling, Iteration 1 can continue through the pipeline. At $t = 6$ Iteration 1 reaches the stage before Iteration 0. As all stage buffers can only hold the data of a single iteration, Iteration 1 now has to stall as well because it cannot be moved to Stage 3.

The memory access for Iteration 0 finishes at $t = n$, which allows Iteration 0 to continue. Iteration 1 cannot continue immediately, because the propagation of control signals inside the controller is registered. Instead, Iteration 1 can continue in the next time step $t = n + 1$. The hole in the pipeline that was created is called a bubble. In comparison to the static II model, Iteration 2 is started at $t = n + 1$ instead of $t = n + 2$. Iteration 1 stalls at Stage 3 because of the memory access, like in the static II model. But this stall now has no impact on Iteration 0 and no immediate impact on Iteration 1. Iteration 2 can continue until it reaches the stage before Iteration 1 at $t = n + 3$. At the same time Iteration 0 has already reached Stage 7, where in the static II model it only reaches Stage 7 after the memory access of Iteration 1 at Stage 3 has finished at $t = m$. Assuming $t = m$ is later than $t = n + 3$, the dynamic II model could gain $m - n + 3$ clock cycles.

3.3.2 Pipeline level

On the pipeline level of the dynamic II model, the handling of all VLOs and memory dependencies is done locally in stage transitions. Because the model does not guarantee a fixed II, it cannot rely on an appropriate fixed schedule to resolve the inter-iteration dependencies in the MDG. Instead the model uses additional tokens to resolve them.

Similar to the static II model, the pipeline of this model is split into controller and data-path. On the pipeline level, the behaviour is mostly defined by how and when the controller moves iterations between the pipeline stages. From the control-conditions evaluated in the data-path, only the conditions for VLOs and the end condition have an impact on the behaviour at the pipeline level.

The dynamic II model uses few tokens to indicate the position of data within the pipeline. Similar to the static II model, all data of a single iteration is generally represented by a single token. The exception to this are VLOs and inter-iteration dependencies, which need additional handling.

Each transition of an iteration (meaning the data associated with it) from one stage to the next is handled independently and simultaneously in the dynamic II model in contrast to the global decisions in the static II model.

The decision if a transition is executed depends on a number of factors. It is always checked that sufficient buffer space is available in target stage of each transition. Also, it has to be made sure that inter-iteration dependencies are not

violated and VLOs must be handled. Intra-iteration dependencies are resolved by the staged execution, similar to the static II model. The presence or absence of inter-iteration dependencies and VLOs creates a number of transition variants. The variants are presented after two detailed examples covering all elements used in the handling of stage transitions in the controller and data-path.

Before explaining the two detailed transition examples in Figure 3.12, all the elements are presented. The token for the staged execution is stored in the *stage token* buffer on each stage. Inter-iteration dependencies are handled by an additional *dependency token FIFO* and an optional *data FIFO*. The dependency token FIFO stores the tokens for inter-iteration dependencies. The data FIFO stores the data used by previous stages over the inter-iteration dependencies. Both the dependency token and data FIFO are always kept synchronous. For all dependencies from or to a specific stage, the token FIFO is only generated once. In the case the inter-iteration dependencies are memory dependencies only, the data FIFO is not used as the memory system is used to transfer the data. Note that in the figures, the FIFO are shown as coming from or going to an unspecific stage after or before the stages in the figure, respectively. The *local controller* combines all necessary informations on a stage and decides if an iteration can move to the next stage.

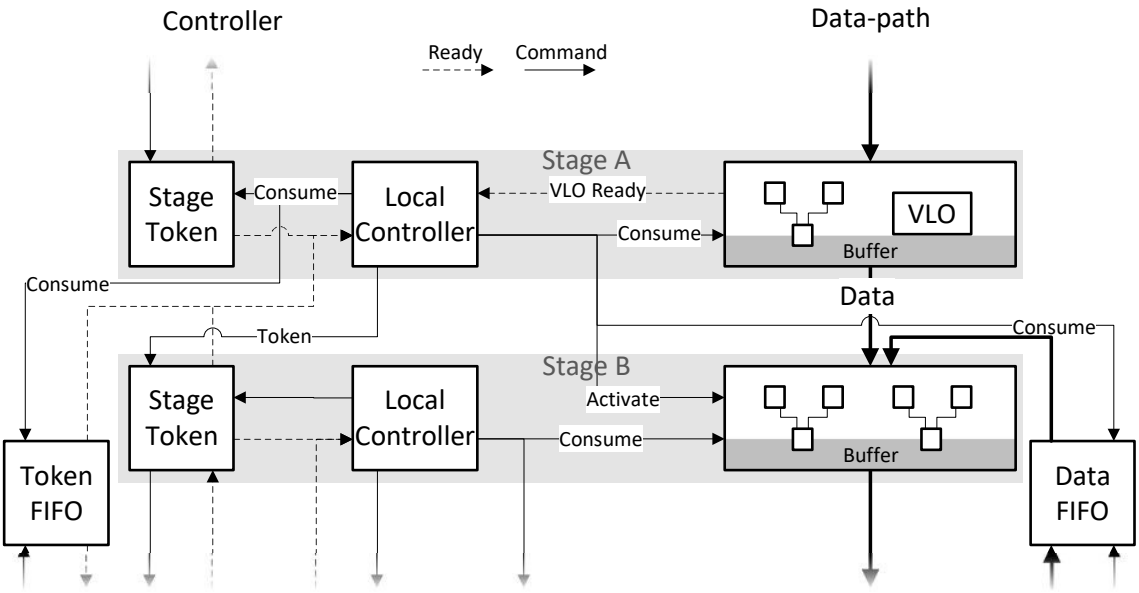
Both examples show a transition from a multi-threaded to a single-threaded stage with either an incoming (Figure 3.12a) or outgoing (Figure 3.12b) inter-iteration dependency. The examples differ in three points.

1.) The first, obvious difference is whether an incoming or outgoing inter-iteration dependency is involved in the transition. In Figure 3.12a, the incoming dependency comes from a stage *after Stage B* to Stage B. In Figure 3.12b, the outgoing dependency goes from Stage A to a stage *before Stage A*.

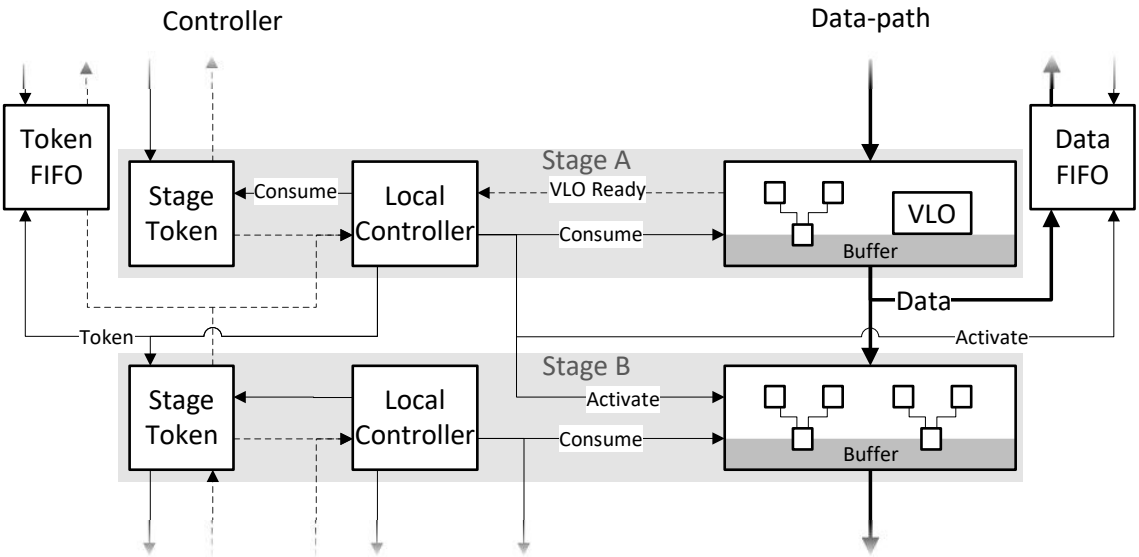
In both examples, Stage A contains at least one VLO, which needs additional handling beyond the basic operation handling with the stage token. The local controller checks if the stage token is *Ready* (the stage contains an iteration) and if the successor Stage B is *Ready* to accept an iteration. Since Stage A also contains VLOs, the local controller checks whether each VLO is either finished or was not executed (more details on that in the stage level description).

2.) The second difference is in what the local controller has to do with the inter-iteration dependency. For the incoming inter-iteration dependency to Stage B, the local controller has to check if the dependency token FIFO and thus the data FIFO has a token or data available, respectively. For the outgoing inter-iteration dependency to a previous stage, the local controller checks if the dependency token FIFO and thus the data FIFO is ready to accept a token and data.

If all conditions are satisfied, the local controller activates the successor Stage B and submits the stage token in every transition variant. When the stage is activated, all operations in that stage are executed and their results are written into



(a) Incoming inter-iteration dependency



(b) Outgoing inter-iteration dependency

Figure 3.12: Stage transitions (detailed overview)

the output buffer, waiting to be consumed by the next stage. At the same time, the data in Stage A is consumed to make space in its output buffer.

3.) The last difference is, whether tokens (and data) are consumed or new tokens (and data) are generated. In the case of an incoming dependency, the token and data from the dependency token and Data FIFOs are consumed. In the case of an outgoing dependency, a new token is written into the dependency token FIFO and the data moving over the dependency edge is written into the data FIFO.

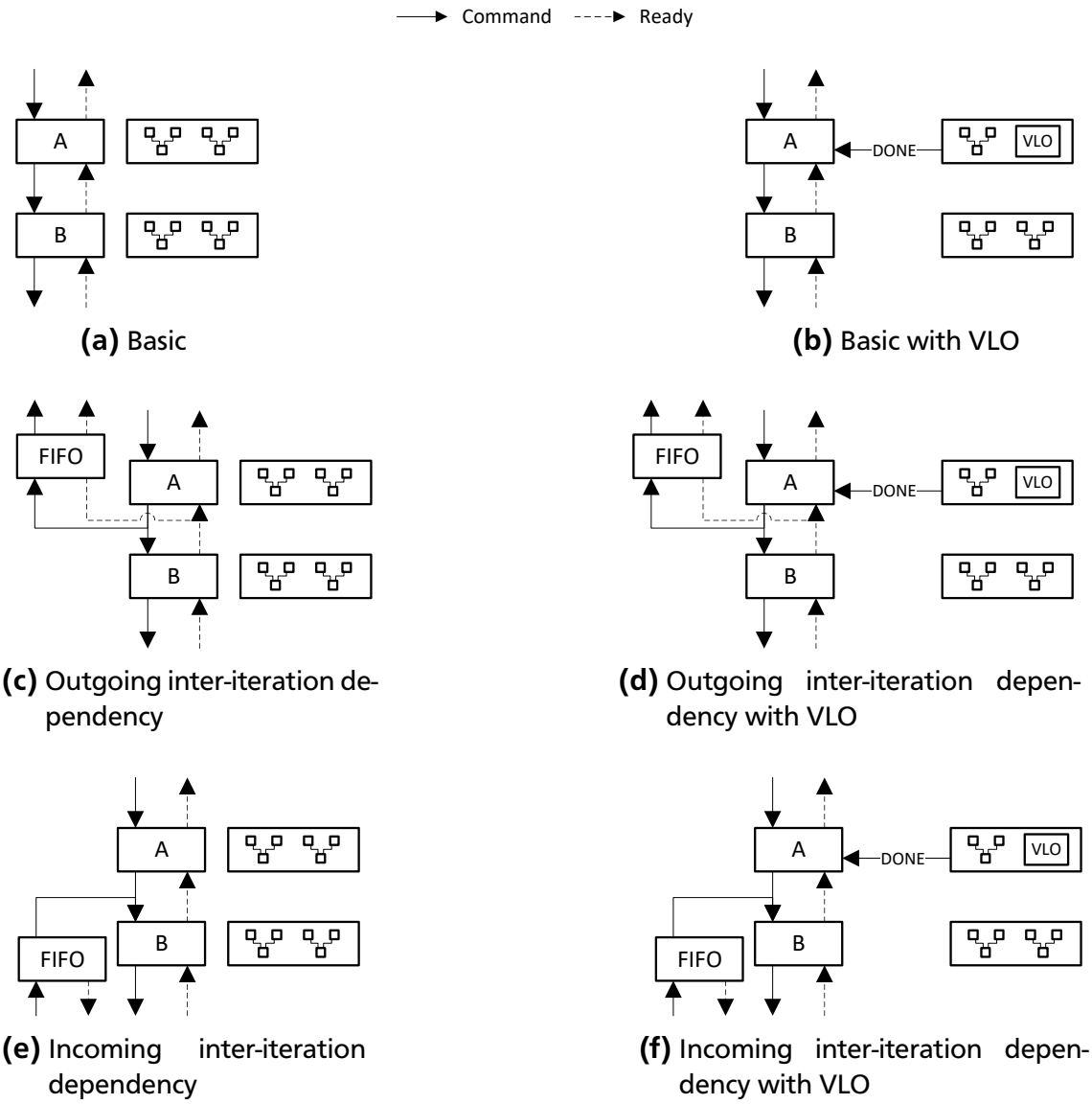
While both figures include all used mechanisms, the data and dependency token FIFOs are only needed in variants that contain inter-iteration dependencies, either data or memory. Memory dependencies are handled using only dependency token FIFOs, as the data is managed by the shared memory. Data dependencies require the additional data FIFOs to transfer the data over the inter-iteration dependency edge. Similar, VLO handling is only integrated when there are actual VLOs on the particular stage.

General models need to explicitly track memory dependencies, e.g., using special activation tokens [GL11]. However, at the stage-based granularity used in this work, explicit dependencies can often be omitted (relying on the staged execution order), or be folded onto other dependencies. Both ways reduce the number of dependencies that need to be explicitly tracked using dependency tokens. This folding is shown in Section 3.3.2.1.

This work classifies stage transitions by two parameters. The first parameter is whether inter-iteration dependencies are involved and if the stage is a source or target of that dependency. The second parameter is whether the source stage of the transition contains VLOs. This leads to the following six variants of stage transitions, which are shown in Figure 3.13. To simplify the figures for each variant, the difference between memory and data dependencies is omitted and always represented by just the dependency token FIFO.

In the left column, all variants without VLOs are shown while all variants with VLOs in Stage A are in the right column. (Note that VLOs in Stage B are irrelevant, as input queues guarantee they can always accept data when the stage can (see Section 3.3.4.2). The rows show from top to bottom the basic transitions without inter-iteration dependencies, transitions with outgoing inter-iteration dependency, and transitions with incoming inter-iteration dependency.

In all variants the local controller at stage A evaluates the *Ready* signal from the succeeding stage B. If stage B signals that it cannot process another iteration via not sending a *Ready*, stage A does not submit the current iteration. In the case that stage A contains at least one VLO, the local controller additionally checks that all VLOs in the stage are finished or were not executed at all. These basic variants are shown in Figures 3.13a and 3.13b.



(a) Basic

(b) Basic with VLO

(c) Outgoing inter-iteration dependency

(d) Outgoing inter-iteration dependency with VLO

(e) Incoming inter-iteration dependency

(f) Incoming inter-iteration dependency with VLO

Figure 3.13: Stage transitions (simplified, only control signals, data connections are omitted)

Figures 3.13c and 3.13d show the transitions where stage A has outgoing an inter-iteration dependency to a previous stage. Such an inter-iteration dependency occurs when data from the iteration in stage A is used by a successor iteration in an earlier stage. In addition to the basic transition rules, the iteration moves from stage A to stage B only when both the FIFO and stage B are ready to accept the iteration. The iteration then simultaneously moves to the FIFO and stage B.

Figures 3.13e and 3.13f show the transition where stage B has an incoming inter-iteration dependency from a later stage. Such an inter-iteration dependency occurs when the iteration coming from stage A requires data from the previous iteration in stage B. In addition to the basic transition rules, the iteration moves from stage A to stage B only when the FIFO has data-available and stage B is ready to accept an iteration. This combines the previous iteration's data in the FIFO, with the new iteration coming from stage A. For the first iteration, the FIFO is initialized with dummy data which is discarded in the stage logic.

3.3.2.1 Dependency Folding

```
x = 0;
while (x < 5) {
  tmp = *b;
  *a = tmp + 1;
  x = tmp / 2;
  *c = *d + 1;
}
```

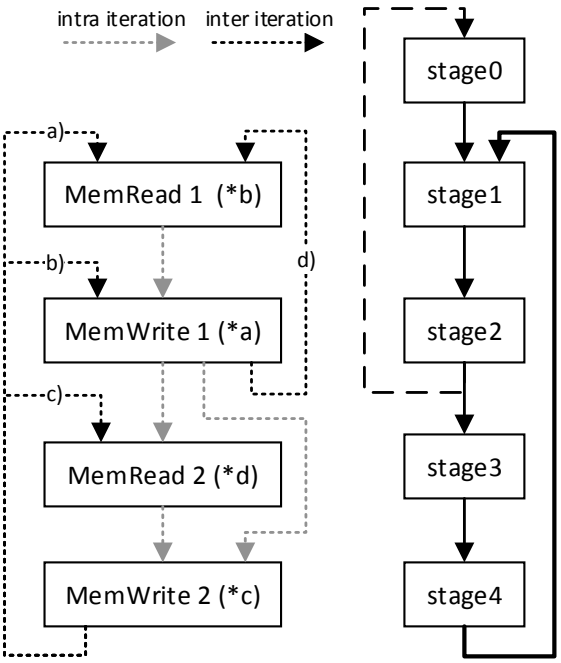


Figure 3.14: Handling memory dependencies (II=3)

At the stage-based granularity used in this work, explicit dependencies can often be omitted (relying on the staged execution order), or be folded onto other dependencies. This reduces the number of dependencies that need to be explicitly tracked using dependency tokens.

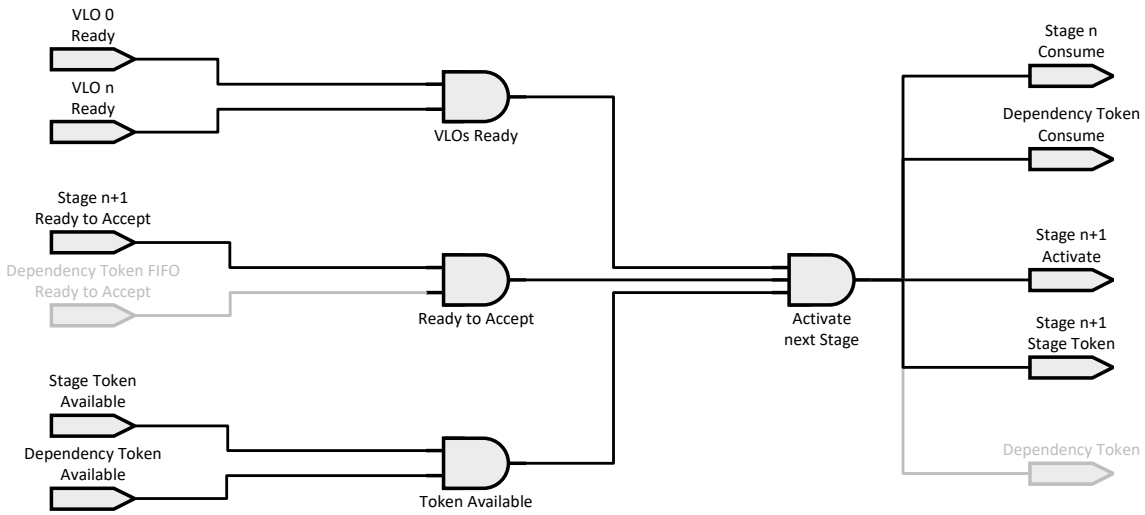
Figure 3.14 illustrates this on a simple example with II=3 using memory dependencies as representatives for all dependencies. For the code fragment given on the left side, the potential memory dependencies shown in the center exist (distinguishing between intra- and inter-iteration dependencies). Intra-iteration dependencies do not need to be tracked explicitly, as assigning each memory operation to a stage (Stage 1 . . . 4) in program order ensures that these dependencies will be satisfied. On the other hand, without explicit tracking, the dynamic model cannot ensure that MemWrite2 is always executed *prior* to MemRead1 (Inter-iteration dependency *a*), MemWrite1 (Inter-iteration dependency *b*), and MemRead2 (Inter-iteration dependency *c*). Unless it can be proven (e.g., using alias analysis), that these potential dependencies do not actually exist, this execution order must be guaranteed using explicit tracking in the dynamic model.

In a purely operator-based dynamic execution model, MemRead1 would wait for two individual tokens from its inter-iteration predecessors MemWrite1 and MemWrite2, requiring logic and wiring area on the device. However, the tracking effort can be reduced by exploiting the stage-based execution, leading to a simplified controller as shown at the right side of Figure 3.14: Dependency *(d)*, execute MemWrite1 prior to MemRead1 of the next iteration, is *already* enforced by the inter-iteration data-dependency (Stage 2, Stage 0) that ensures the correct update of the loop decision variable x (which is the root cause for II=3). Memory edges *(b)* and *(c)* can be folded onto memory edge *(a)*, which, together with the natural stage order present in all presented models, ensures that MemWrite1 and MemRead2 will be executed after MemWrite2. Thus, in this example, only that last memory dependency needs to be explicitly tracked, leading to a dependency token FIFO (from Stage 4 to Stage 1) in the controller.

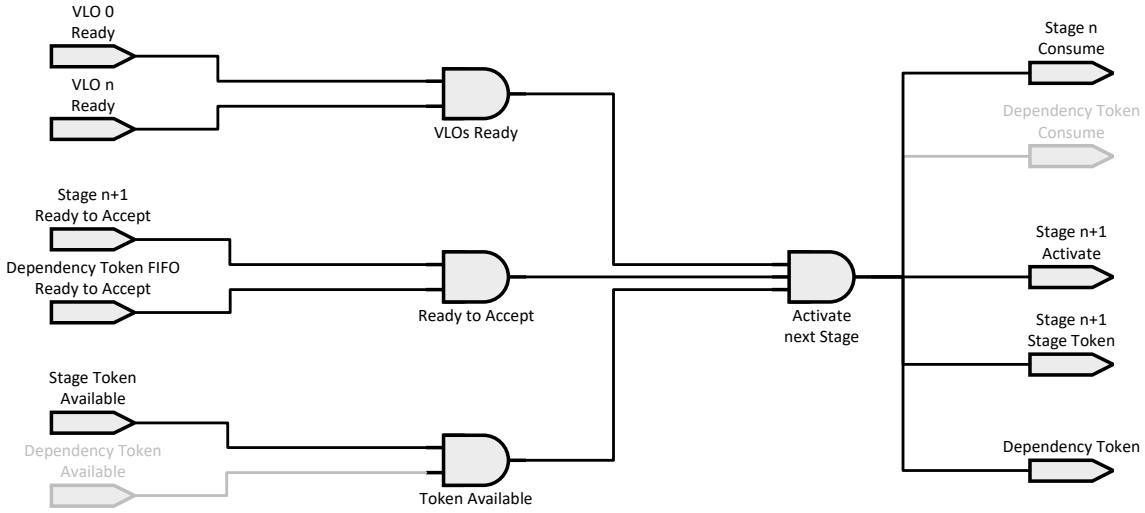
3.3.3 Stage level

In the controller part of the stage level, the logic inside the local controller in each stage will be shown. Similar to the pipeline level, the details of the logic are shown for incoming (Figure 3.15a) and outgoing inter-iteration dependencies (Figure 3.15b). In both figures, elements which are not used in that case are show in light grey.

In both figures, the left side shows the different ready or available status signals for the local controller which were shown as dashed lines in Figures 3.12a and 3.12b. On the right side, the control signals generated by the local controller are shown which were solid lines in Figures 3.12a and 3.12b. The local controller uses a two stage combinational logic. On the first stage, it is checked if all VLOs have finished their computation, all token buffers are ready to accept a new token and all necessary tokens are available. If all this is given, the second stage sends



(a) Incoming inter-iteration dependency



(b) Outgoing inter-iteration dependency

Figure 3.15: Local Controller logic (Unused elements are shown in light grey)

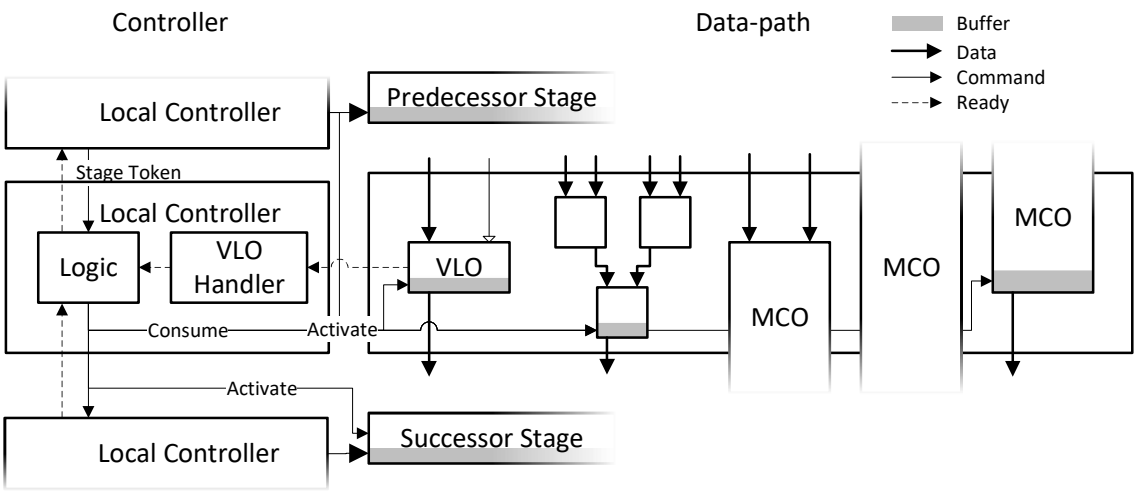


Figure 3.16: Controller excerpt and pipeline stage with chained basic operations, a VLO, input, intermediate and output stage of MCOs. Shaded parts of operations contain a register to store data until it is consumed by the successor stage

the activation and consumption control signals. Also, the tokens are sent to their target buffers.

The local controller contains a VLO handler for each VLO. These handlers receive the information which VLOs are executed or finished from the data-path. When a stage is activated and the control-dependency of a VLO is fulfilled, the handler remembers that the iteration can move into the successor stage only when the VLO signals its completion. At the same time, the VLO receives its parameters and the execution is started. The result is written into the output buffer of the VLO when it finishes its execution. The details for the VLO execution is described in Section 3.3.4.2 on the operation level.

Figure 3.16 shows a single stage of dynamic II model. Equal to the static II model, all basic operations in a single stage are combined into combinational chains. When the stage is activated, all output buffers are activated to store the combinational operation chains' result at the last operation of each chain. The difference to the static II model is, that the buffers are not always a simple register but can also be a queue to hold more than a single value until it is consumed by the successor stage. The idea behind this assumes that after a cache miss, often multiple hits occur. Queues allow iterations to get closer to memory access, so that they can be executed earlier than if they could not get as close to the memory access stage without using queues.

The configuration of this buffer as either register or queue is currently decided by compilation parameters. The options are evaluated in Section 9.2 among others. All buffers on a stage are configured to use the same buffer type and size.

Also equal to the static II model, MCOs can be a part of a stage with either their input, intermediate or, output stage. While the static II model required that MCOs can be halted, the dynamic model has to use a different approach to keep MCOs in sync with the overall pipeline. The activation of the output buffer of each MCO does not directly correspond with the stage activation, but with the time when a value moved through the whole MCO pipeline. This approach requires the use of queues and is described in Section 3.3.4.1 on the operation level.

3.3.4 Operation level

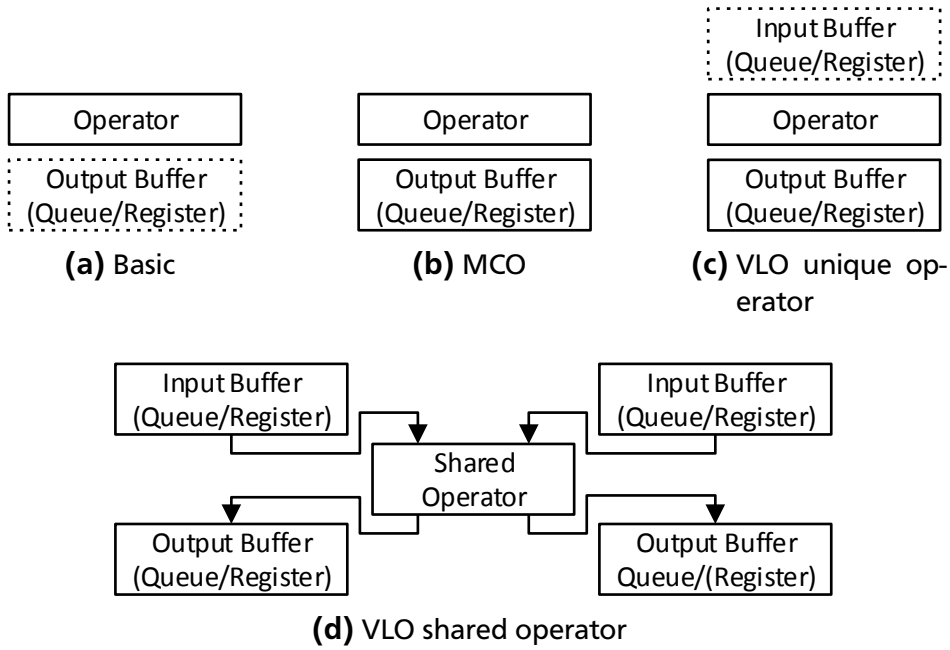


Figure 3.17: Operation type schemas

As written in Section 3.3.3, in the dynamic II model, the buffers of all operations can be either a simple register or a queue (Figure 3.17), instead of only a register in the static II model. While the basic operations have further changes, the handling of MCOs and VLOs is changed however. These changes are described in the following sections.

3.3.4.1 Multi-Cycle Operation (MCO)

The stage-based dynamic II model is based on the assumption that data can be stalled on each individual stage. MCOs, especially ones generated through third party tools such as XILINX CoreGen [Xil00], often cannot be stalled at their individual internal pipeline stages. While this was fine in the static II model, in the

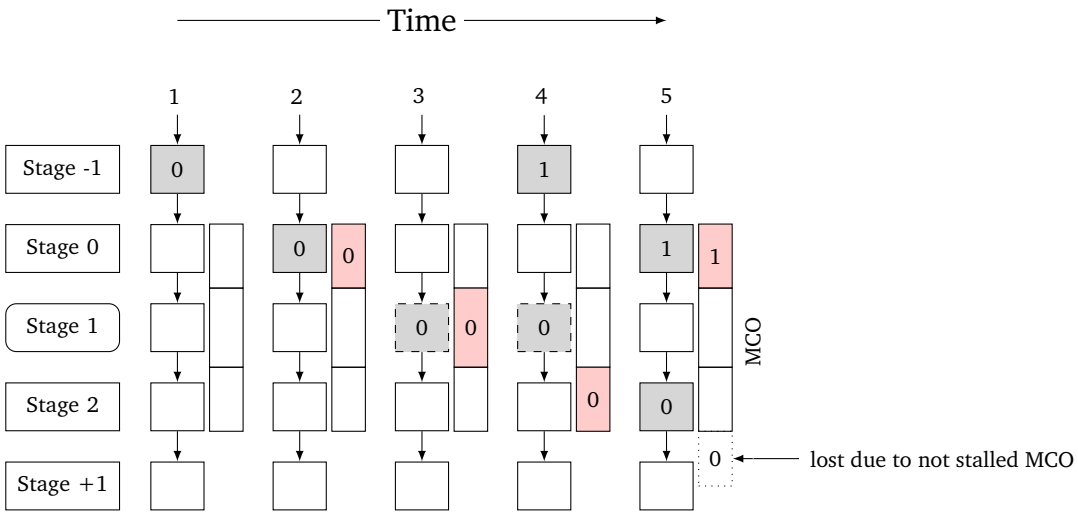


Figure 3.18: De-synchronization of pipeline if MCO cannot be stalled
(Black: Data-Path, Red: Multi-Cycle Operation)
Iterations are shown as numbered boxes
Dashed iterations are stalled, rounded stages contain VLOs

dynamic II model this can lead to errors. Figure 3.18 and 3.19 show examples for these errors. Both figures show a pipeline that contains a multi-cycle operation that spans multiple stages. In Figure 3.18 and 3.19 these are three and four stages respectively. Stage -1 and +1 denote the stages before and after the multi-cycle operation.

In Figure 3.18 the MCO is never stalled. At time $t = 3$ the pipeline is stalled because of a cache miss, but the data in the MCO continues to the next stages until it becomes lost at $t = 5$ due to leaving the MCO pipeline.

On the other hand in Figure 3.19, the MCO is stalled as soon as the pipeline is stalled on one of the stages which contains the multi-cycle operation. At the start the pipeline already contains the two iterations 0 and 1. Iteration 0 has already entered the MCO and is at Stage 1. The pipeline runs without interruption until $t = 3$, where Iteration 1 stalls at Stage 1. Because the pipeline stages are stalled individually, the other iterations can continue. Thus at $t = 4$ Iteration 0 leaves the part of the pipeline which contains the MCO. But as the MCO cannot be stalled individually but is stalled completely, the result for iteration 0 hangs in the MCO pipeline. This results in a desynchronization of the data and leads to failure of the complete application. Ignoring this, at $t = 5$ another failure case is shown. As for the stall of Iteration 1, the multi-cycle operation is still stalled, the data for Iteration 2 cannot enter the MCO and is lost.

As it can be seen from both examples neither stalling nor letting the operation continue is an option. But it is also not possible to stall all pipeline along a MCO

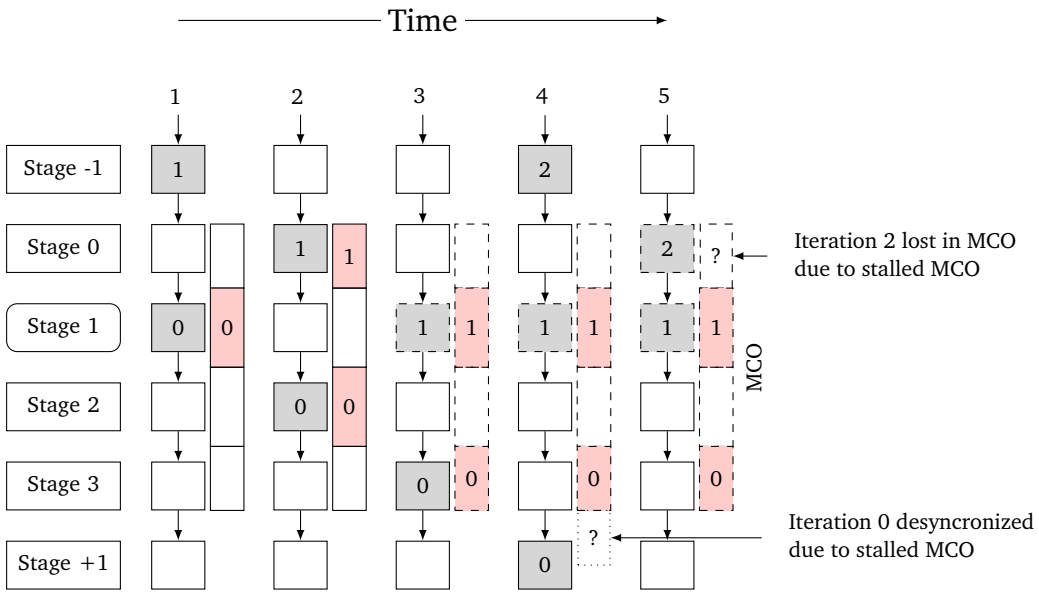


Figure 3.19: De-synchronization of pipeline if MCO cannot be individually stalled
(Black: Data-Path, Red: Multi-Cycle Operation)
Iterations are shown as numbered boxes
Dashed iterations are stalled, rounded stages contain VLOs

as that can lead to deadlocks if the MCO spans over both source and target of a backedge. To re-synchronize and store the results a queue is added to the end of the operation. This queue stores all the results until they are consumed by the successor stage. The queue has to hold all possible data which could leave the operation until a stall has propagated backwards through all the single-cycle stages until the stage directly before the multi-cycle operation.

Figure 3.20 shows the two general cases whether a queue is needed or not at the end of a MCO. This solely depends on whether the MCO's output stage is before the earliest source stage of any inter-iteration dependency. In Figure 3.20a, the last stage of the MCO is on the same stage as the source of the inter-iteration dependency (Stage 2). Even when the VLO on Stage 1 stalls, the MCO requires no additional queue (assuming the MCO can be stalled).

In Figure 3.20b, however, the last stage of the MCO (Stage 2) is after the source of the inter-iteration dependency (Stage 1). If the MCO would be simply stalled it would lead to the failure case shown in Figure 3.19 at $t = 5$, resulting in the loss of the data of Iteration 1. Without the stall, it would lead to the failure case shown in Figure 3.18. By adding a queue as shown in Figure 3.20b, both failure cases are prevented. The size of this queue is the number of stages from the MCO which are after the source of inter-iteration dependency (here a single stage). Additionally,

if some of the intermediate stages are configured to contain a queue, the queue size of the MCO is increased by the accumulated size of these stage queues.

3.3.4.2 Variable Latency Operation (VLO)

In the dynamic II model, resource sharing is directly integrated and was less an afterthought than in static II model. Because of that, the definition of VLOs was extended to include resource sharing. For resource sharing, VLOs now use a *shared* operator. Without resource sharing, they use a *unique* operator instance.

In the case of shared operator instance (Figure 3.17d), at each point the instance is used, a stub is placed for the operation, containing only input and output buffers. The stub is connected to the shared operator through an arbiter. The arbiter controls the access to the shared resource. The details for that are explained in Section 3.3.5.

If it is a unique operator instance (Figure 3.17c), the VLO is similar in all details but the buffers to the static II model. Like all other buffers in the dynamic II model, they can be either queues or simple register. As it is unknown at compile time whether values are still in the input buffer or are waiting for consumption in the output buffer, both buffers have to be the same size. This size is equal to buffer size of the stage token FIFO or basic operation buffers. The execution of unique instances then is equal to static II model.

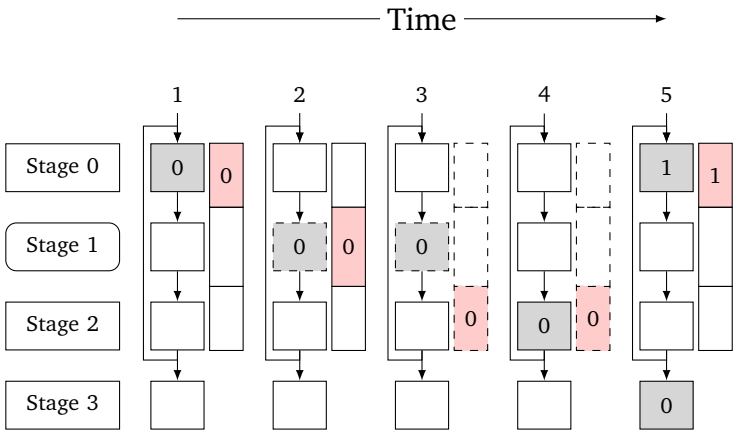
3.3.4.3 Memory Access

The general idea behind executing memory accesses does not change compared to the static II model (see Section 3.2.4.4). But resource sharing method is quite different here and is explained in Section 3.3.5.

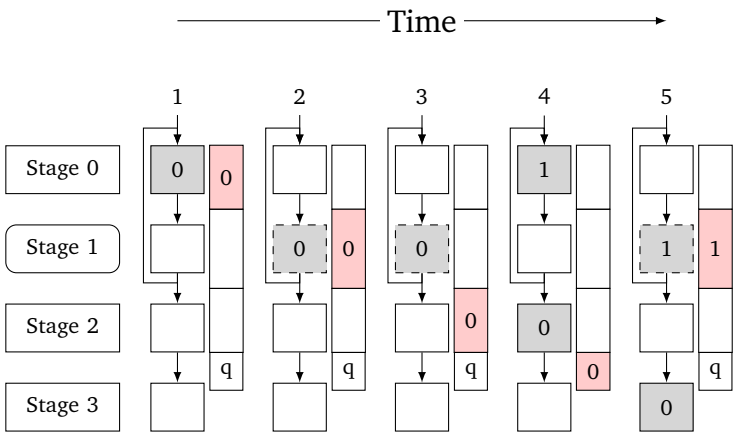
3.3.5 Resource Sharing

Similar to the static II model, memory accesses are used as stand-ins for all resources that might have only limited availability. Because the dynamic II model cannot guarantee a constant distance between all iterations, it cannot use a resource sharing method like the static II model.

In Section 3.3.4.2 it was described that VLOs with a shared operator are implemented by stubs connected to the actual operator by an arbiter. Figure 3.21 shows how an operator is shared in this manner between two operations. On the stage, each operation is represented by a stub containing only input and output buffers. The arbiter selects which stub can access the operator at a given. It does this by controlling a multiplexer, demultiplexer, and the operator. For presentation



(a) MCO encompassed by inter-iteration dependency, no queue required



(b) MCO ends after inter-iteration dependency source, queue required (empty queue marked by "q")

Figure 3.20: Queue usage for MCOs

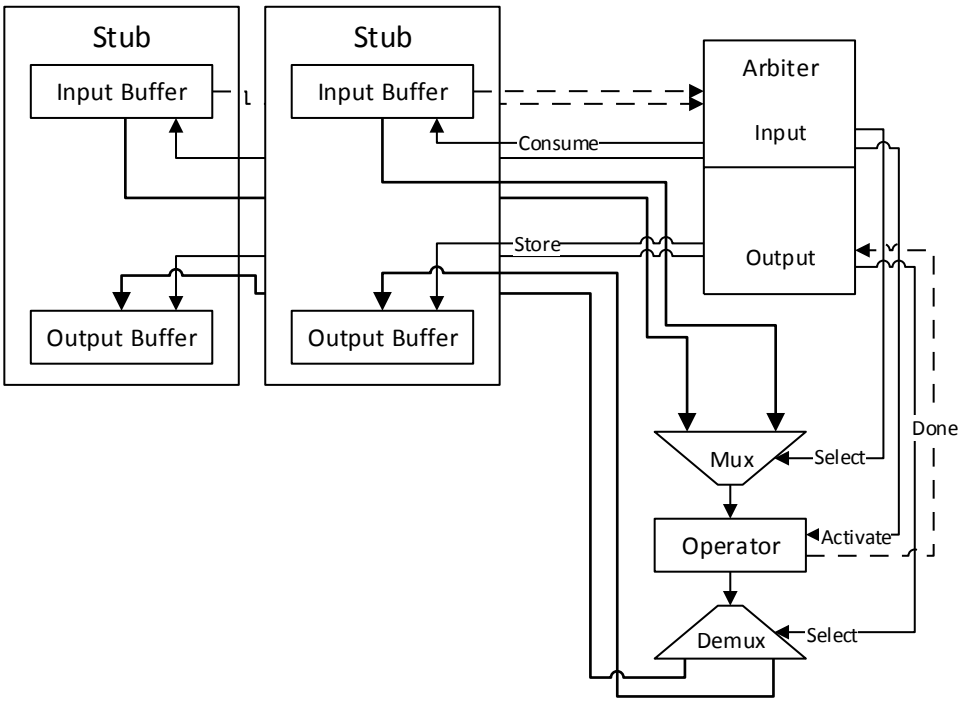


Figure 3.21: Shared operator between stubs

purposes the arbiter is split into input and output parts, which specify the signals uses in reading from the input buffer and writing into the output buffer of the stubs. For the explanation it is assumed that the shared operator is currently idle and both stubs have parameters in their input buffers and thus want to access the operator simultaneously. The arbiter receives an *available* signal from both stubs' input buffers. It then selects one of the stubs that is allowed to access the actual operator. The arbiter uses the input multiplexer to transfer the parameters from the selected stub's input buffer and activates the operator. This transfers the input parameters from the buffer to the operators and thus they are consumed from the selected stub. Now the arbiter waits until the operators signals their completion. Then the arbiters controls the output demultiplexer to write the result of the operator into the output buffer of the previously selected stub. In turn this signals the controller in the pipeline that the VLO is finished and has data available to be consumed. Also, the arbiter can now select the next stub's parameters.

The arbiter is unaware of the dependencies modelled in the MDG. The construction of the CDFGs and the controller assures that only instructions whose dependencies are fulfilled are reaching the arbiter.

The input buffer of each stub is generally configured to contain an output register. This means that it takes an additional clock cycle for the operator to be started after data was written to the input buffer. Most operators which are multiplexed have their own output registered. Because of that, the compiler allows a combi-

natorial short-cut through the output buffer in case the buffer is empty. However, this only done when few (less than 16) stubs share the same operator. If more than 128 stubs are connected to a single operator, the arbiter receives an addition output register to reduce the critical path before the operator.

3.3.6 Alterations to Hardware Execution Model

This section shows how the dynamic model can be adapted to include parts of other execution models like basic blocks or (hardware) functions. In Chapter 5 a method to emulate the basic block with the building blocks from the pipelined model will be shown.

3.3.6.1 Basic Block Extraction

In Section 2.2.3 it was shown that applications with extremely unbalanced basic blocks results in unbalanced pipelines. This in turn leads to inefficient pipelines compared to the basic block model. This section shows a method how the impact of such basic blocks can be reduced.

The basic idea is that individual BBs are directly integrated into the CDFG, but are represented by a nested (single iteration) loop in the pipeline model. As all nested loops are represented by VLOs (scheduled with a latency of one cycle), it is possible to improve the schedule by replacing the operations in seldom executed BBs with a short (schedule wise) VLO. While this introduces a certain amount of overhead (two clock cycles for extra input and output in nested loop), it can significantly reduce the II of loops. However, currently the minimum execution time for any (extracted) loop is three to five clock cycles (depending on the execution model), because the execution models are not yet optimised for very short loops.

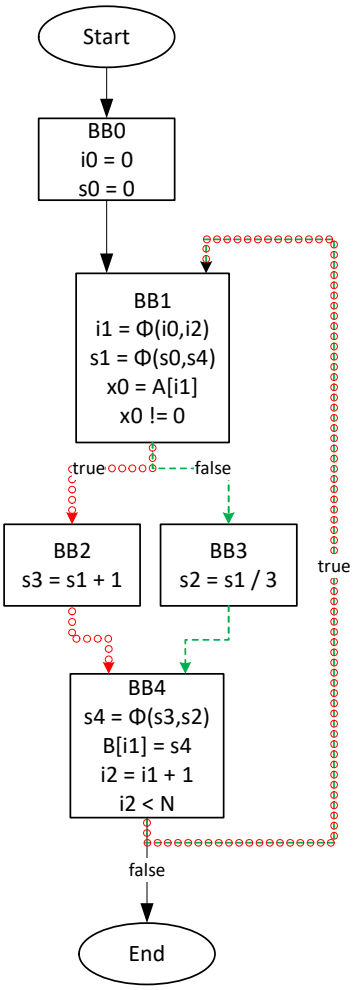
In the previous example favouring the basic block FSM model (Figure 2.15 in Section 2.2.3.1), which is shown again in Figure 3.22, two possible paths (marked as a (green) and b (red)) through each loop iteration are possible. While path b contains only short operations with one cycle latency, path a contains a single division operation with a latency of ten. By applying the basic block extraction on BB3 which contains the division, the II can be reduced significantly from 12 to 5 in the main loop (compare Figures 3.22c and 3.22d). The rest of the BBs were still constructed into a single CDFG.

3.3.6.2 Hardware Functions

To reduce resource requirements of a data-path the re-use of operations or their operators (as shown for memory accesses in Section 3.3.5) is an often-used technique. Now instead of re-using single operations, here we try to re-use entire parts

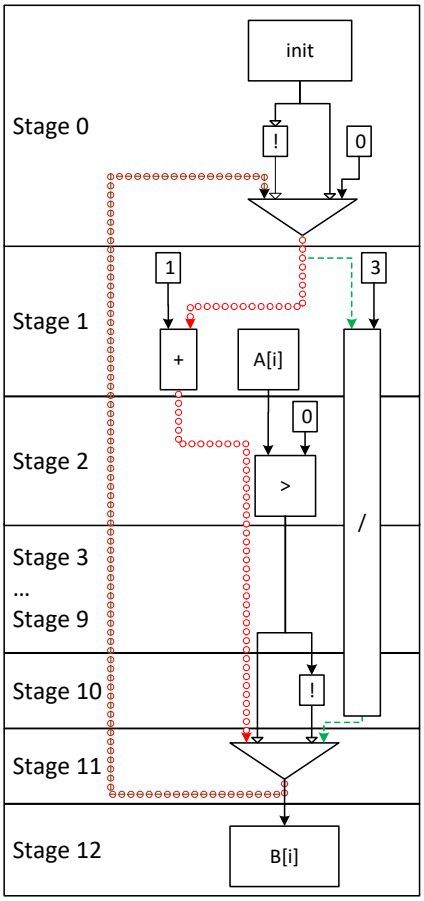
```
int A[N], B[N];
s = 0;
for(i=0;i<N;i++) {
  x = A[i];
  if(x != 0)
    s = s + 1; // fast
  else
    s = s / 3; // slow
  B[i] = s;
}
```

(a) Code

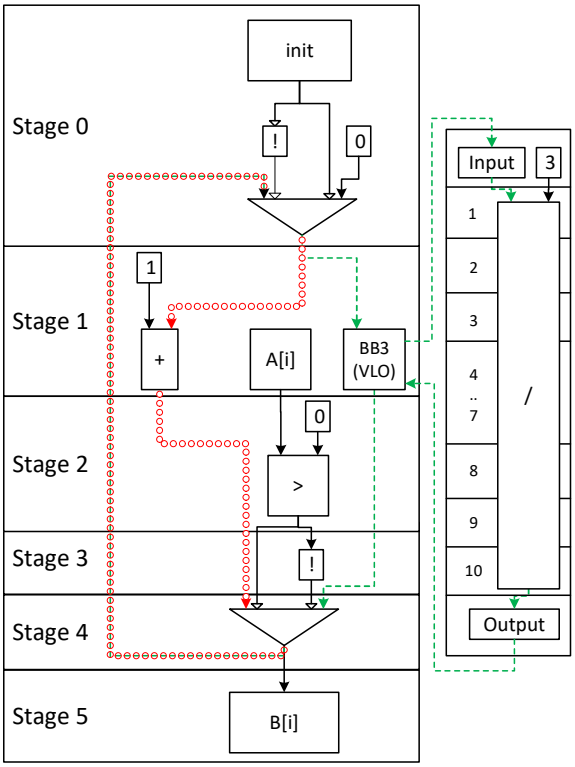


(b) SSA CFG

Figure 3.22: Unbalanced branches (continued on next page)



(c) CDFG without BB extraction (II==12)



(d) BB3 extracted and integrated as nested loop (II=5)

Figure 3.22: Unbalanced branches (continued from previous page)

of an application similar to how it is done in software. In software, entire parts of a program are often re-used by putting them into a *function* which can be used by *calling* it from different points in the program with different parameters.

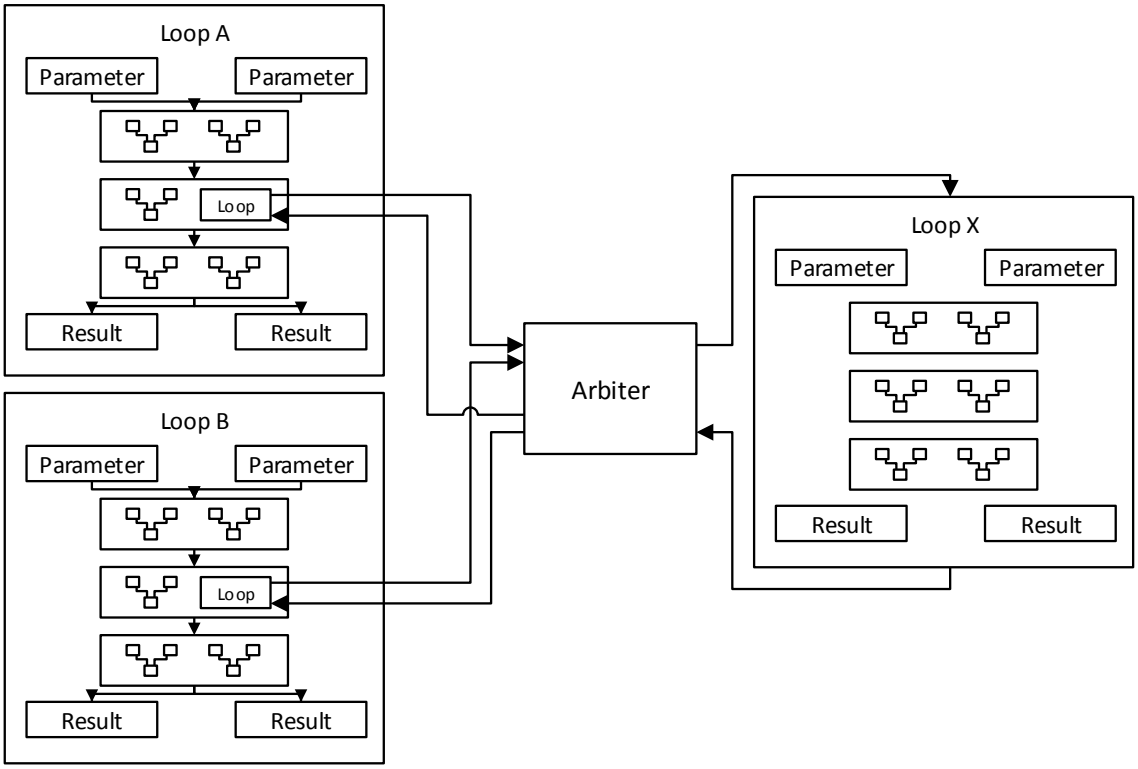
The idea here is to basically do the same in hardware. To this end, instead of inlining all functions, some functions are designated as *hardware functions*. For each of these hardware functions, a separate data-path is created which is integrated into the CDFG like a normal nested loop. But instead of directly executing the nested loop, the multiplexing method for shared operators (Section 3.3.5) is re-purposed to control the access to these hardware functions.

Figure 3.23 shows how hardware function X is called from within two different functions, implemented by the loops A and B. The same arbiter used for shared operators now selects between which loop VLO should be allowed to access the loop implementing function X. Because of this arbiter-based approach, hardware functions are only possible without recursive (indirect or direct) function calls. However, this is no new additional restriction as the overall execution model never supported recursive functions anyway.

Each hardware function is implemented as the operator in resource sharing method presented in Section 3.3.5, thus having the same delay (at least one addition cycle) until the operator is activated.

```
...
for (...) { // Loop A
...
x = X(a,b);
...
}
...
for (...) { // Loop B
...
y = X(c,d);
...
}
```

(a) Code



(b) Nested loop using arbiter based resource sharing

Figure 3.23: Hardware function: sharing a nested loop's pipeline

4 Multi-threading

This chapter describes the multi-threading method used to improve the efficiency of the generated pipelines. This will be done by increasing the throughput similar to creating multiple copies of the data-path, but without the linear area overhead incurred by creating copies. Instead, this work increases the utilization of a single pipeline by interleaving the iterations of multiple threads on the same hardware.

Examples for this interleaving will be shown based on the traditional C-slow [LRS83; LS91] method which is then adapted to support applications with VLOs like cached memory accesses, nested loops or other instructions whose execution time is data dependent.

Afterwards, the multi-threaded execution model is presented analogously to the previous models' explanations.

4.1 Example

To explain the idea behind the multi-threaded execution model, two examples will be shown. The first example will show the traditional C-slow method with a static interleaving of threads. The second example will the show the improved model which can dynamically reorder threads. Both examples are using a pipeline with $II = 3$.

4.1.1 Static Interleaving

One of the simplest form of multi-threading is to process N independent data streams in the data-path, where $N = II$ to improve throughput. The data streams would be externally interleaved/deinterleaved on a fixed round-robin (in-order) basis, using the principles of C-slow execution (with $C = N$) introduced by Leiserson et al. However, the original approach is limited in that in applies only to constant-latency operators.

In Figure 4.1a at $t = 1$, $t = 2$ and $t = 3$, Thread 1 (grey), 2 (red) and 3 (red) respectively enter the pipeline. With these three threads, the pipeline is now completely filled because at $t = 4$ the next iteration of the first thread (number in grey box is 1) is started. As the pipeline has no VLOs, the execution is never halted and continues until completion.

In Figure 4.1b a VLO is added to pipeline in Stage 3. Similar to the static II model, the whole pipeline is stalled when the VLO is executed. Beginning with

the situation from the previous example, at $t = 4$ the pipeline has to be halted while the VLO on Stage 3 is executed until $t = n$. For Thread 2 (red) and Thread 3 (green), the VLO does not stall (e.g. cache hit during a memory access). A second stall of Thread 1 occurs at $t = n + 2$ which lasts until $t = m$.

Assuming that these global stalls through memory accesses generally take longer than one clock cycle, it becomes quite obvious that the global stalling is not a good approach for multi-threaded pipelines with VLOs, as a single slow thread limits the execution speed for all other threads.

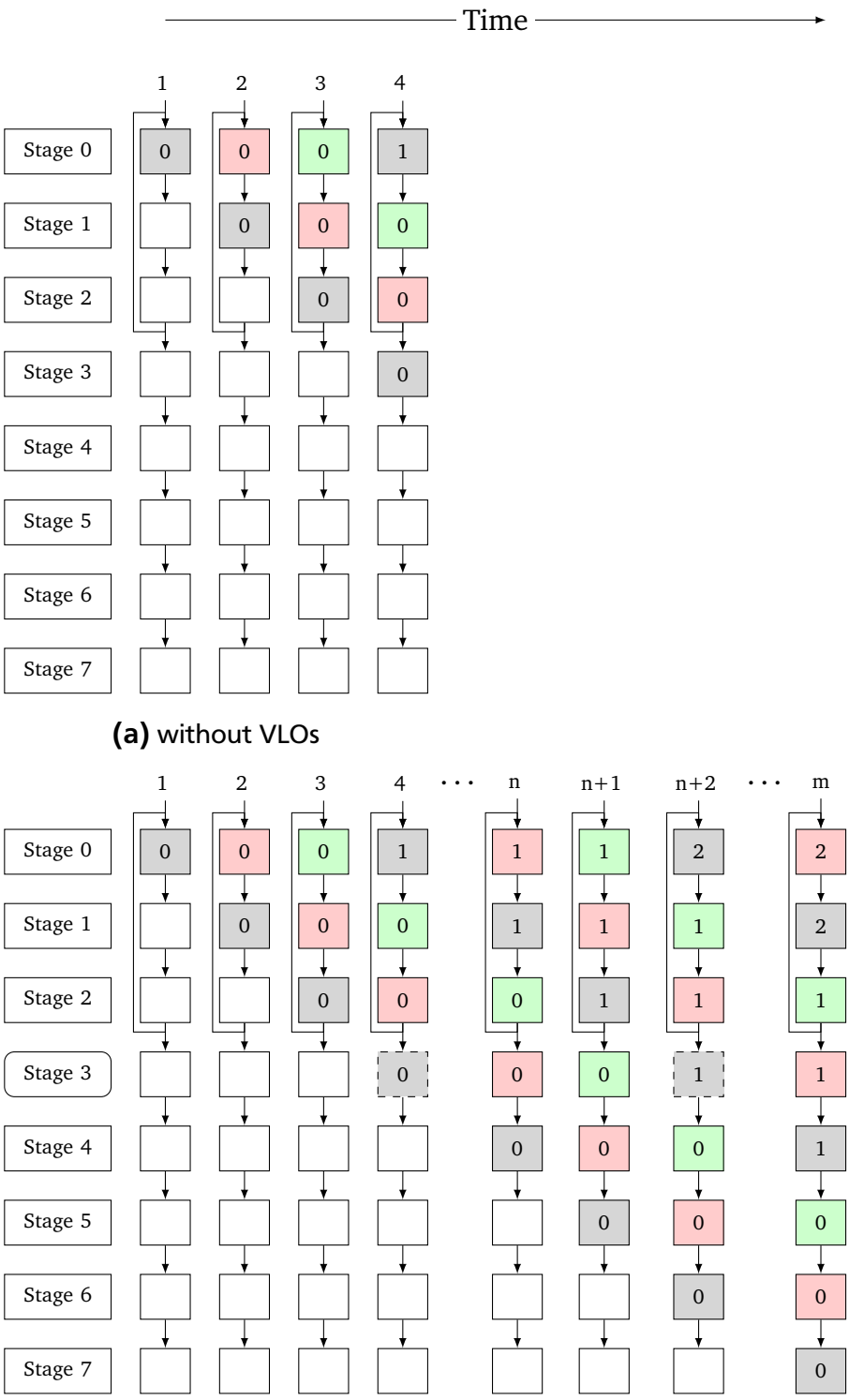
4.1.2 Dynamic Reordering

To reduce the impact a stalled thread has on other threads, the interleaved model was improved by allowing to *change* the order of threads. This reordering is used to permit threads to overtake a stalled thread.

In Figure 4.2, the interleaved execution of two threads with dynamic reordering is shown. Stage 3 contains a memory access as the sole VLO of the pipeline. This stage and Stage 0 are both stages on which threads can be reordered. Why Stage 0 can also reorder threads will be explained later in the execution model. But note that this is necessary for the correct behaviour of the execution model.

The example starts at $t = 3$ with Iteration 0 and 1 of Thread 1 (grey) and Iteration 0 of Thread 2 (red). At this point T_1I_0 (Iteration 0 of Thread 1) is stalled at Stage 3. At $t = 4$, T_2I_0 enters Stage 3, also stalling, and T_1I_1 is started. At $t = 6$, both T_1I_1 and T_2I_1 moved up to the stalls at Stage 3. From this point, all iterations in the pipeline are stalled until $t = n$. No new iteration for Thread 1 is generated in Stage 0, because Iteration 1 has to move simultaneously to Stage 3.

At $t = n$, T_2I_0 finishes its memory access, faster than the earlier started access of T_1I_0. The reordering in Stage 3 allows Thread 2 to overtake Thread 1, moving to Stage 4. Note that T_1I_0 is stalled until $t = n + 2$. In the pipeline configuration of the example, this leads to the situation where T_2I_1 is stalled because it cannot reach the reordering stage at Stage 3. The compiler can use additional features to prevent this which will be shown in the detailed model description. At $t = n + 3$, T_1I_1 enters Stage 3 and the memory access again stalls. When at $t = n + 5$, T_2I_1 reaches the memory access at Stage 3, it immediately finishes the access (e.g. due to a cache hit). This leads to Thread 2 again overtaking Thread 1. At $t = n + 7$ the same situation as at $t = n$ occurs, where Thread 2 is stalled because it cannot reach the reordering stage at Stage 3. At $t = n + 8$, the stalled access finishes and the pipeline continues until completion.



(a) without VLOs

(b) with VLOs (entire pipeline stalls)

Figure 4.1: C-slow multithreading behavior with $II = 3$ and three interleaved threads. Iterations are shown as numbered boxes
Dashed iterations are stalled, rounded stages contain VLOs

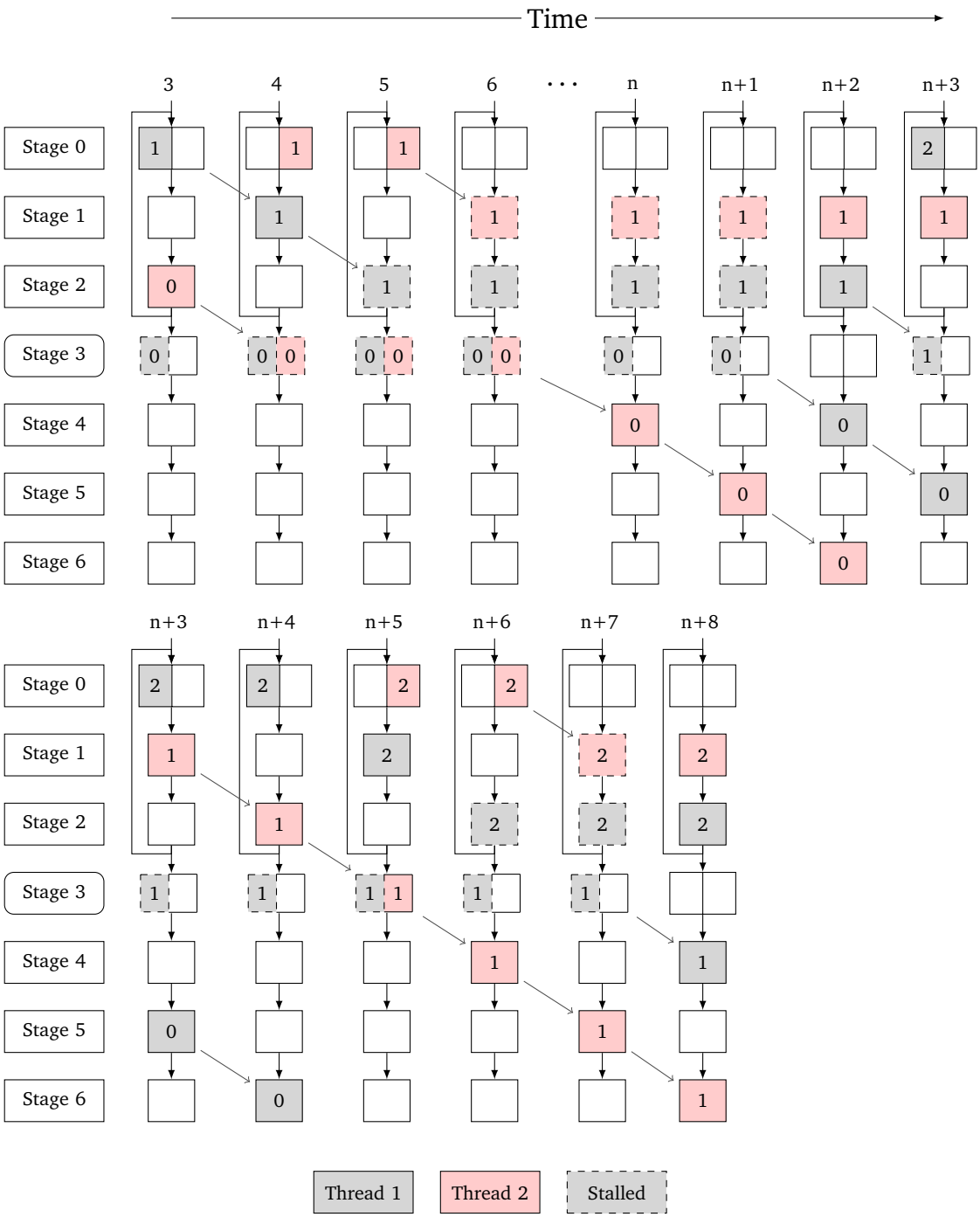


Figure 4.2: Thread Reordering
Iterations are shown as numbered boxes
Dashed iterations are stalled, rounded stages contain VLOs

4.2 General Idea

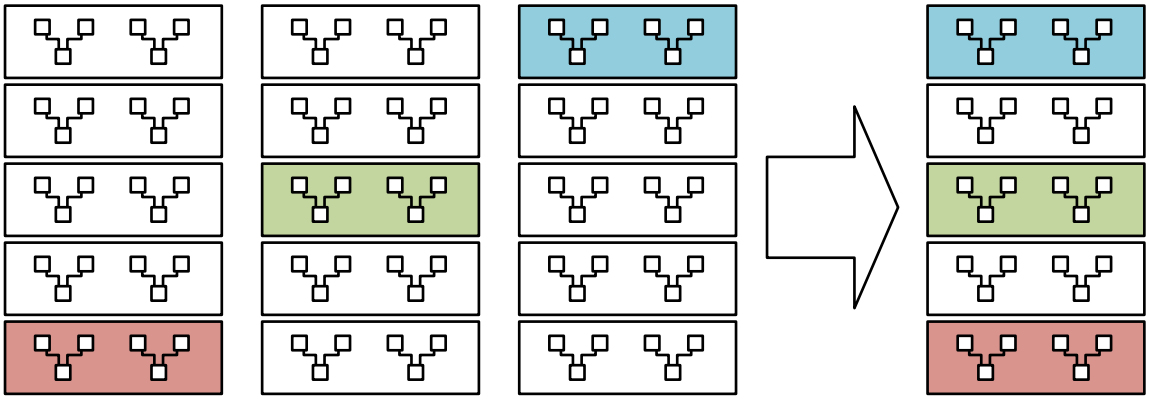


Figure 4.3: Combining three pipeline copies into a single multi-threaded pipeline

The general idea behind the implementation of the multi-threading is that n single-threaded pipeline copies are *merged* into a single pipeline supporting the parallel execution of n threads, as shown in Figure 4.3. This turns each copy into a thread slot in the merged pipeline. The threads are executed in the interleaved manner that was shown in the example for multi-threading with VLOs. To distinguish between the data of each thread a Thread Identifier (TID) is used. Using a Hardware Thread Scheduler (HTS) unit, the threads are dynamically reordered according to the real execution times of the VLOs.

The execution model is an extension of the dynamic II model (Section 3.3). The following sections show all additions which are necessary for the multi-threading support. Because it is based on the dynamic II model, the adapted stage transitions of the multi-threaded model are again shown on the pipeline level. Details of the HTS unit and necessary changes to MCO and VLO handling are shown in the stage and operation levels.

To merge the n single-threaded data-paths, some of the data-paths' elements are replicated per thread to allow thread reordering. These replicated elements will be shown as stacks of the single-threaded element. Note that certain elements which are shown as replicated can be slightly different between the explanation levels. This is because the higher levels omit some details.

4.3 Model Extensions

To add multi-threading to the dynamic II model, several extensions are necessary. These concepts will be described in this section. The fundamental addition is the use of a thread identifier to identify all data and tokens belonging to a thread. Also, each stage of the pipeline is now either a single- or multi-threaded stage,

meaning a stage without or with reordering capabilities. The hardware-software communication uses the TID to communicate with a specific thread. Also the shared memory is partitioned for the use with multiple threads.

4.3.1 Thread Identifier (TID)

The TID is an integer number which is used to identify all data belonging to a single thread. In general, the TID is implemented for n threads as a $\lceil log_2(n) \rceil$ bits width value. At points where a selection out of multiple threads is necessary, multiple threads' TIDs are represented by a n bits width bit-field, where each bit is assigned to a single thread. This second type is called a TID-set.

In the dynamic II model, each token represents a single iteration or dependencies on an iteration. In the multi-threaded model, each token is extended by the TID to associate each iteration or dependency with a thread. Because of the stage based model, the position of data is represented by the stage token, the values are generally stored in the buffers without additional TIDs.

4.3.2 Single-/Multi-threaded Stage

The pipeline stages in the multi-threaded model are either single- or multi-threaded stages. A single-threaded stage is similar to a stage in the dynamic II model and differs just by the addition of a TID in the stage token. A single threaded stage never changes the order of iterations or threads in its output buffer.

On the other hand, a multi-threaded stage has duplicated output-buffers, one for each thread, to allow the reordering of threads. Note that iterations of a single thread are never reordered.

4.3.3 Pipeline Hierarchy

While the pipeline hierarchy itself does not change, some elements at the hierachy level borders are replicated. Figure 4.4 shows that for multi-threading, all buffers in the hierarchy which are used for exchanging parameters and results are replicated so that each thread has its own set of buffers. In addition to that, the TID of each thread that moves from one nesting level into another is transmitted along with the parameters and results. Note that pipeline itself is shared between each thread as it was mentioned in the general idea behind the multi-threading model.

4.3.4 HW-SW Communication

The communication between hardware and software in the multi-threaded model is generally equivalent to both the static and dynamic II model. The difference

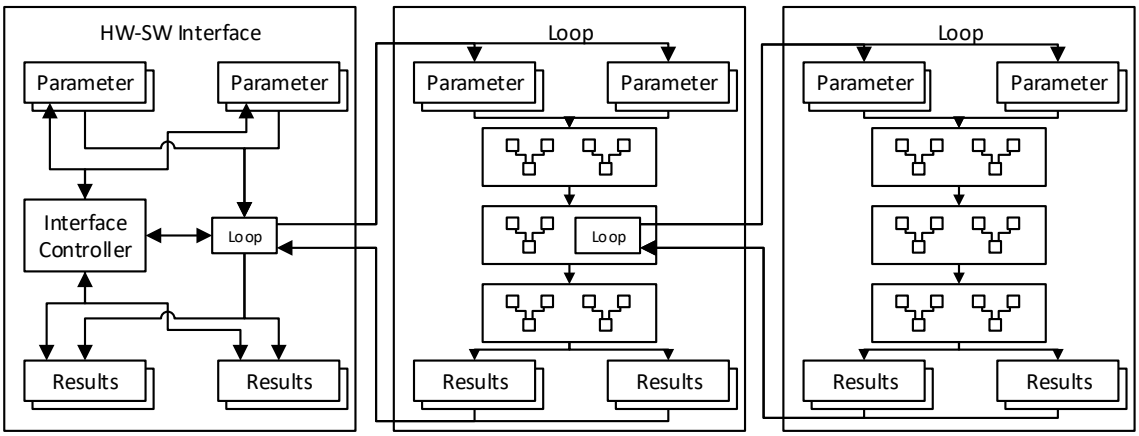


Figure 4.4: Hierarchy of nested loop pipelines and HW-SW Interface

again is that the TID has to be integrated into each communication channel. The details are shown in each target system description. The general idea is to use the TID in each communication message to indicate to which thread it belongs. That way the interface controller can write and read or control the individual threads. The shared memory was partitioned into per-thread cache area to prevent interference between the individual threads.

4.3.4.1 Registers

In the hardware-software interface, all registers (see Section 3.1.6.1) are replicated for each thread. The TID is used to address the registers belonging to a specific thread. While the hardware can access registers for multiple threads simultaneously, the software can only access the registers for a single thread at a time. The cause for this is that all target systems currently only have a single communication channel to these registers shared between all threads.

4.3.4.2 Shared Memory

As the multi-threaded model currently supports no way to synchronize inter-thread communication using semaphores, barriers or other methods, the memory for each has to be thread exclusive. For that, the shared memory is partitioned into independent areas, with one area for each thread, as shown in Figure 4.5.

To prevent cache thrashing, the cache system configured to thread specific caches in the target systems where this configuration granularity is possible. All target systems use individual cache ports for each thread to allow the simultaneous access of multiple threads.

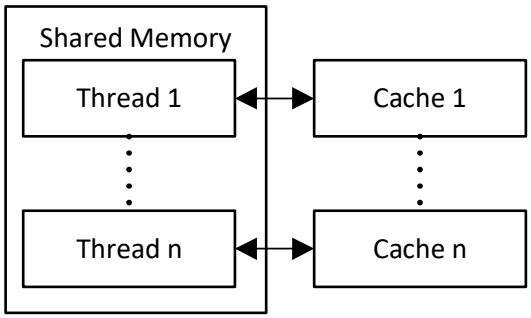
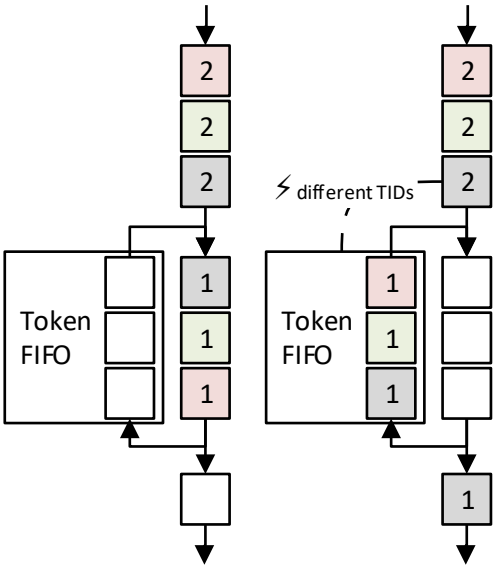


Figure 4.5: Shared memory and cache per-thread partitioning

4.3.5 Pipeline level

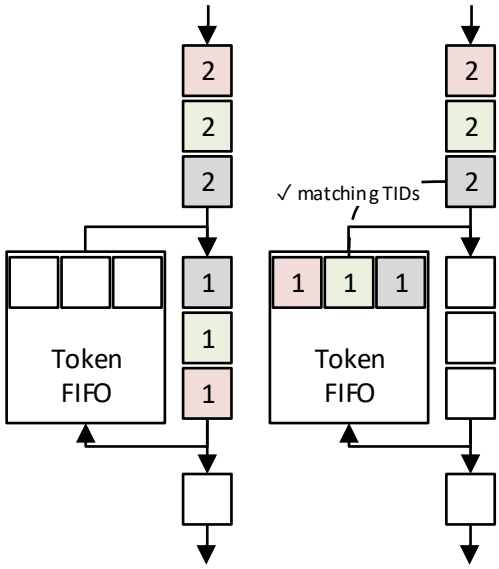
Because the multi-threaded model is based on the dynamic II model, the stage transitions are based on the rules of the dynamic II model with adaptations for supporting multiple threads. In general, only the controller and token elements are replicated per thread to minimize the area overhead of multi-threading. On the pipeline level, the description will be focusing on these elements. The transition conditions are resolved individually for each thread by these duplicated elements. Then, from all threads which are ready to advance into the next stage, an additional Hardware Thread Scheduler (HTS) unit selects one thread that is then allowed to advance. The TID, used to identify the thread of each token or data, is stored explicitly with the stage token in single-threaded stages and implicitly associated with the duplicated per-thread elements.

Figure 4.8 shows the two detailed transitions examples from Figure 3.12 with the duplicated elements and the additional HTS unit. Both examples show a transition from a multi-threaded to a single-threaded stage with either an incoming (Figure 4.8a) or outgoing (Figure 4.8b) inter-iteration dependency. Note that the token and data FIFO for the inter-iteration dependency always has per-thread replicated elements. This is done to avoid deadlocks. Figures 4.6 and 4.7 show a comparison of two examples with a single- and multi-threaded token FIFO. In Figure 4.6, between a) and b) the single-threaded token FIFO is filled with dependency tokens (introduced in Section 3.3.2) from Iteration 1 of each thread (red, green, grey). But as the order of Iteration 2 of each thread got reversed by dynamic thread reordering, the token at the front of the FIFO is for another thread than the stage token of the earliest Iteration 2. Thus, they cannot be used in combination with the stage token. Because this never resolves, a deadlock situation is reached. In Figure 4.7, the deadlock is avoided by using a multi-threaded token FIFO. From this FIFO, the token of each thread can be combined with whichever thread requires it.



(a) Shortly before deadlock **(b)** Deadlock

Figure 4.6: Single-threaded FIFO (only first token can be accessed)



(a) No deadlock **(b)** No deadlock

Figure 4.7: Multi-threaded FIFO (direct access to all thread tokens)

After the following detailed transition examples, all cases and their rules will be shown with simplified figures as in the dynamic II model's explanation. Again, both examples show a transition from a multi-threaded Stage A to a single-threaded Stage B, which differ in three points.

1.) The first difference is whether an incoming or outgoing inter-iteration dependency is involved in the transition. In Figure 4.8a, the incoming dependency comes from a stage *after Stage B*. In Figure 4.8b, the outgoing dependency goes to a stage *before Stage A*.

Both have in common that in the multi-threaded Stage A, the stage token, local controller and data buffer are duplicated for each thread. The operations themselves are not duplicated. Each of the duplicated local controllers evaluates only the *Ready* signal of the other duplicated elements belonging to the same thread. Here these are the Stage Token and Token FIFO. From the VLO in Stage A, the local controller receives a signal for each thread its execution is completed for that particular thread (the details of VLO execution are shown in Section 4.3.7.2 in the operation level). The single *Ready* to accept data from Stage B is connected to each local controller instance because the single-threaded stage has a single buffer for all threads.

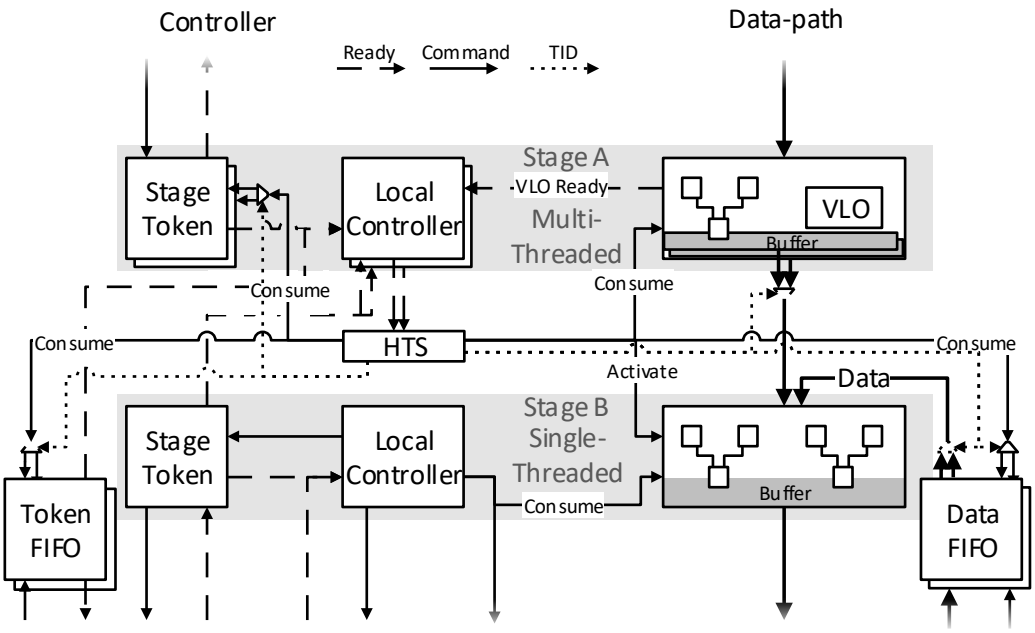
2.) The second difference is in whether the Token FIFO is checked if a token is available or if it is *Ready* to accept a new token respectively. The status of the Token FIFO corresponds with the Data FIFO which is not checked.

In both cases the HTS unit then selects one of the threads, which has all conditions fulfilled, to advance into the next stage. The HTS *Command* signals to consume tokens and data and activate the target stage and buffers. Where it is necessary to select from one of the duplicated elements, the HTS unit uses the TID to tell a multiplexer which thread's element it has to select.

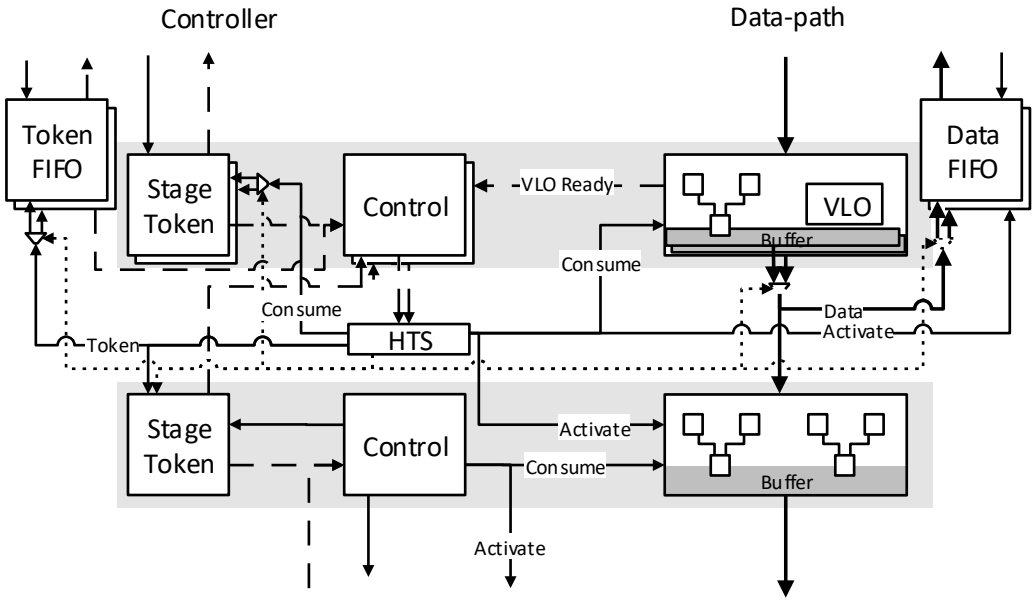
3.) The last difference is in which data is consumed and which elements are *Activated* to accept and store new data or tokens. In both cases, multiplexers are used for the selection and consumption of the stage token and data buffer from Stage A. In Figure 4.8a, they are also used for the selection and consumption of the token and data FIFO from the incoming inter-iteration dependency. In Figure 4.8b, they are also used for the selection and activation of the token and data FIFO for the outgoing inter-iteration dependency.

The combination of these multiplexers for selection and the duplicated buffers will further be called the Thread Context Storage (TCS). In Figure 4.9b, overlays show which parts of the stage logic in Figure 4.9a constitute the TCS.

Section is continued after Figure 4.8 and Figure 4.9

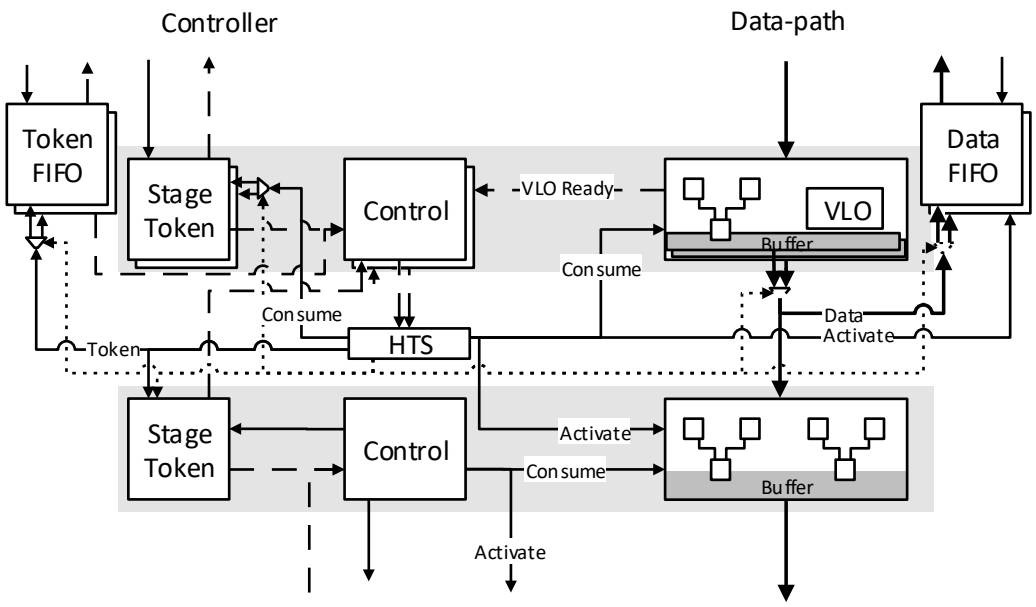


(a) Incoming inter-iteration dependency from later stage to Stage B

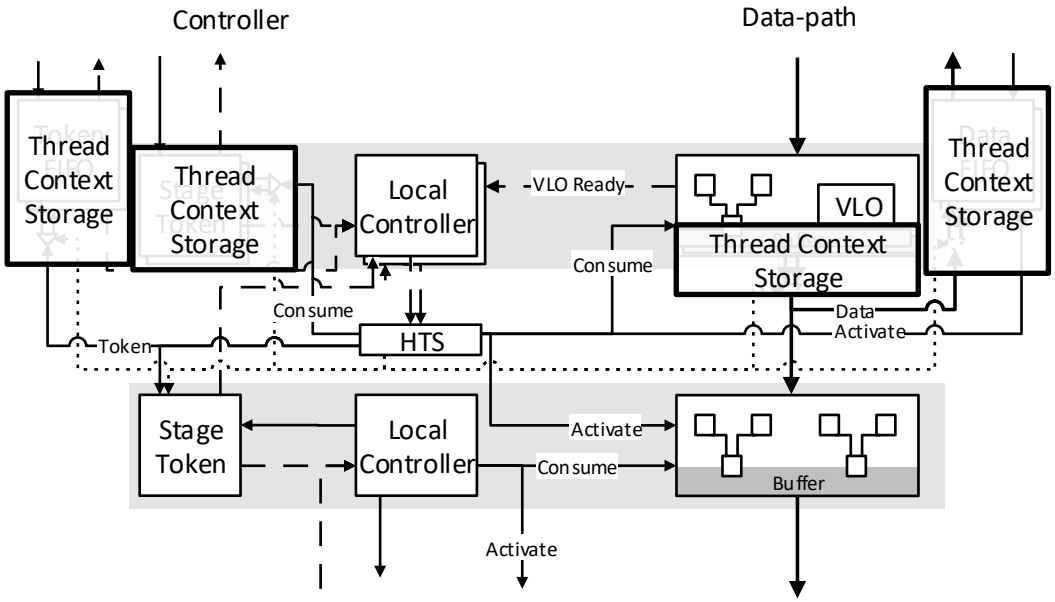


(b) Outgoing inter-iteration dependency from Stage A to earlier stage

Figure 4.8: Stage transitions (detailed overview)



(a) Outgoing inter-iteration dependency from Stage A to earlier stage



(b) Outgoing inter-iteration dependency from Stage A with overlaid Thread Context Storage

Figure 4.9: Location of Thread Context Storage (TCS) (overlaid in bottom figure)

In general, it is assumed that a stage is only multi-threaded when it contains VLOs. Because of this, in the following simplified figures of the transition variants, the distinction between stages with and without VLOs is disregarded. Instead, the variants are classified only by whether the source and target stage is single- or multi-threaded, and whether the transition contains an incoming or outgoing inter-iteration dependency. The number of tracked dependencies is reduced with the same method that was described in Section 3.3.2.1 for the dynamic II model, only that they are now tracked on a per thread basis.

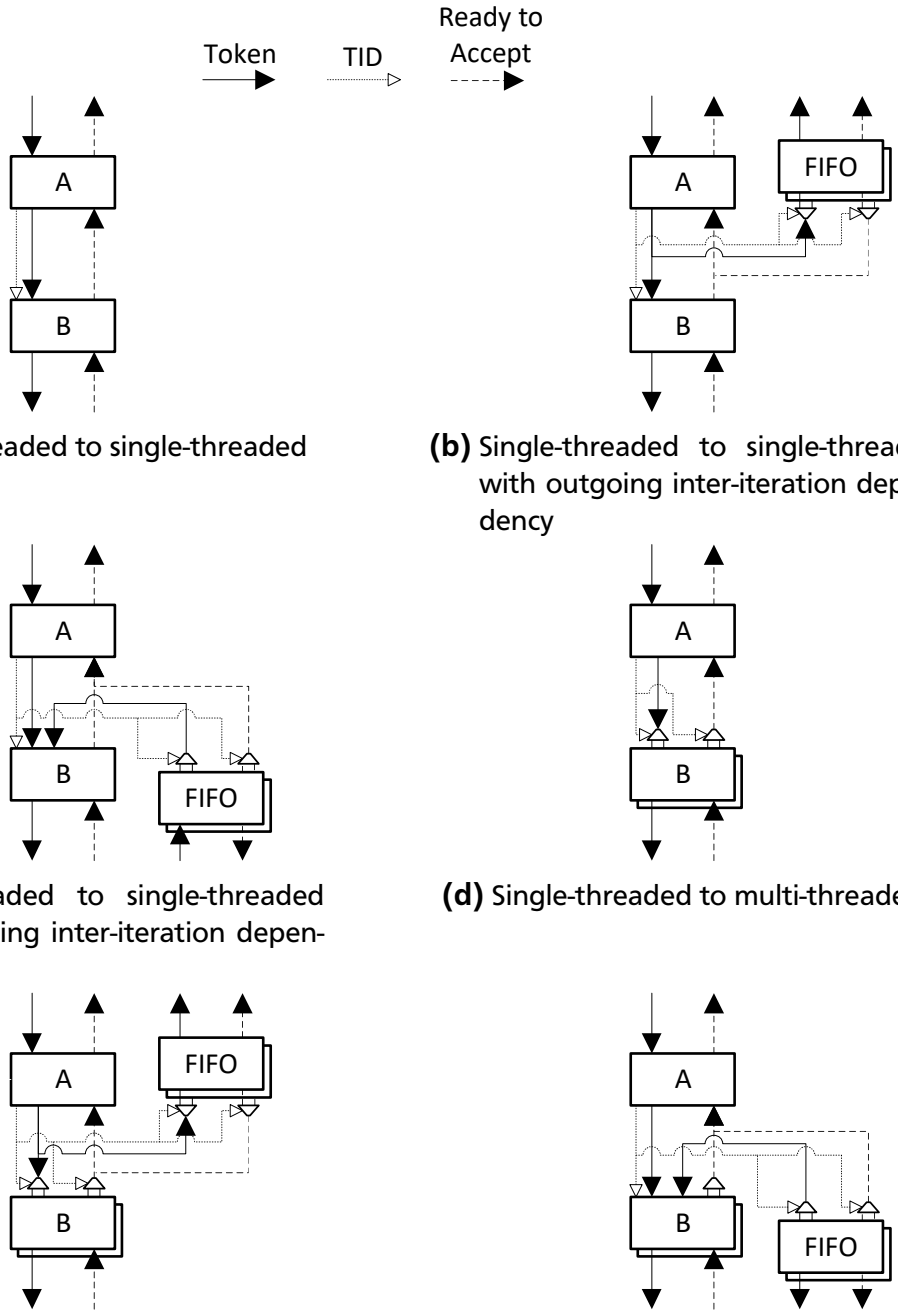
Table 4.1 shows a summary of all stage transition variants in the multi-threaded model. The table is split into three sectors. The first points to the figure of the transition. The second shows the parameters of the transition. Depending on the parameters, the configuration of the controller elements is shown in the third sector.

The transition parameters are as follows: Is the source stage single- or multi-threaded? Is the target stage single- or multi-threaded? Does the transition involve an outgoing, incoming or no inter-iteration dependency?

The configuration elements are as follows: Is a HTS unit required? Which storage elements have to be ready to accept data? Which storage elements have to have data ready to be consumed?

From the table (and the associated Figure 4.10) it can be derived that whenever the source Stage A is multi-threaded, the controller requires a HTS unit to select one of the ready threads to move to target Stage B. When only the target Stage B is multi-threaded, the TID of the thread in the source Stage A is used to select the appropriate thread slot in the TCS of the target stage and eventual inter-iteration dependency FIFO. For an incoming inter-iteration dependency the dependency FIFO is check for a token ready to be consumed. For an outgoing inter-iteration dependency the dependency FIFO is checked if its ready to accept a token. Both checks are done in addition to checks of the source and target stage.

As in the dynamic II model, the buffers in each stage can be configured to use queues instead of register. But in the multi-threaded model their implications on the throughput has increased. Section 4.3.8 will show an example of why some stages should use queues and the rules with which the compiler decides where queues are used.



(a) Single-threaded to single-threaded

(b) Single-threaded to single-threaded with outgoing inter-iteration dependency

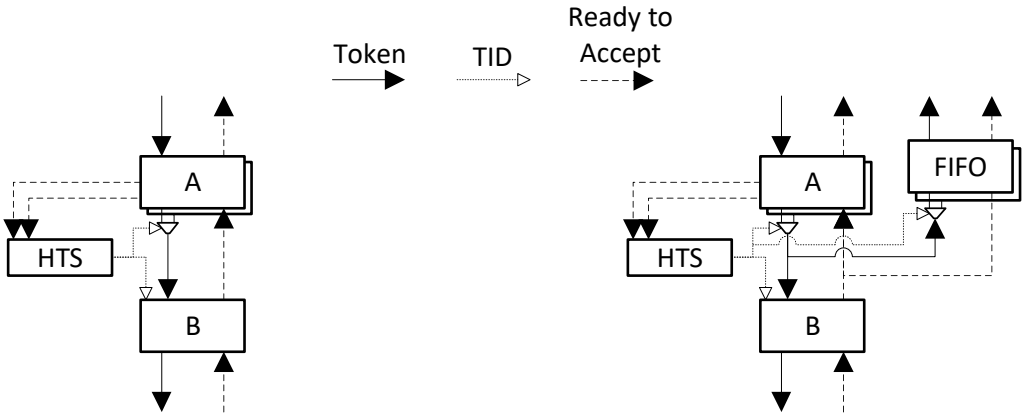
(c) Single-threaded to single-threaded with incoming inter-iteration dependency

(d) Single-threaded to multi-threaded

(e) Single-threaded to multi-threaded with outgoing inter-iteration dependency

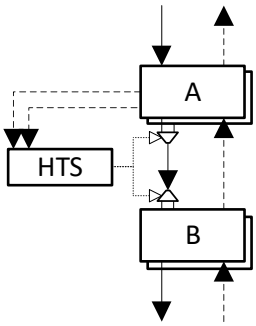
(f) Single-threaded to multi-threaded with incoming inter-iteration dependency

Figure 4.10: Stage transitions in multi-threaded model (simplified, only control signals, data connections are omitted (continued on next page)

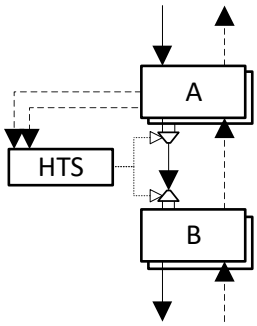


(g) Multi-threaded to single-threaded

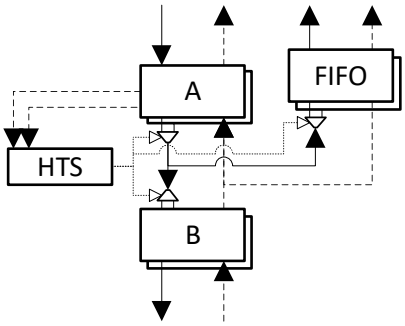
(h) Multi-threaded to single-threaded with outgoing inter-iteration dependency



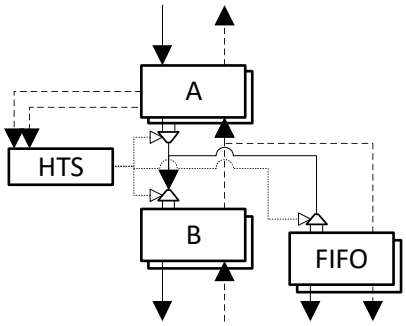
(i) Multi-threaded to single-threaded with incoming inter-iteration dependency



(j) Multi-threaded to multi-threaded



(k) Multi-threaded to multi-threaded with outgoing inter-iteration dependency



(l) Multi-threaded to multi-threaded with incoming inter-iteration dependency

Figure 4.10: Stage transitions in multi-threaded model (simplified, only control signals, data connections are omitted (continued from previous page)

Figure	Parameters			Configuration		
	Threading		Inter Iteration Dependency	HTS required	Ready to accept	Ready to be consumed
	Source	Target				
4.10a	Single	Single	No	No	Target	Source
4.10b	Single	Single	Outgoing	No	Target, FIFO	Source
4.10c	Single	Single	Incoming	No	Target	Source, FIFO
4.10d	Single	Multi	No	No	Target	Source
4.10e	Single	Multi	Outgoing	No	Target, FIFO	Source
4.10f	Single	Multi	Incoming	No	Target	Source, FIFO
4.10g	Multi	Single	No	Yes	Target	Source
4.10h	Multi	Single	Outgoing	Yes	Target, FIFO	Source
4.10i	Multi	Single	Incoming	Yes	Target	Source, FIFO
4.10j	Multi	Multi	No	Yes	Target	Source
4.10k	Multi	Multi	Outgoing	Yes	Target, FIFO	Source
4.10l	Multi	Multi	Incoming	Yes	Target	Source, FIFO

Table 4.1: Stage transitions in multi-threaded model

4.3.6 Stage level

The logic of each (possibly duplicated for each thread) local controller is identical to the logic described in Section 3.3.3 for the dynamic II model. The inputs and outputs of the logic are split implicitly by their associated TID. Thus for single-threaded stages in the multi-threaded model, no changes have been made besides addition of the TID to the stage token for basic operations. The TID in the stage token is sufficient, because the stage token FIFO and the buffers of the basic operations are always kept in sync. If a stage contains a VLO, the model always handles the stage as a multi-threaded stage so VLOs do not change anything for single-threaded stages. The handling of MCOs depends on whether at least one of their stages is on a multi-threaded stage.

For multi-threaded stages a number of changes are necessary in the multi-threaded model. Figure 4.11 shows, that all simple buffers are replaced by Thread Context Storage (TCS) buffers as indicated in the pipeline level. For each thread the TCS contains an exclusive buffer. Using the TID one of the threads' buffers can be arbitrary selected. This is used by the HTS unit to control which thread transitions to the successor stage. The HTS selects a thread and all its data from the TCS buffers in the stage. The handling of VLOs is quite a bit different in the multi-threaded model and is described in detail in Section 4.3.7.2. The general idea is that the result of the VLO is stored in a TCS which allows the parallel completion of the same VLO for different threads. On the stage level, however, the

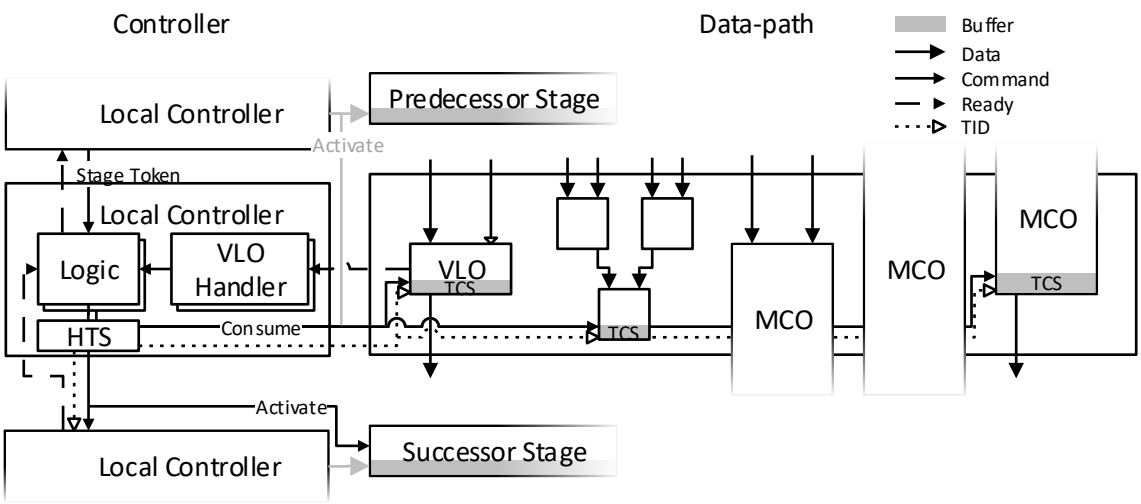


Figure 4.11: Controller excerpt and multi-threaded pipeline stage with chained basic operations, a VLO, input, intermediate and output stage of MCOs. Shaded parts of operations contain buffers (can be TCS) to store data until it is consumed by the successor stage

change is that the duplicated local controller's VLO handler each check whether the VLO is finished for that particular thread. This is generally done by checking for which threads the TCS of that VLO has data available.

Now that the changes (or lack thereof) were shown for both single and multi-threaded stage, the handling of MCOs can be explained. As previously stated, the handling of MCOs depends on whether at least one of the MCO's stages is multi-threaded. If no stage is multi-threaded, their handling is unchanged to the dynamic II model. If a single stage is multi-threaded, each data value in the MCO has to be associated with TID of its parameters at the input stage. Because the order of the threads can change at a multi-threaded stage, the output buffer of the MCO must be configured as a TCS, to allow the arbitrary thread selection. This is necessary even if the output stage itself is single-threaded. An example for this can be seen in Figure 4.12.

The example shows a three cycle MCO at some point in the data-path. At $t = 1$, T_1I_0 (Iteration 0 of Thread 1) has already entered the MCO in Stage 0. It is directly followed by T_2I_0. At $t = 2$, T_1I_0 stalls at the VLO in Stage 1 and T_2I_0 enters the MCO. At $t = 3$, T_2I_0 finishes the VLO immediately and overtakes T_1I_0. At $t = 4$, T_1I_0 leaves the MCO and is stored in TCS added to the MCO. At $t = 5$ finally, T_2I_0 also leaves the MCO but skips the TCS as it is immediately moved to the next stage.

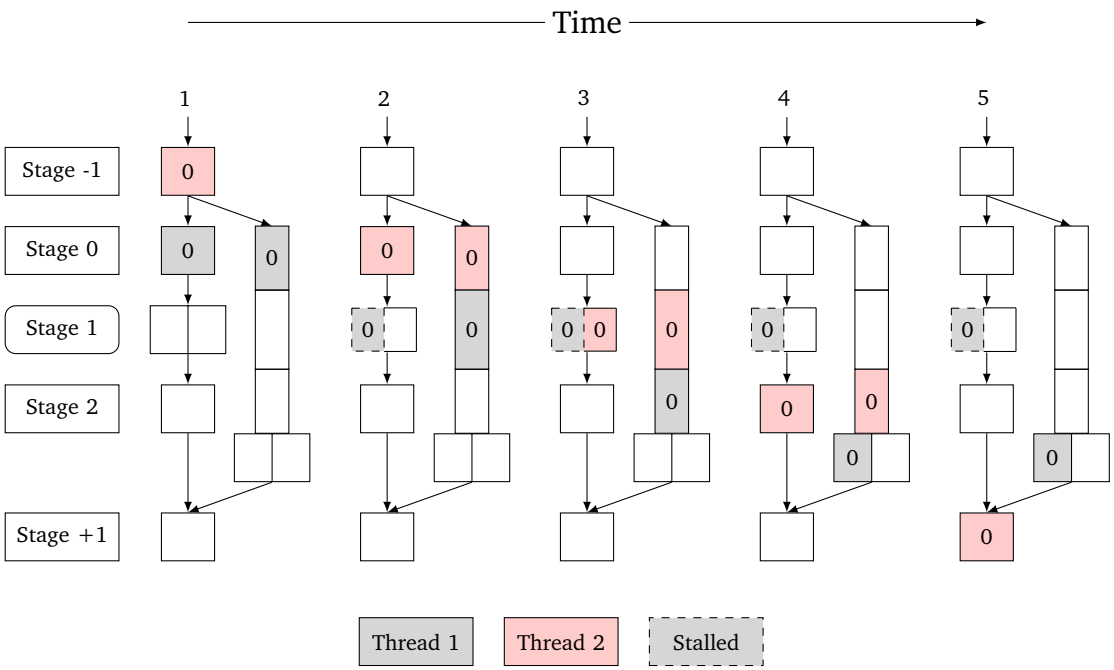


Figure 4.12: TCS after MCO to support thread reordering (MCO is right, data-path left)

4.3.7 Operation level

Depending on the stage an operation is located on, the output buffer is either a simple register, queue, or TCS. Because single-threaded operations in single-threaded stages are very similar to operations in the dynamic II model, only the figures for operations in multi-threaded stages will be shown here. Single-threaded stages can only contain basic or multi-cycle operations, as VLOs are only used in multi-threaded stages. For this single-threaded behaviour, see Section 3.3.4. On the other hand, multi-threaded stages can contain all types of operations, but only MCOs and VLOs need extra handling besides the configuration of the buffer as a TCS, which is required for all operations in a multi-threaded stage (as explained in Section 4.3.6).

4.3.7.1 Multi-Cycle Operation (MCO)

Besides the configuration of the MCO's buffers (explained in Section 4.3.6) nothing else has to be changed for the MCO handling. Each MCO still uses the same single-threaded operators from the dynamic II model.

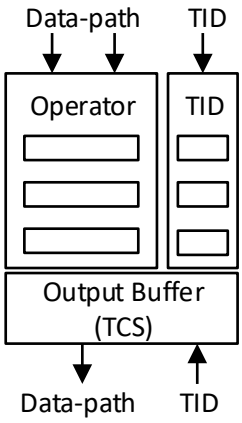


Figure 4.13: MCO for multi-threaded stages (3 cycles)

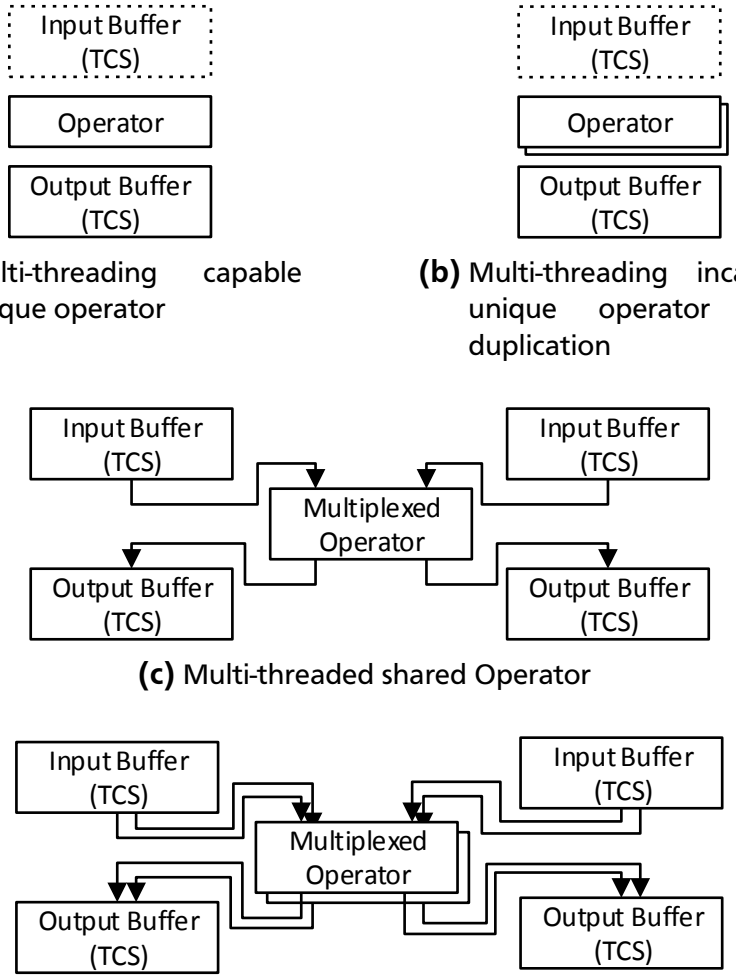
In addition to the buffer's configuration as TCS, the operator is accompanied in lockstep by a simple pipeline for keeping the TID of the input values in sync with the operator's pipeline. When the result comes out of the operator's pipeline, the TID is used to assign the result to the correct thread in the TCS. Finally, the TID is used to select the result of the MCO by the HTS (shown in Stage level).

4.3.7.2 Variable Latency Operation (VLO)

Similar to the dynamic II model, a VLO can either use resource sharing or not through a shared or unique instance of the operator respectively (see Section 3.3.4.2 for definition of *shared* and *unique* operator). In addition to that, the multi-threaded model differentiates between multi-threading *capable* and *incapable* operators. A multi-threading capable operator must be able to execute multiple threads. It should also be somehow possible to execute them in parallel by distinguishing the data using TIDs. Unlike MCOs this cannot be handled with a simple lockstepped pipeline, as the number of cycles is unknown and threads could be reordered in the VLO.

Combining these two distinctions results in four types shown in Figure 4.14. They are ordered by resource sharing followed by their multi-threading capability.

Figures 4.14a and 4.14b show unique operator instances which are either multi-threading capable or incapable. In case the operator does not support multi-threading, the operator is duplicated for each thread. Currently the only multi-threading capable operators are nested loops. As the memory system currently does not support TID, memory accesses are treated as multi-threading incapable operators. Also, closed IP cores with a variable latency which cannot associate a TID with each data input are not multi-threading capable. In all these cases as a workaround, separate instances of the operator are created for each thread by duplication.



(a) Multi-threading capable unique operator

(b) Multi-threading incapable unique operator using duplication

(c) Multi-threaded shared Operator

(d) Multi-threading incapable shared operator using duplication

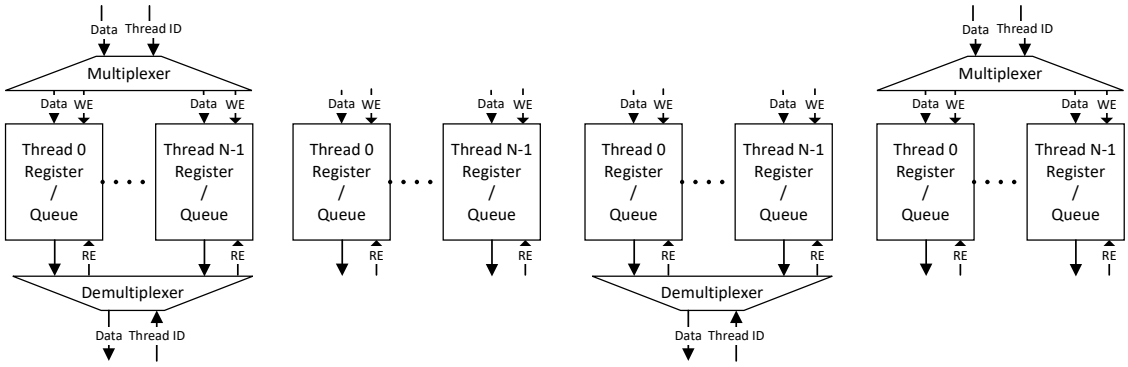
Figure 4.14: VLO types in multi-threaded execution model

Figures 4.14c and 4.14d show shared operator instances which either support or do not support multi-threading. In case the operator does not support multi-threading, the whole multiplexing logic is duplicated for each thread. The multiplexing logic itself is the same as in the dynamic II model shown in Section 3.3.5. In case of a multi-threaded operator, the TID is associated with each calculated value in the operator.

Memory accesses in the multi-threaded model are currently handled with duplicated shared operators. The memory operators have to be duplicated as they do not directly support multi-threading. Each thread uses its own memory region and all memory operators of a single thread are connected to a single cache. This is done to prevent cache thrashing by unrelated threads (see Section 4.3.4.2).

Nested loops are handled by either multi-threaded unique or shared operators depending on whether hardware functions (see Section 3.3.6.2) are used. When the nested loop is not shared (multiplexed between multiple call-sites), the simple case of a multi-threading capable unique operator (Figure 4.14a) can be used. However, when the nested loop is used from multiple call-sites, a multi-threaded shared operator (Figure 4.14c) has to be used.

4.3.7.3 Thread Context Storage (TCS)



(a) Single write and read **(b)** Parallel writes and reads **(c)** Parallel writes and single read **(d)** Single write and parallel reads

Figure 4.15: Thread Context Storage variants

As explained on the pipeline level (Section 4.3.5), the TCS is a necessary element for thread reordering. With it the HTS can select all data and tokens belonging to a thread in multi-threaded stages and initiate the transition into the next stage.

Each of the TCS variants shown in Figure 4.15 are based on the basic idea of duplicating buffers (Figure 4.15b). The other variants are created by adding multiplexers for input or output selection where needed.

The basic variant (Figure 4.15b) allows the parallel write and read of values from all thread buffers. In general the multi-threaded model does not use this. In the future it could be used to create stages which allow the parallel processing of all threads by duplicating all operators for each thread. To do this, however, it would require at least two such stages in succession, allowing multiple threads to transit simultaneously. This is because stages generally only have TCS as an output buffer. So the output buffer of the previous stage has to provide the data in parallel and the next TCS has to store all threads in parallel. This could increase the throughput for the cost of increased resource requirements.

The most common variant (Figure 4.15a) allows only a single thread write and read to the buffers at a time. But the multiplexers allow the HTS unit to arbitrarily select data from the threads, which is necessary for reordering.

Variants (Figures 4.15c and 4.15d) are both used for the implementation of VLOs with duplicated operators. 4.15d is used for the input buffer and 4.15c for the output buffer. This way the duplicated operator VLO type can execute all operators in parallel, where it is possible that multiple operators want to read new parameters or write their result at the same time.

4.3.8 Queue Usage

To show the impact of queues in the multi-threaded model, the behaviour without or with queues, as shown as in Figures 4.16 and 4.17 respectively, is analysed and compared. Figure 4.17 is configured to contain a two entry queue for each thread in Stage 3's TCS. Pipeline bubbles are omitted in the behaviour for brevity. Stage transitions of iterations are highlighted by diagonal arrows between the stages and time steps. Iterations of Thread 0 are highlighted by grey entries, Iterations of Thread 1 by red entries. In the following description a specific iteration x for a specific thread y will be abbreviated by $T_y I_x$. For example Iteration 0 of Thread 1 would be $T_1 I_0$.

From $t = 3$ until $t = 5$ both configurations have equal behaviour. At $t = 6$ this changes, however. The queue in Stage 3 allows $T_0 I_1$ to enter Stage 3, whereas without the queue the iteration and pipeline is stalled in Stage 2. With the queue, the pipeline stalls after Iteration 0 and 1 of both threads have entered Stage 3.

In both configurations, at $t = n$, $T_1 I_0$ finishes the memory access in Stage 3 and can continue to Stage 4. Now assuming that all further memory accesses of Thread 1 are cache hits and thus resolved in one cycle, the configuration with queue allows that $T_1 I_1$ can move to Stage 4 at $t = n + 1$. However, without the queue $T_0 I_1$, which is stalled by Iteration 0 of the same thread, in turn stalls and thus prohibits $T_1 I_1$ to enter Stage 3 and do the memory access. Only when at $t = n+2$ the memory access for $T_0 I_0$ is finished, $T_1 I_1$ can enter Stage 3 at $t = n+4$.

Comparing $t = n + 5$ for both configurations shows that with the queue Thread 1 already could execute the memory access for Iteration 2 while without it could only execute the access for Iteration 1. At $t = n + 8$ it can be noticed that the slower Thread 0 only finished its Iteration 1 in Stage 8, independent from the selected configuration.

The example shows how queues can drastically improve the multi-threaded model by giving more leeway for threads to be reordered. As the length of queues is finite, queues cannot magically resolve all problems when the pipeline should be flooded by iterations of a Thread with slow memory accesses. However, it will be shown in the evaluation that queues allow faster overall execution.

In general queues have at least 16 elements because the distributed ram structures on the target Xilinx FPGAs have wasted resources when used with smaller queues. However, when a queue is used in the TCS at the end of MCO, it potentially has to be configured for a larger queue size. Besides the addition of multi-threading this corresponds to queue size configured in the single-threaded model for MCOs in Section 3.3.4.1

4.3.9 Placement of Multi-threaded stages

While technically each stage of the pipeline can be multi-threaded, it generally makes sense to do so where multiple threads can catch up to each other. In other places, there will always be only a single thread in the stage, never using the reordering capabilities of the stage. In turn this leads to unused wasted resources. As a simple approximation, this work assumes that all stages which contain at least one VLO are candidates for a multi-threaded stage.

Remembering the pipeline hierarchy of nested loops (Section 4.3.3), stages with a VLO representing a nested loop should always be a multi-threaded stage. Without multi-threading, only a single-thread could enter the nested loop, preventing all multi-threading in the nested loop and all its children. Because of this, two categories of multi-threaded stages, *mandatory* and *optional*, are defined.

Currently, the only stages which are mandatory multi-threaded are all stages with a VLO representing a nested loop. While all mandatory multi-threaded stages have to be included in the pipeline to allow multi-threading at all, the compiler can select which optional multi-threaded stages are used.

Optional stages are not necessary for the interleaved multi-threading. However, they allow the reordering of threads. Thus, to improve the throughput, some optional multi-threaded stages should be used, i.e. stages with memory accesses that are often stalled. In Section 9.3, selection heuristics will be shown and evaluated by analysing their impact on overall execution time and resource efficiency. Without these heuristics the default behaviour of the compiler is currently to create all stages with VLO as multi-threaded stages.

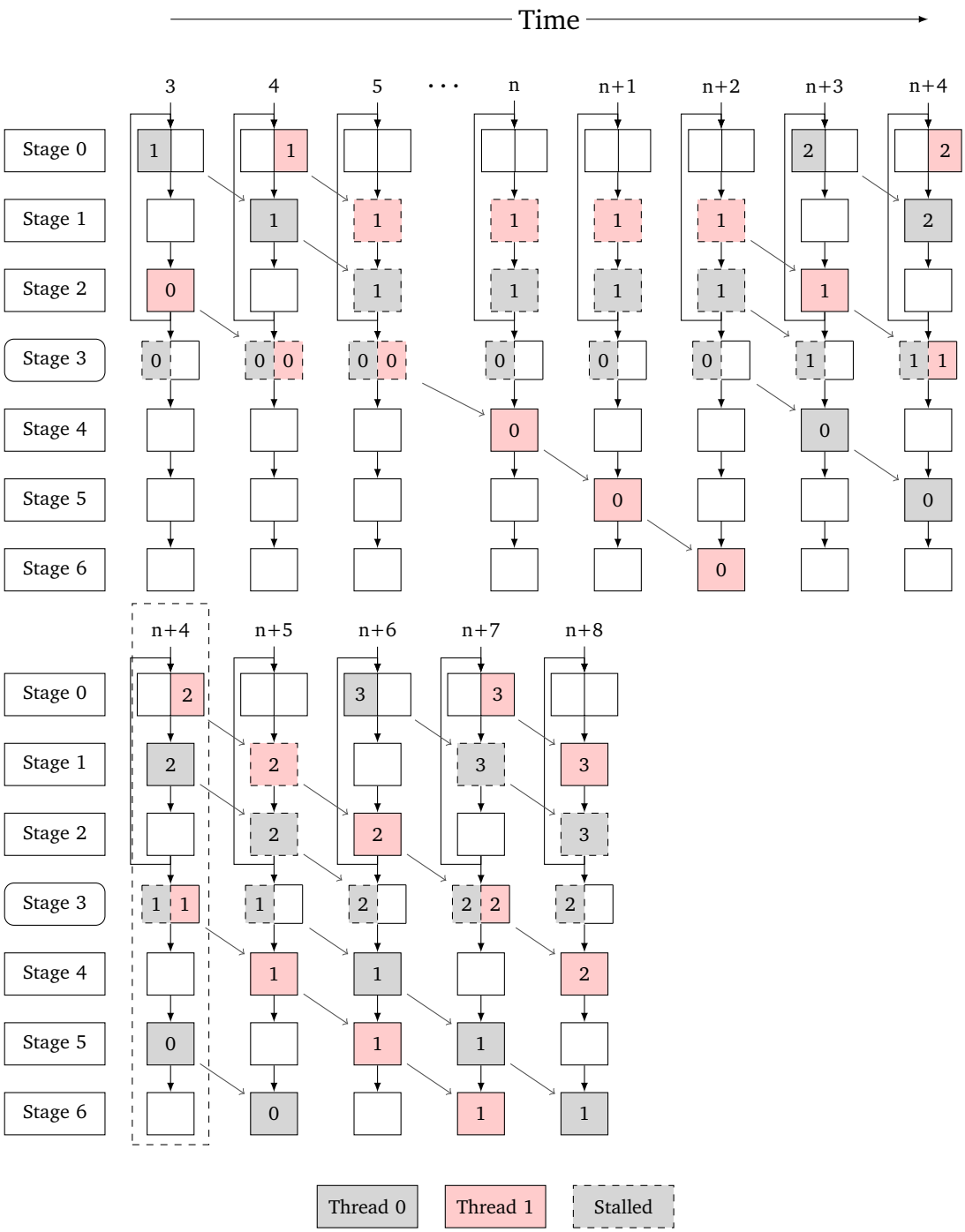


Figure 4.16: Thread Reordering without Queues (state $t = n + 4$ in dashed box is duplicated)
Iterations are shown as numbered boxes
Dashed iterations are stalled, rounded stages contain VLOs

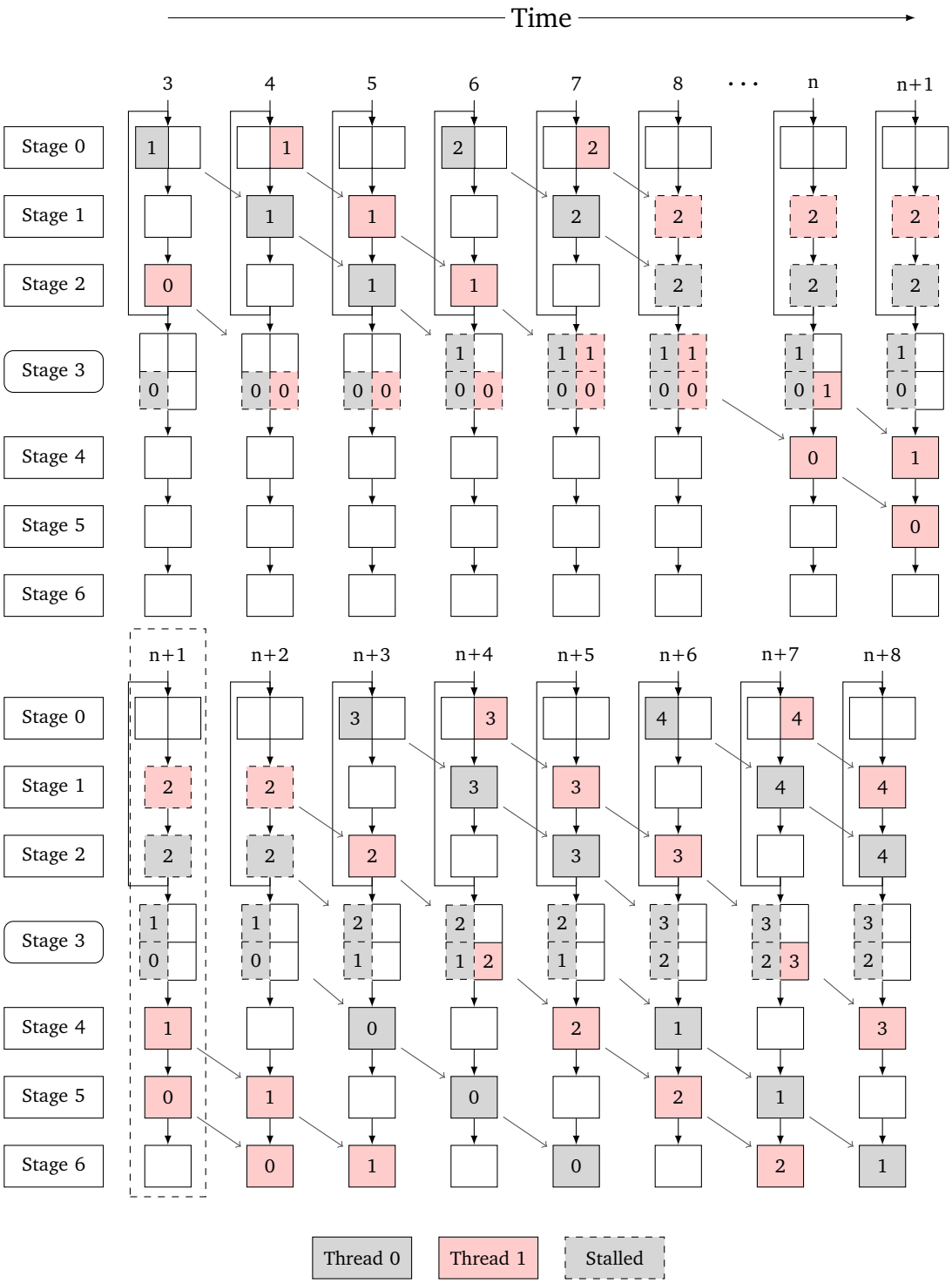


Figure 4.17: Thread Reordering with Queues (state $t = n + 1$ in dashed box is duplicated)
Iterations are shown as numbered boxes
Dashed iterations are stalled, rounded stages contain VLOs

5 Basic Block Execution Model

Through the evaluation of the pipelined model and comparing it with the basic block models from other works it became apparent that not all applications are suited for a pipelined execution model. As an experiment, a multi-threaded basic block execution model was developed. This model executes each basic block with the pipelined execution model as loops having a single iteration. In the future, this will allow further refinements to mix the executions models as described in Section 10.1. Two variations were implemented for the basic block execution model, because it was noticed that the first variant is quite limited in its applicability.

The first implementation is based on the extraction of basic blocks from Section 3.3.6.1. Here, each basic block will be extracted into its own nested loop. These nested loops are then executed as VLOs within the pipelined parent loop. The evaluation of this approach showed that it has shortcomings which led to deteriorated performance for many benchmarks.

The bad performance results arise from a high II which is caused by the pipelined execution model of the parent loop. The II in this implementation is still dependent on the longest path through all basic blocks, even when they are not executed. Figure 5.1 shows such a case. Assuming all basic blocks require a clock cycle to execute the path from BB0 to BB4, they could be executed in two cycles when executed with a FSM. By using a pipeline as the parent of the basic blocks, the stages containing BB1, BB2 and BB3 have to be activated, even if they are not executed. This results in always requiring 4 cycles to execute a iteration while a basic block model requires at best only two cycles executing BB0 and BB4. In all other cases it uses the same four cycles as the pipelined model. The extraction of basic blocks can be useful to reduce the II when used in the right conditions (rarely executed basic blocks with long operations) as was shown in Section 3.3.6.1.

For situations without these right conditions, a second implementation was developed which is still based on the extracted basic blocks, but instead of trying to execute them as VLOs in a pipelined loop they are executed with a generic FSM based controller. One of the first uses of such a FSM based controller is in [Cam91]. While in that work it was used for synthesis of behavioural VHDL, it shows the concept of using a FSM to model a CFG. In both variants, the execution of a basic block takes at least 3 to 5 clock cycles depending on used execution model for the block. Currently the execution models are not optimised for very short pipelines or basic blocks.

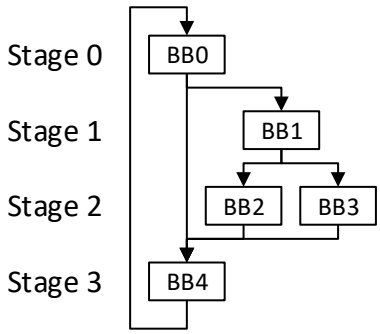


Figure 5.1: Unbalanced basic blocks scheduled to pipeline stages

As LegUp uses an implementation of this model and is readily available, the implementation in this work takes some cues from LegUp. In general, LegUp executes each basic block as a combination of combinatorial logic with registers controlled by their FSM based controller.

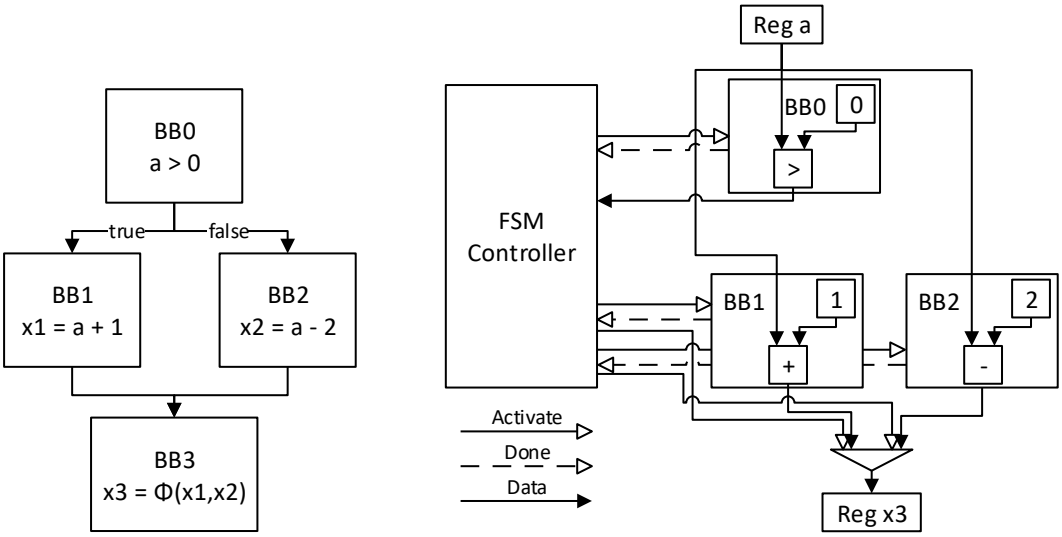
In Nymble this was adapted to execute each block with the pipelined execution model[1] and store the results in registers located in the basic block graph as shown as in Figure 5.2. For each value in the SSA-CFG a register is generated. If a value is the result of ϕ-instruction, an input multiplexer of the register is controlled by the FSM controller according to last executed block. In general, the basic block loops only execute a single iteration each time they are activated, but it will be possible to integrate whole nested loops as a pseudo basic block. This idea is described more in Section 10.1. The controller *activates* a basic block according to its state and waits until the block signals that is *done*. If the block contains branch decision, the result of the operations evaluating the decision is sent to controller upon finishing the block. The controller activates the appropriate successor according to the CFG.

The basic block model and the basic block extraction are evaluated and compared to each other in Section 9.4.

5.1 Multi-threading

To allow for multi-threading with a FSM that usually only has a single state, two problems had to be solved. As each thread requires its own state, a separate FSM instance is created for each thread. The second problem is how to handle the block transitions when multiple threads want to enter the same block, as shown in Figure 5.4. Nested loops in the pipelined model are created with the assumption that only one thread at a time enters them. By solely using the pipelined model, this is guaranteed by the model itself. However, the separate instances of the FSM

[1] Does not have to be the pipelined model. All models can be mixed, see Section 10.1.



(a) SSA-CFG **(b)** Basic Block Implementation

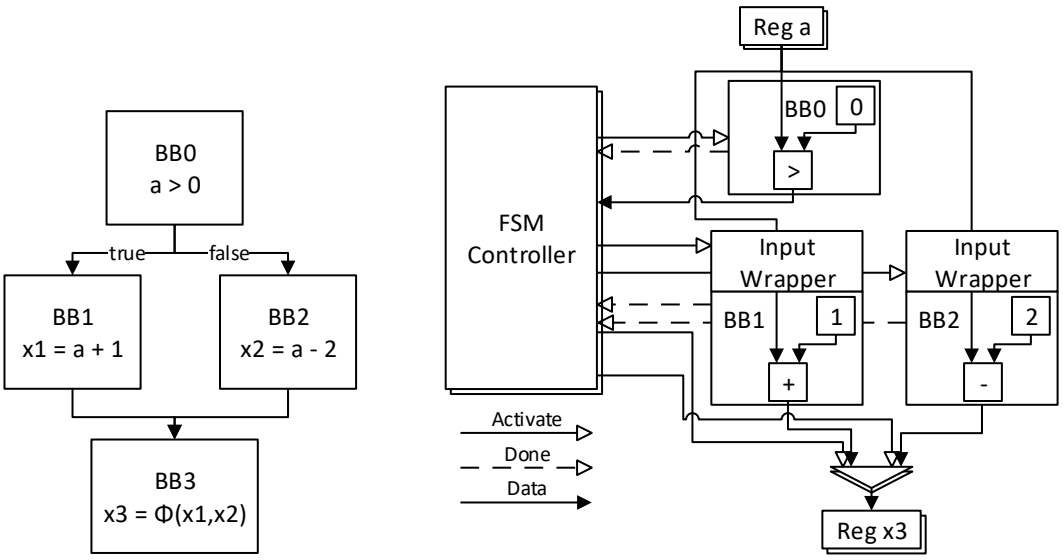
Figure 5.2: Basic block execution model

controller cannot achieve this because they do not communicate or synchronize themselves.

Without trying to synchronize the individual controllers, the solution was to add a wrapper which contains an arbiter that sequentially orders the parallel threads. Each clock cycle the arbiter selects a single thread to enter the block.

An example for resulting multi-threaded basic block implementation can be seen in Figure 5.3. In the future this wrapper could be removed by refinements of the backend. In general these refinements have to change the configuration of the TCS at the input registers of the loop implementing each basic block. Instead of always using the single write and read configuration (see Section 4.3.7.3) it would use the parallel writes and single read configuration.

Because only a single thread can finish a block at a time and this thread is then immediately moved to the next block, it is not necessary to select from multiple finished threads leaving a block. This logic is only necessary for the pipelined model where multiple VLOs, including nested loops, can be on a single stage.



(a) SSA-CFG **(b)** Basic Block Implementation

Figure 5.3: Multi-threaded basic block execution model (duplicated elements are shown as stacked)

(a) Problem: Two threads cannot enter block at the same time **(b)** Solution: Wrapper creates sequential thread ordering

Figure 5.4: Thread 1 (green) in BB1 and Thread 2 (red) in BB2 finished executing their block at the same time. Both threads want to move to BB3.

6 High Level Synthesis

This chapter will describe all the steps between the source code and the generated accelerator taken by the compiler. But before that it will be explained how the generated accelerator executes the application.

6.1 Compile Flow

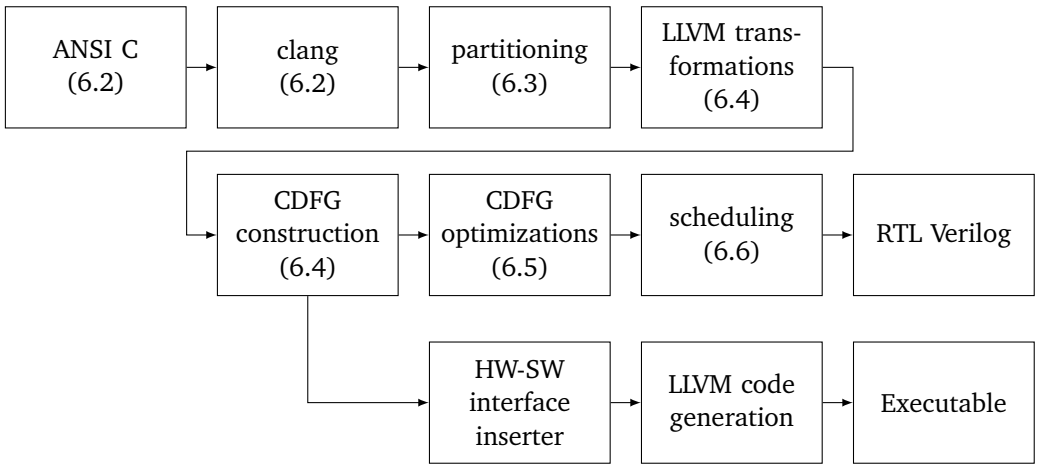


Figure 6.1: Compile flow

The compile flow in Figure 6.1 shows an overview of all steps taken by the compiler to generate the hardware accelerated application from the source code. As explained in Chapter 2, the compilation facilitates the LLVM framework for the front- and middle-end parts of the compile flow. Some existing parts had to be modified for a pragma driven automatic partitioning. In the figure, the parts which require additional explanations are marked by the number of the appropriate section.

The clang C/C++ front-end was modified to accept custom pragma directives described in Section 6.2. clang translates the annotated source of the applications to the LLVM IR. The IR is then partitioned into hardware and software parts according to the defined pragmas. The hardware part is extracted into a separate function and the hardware-software interface is inserted at the position of the extracted function. After this, a combination of LLVM optimization passes (see Section 6.4) are used to simplify and normalize to the hardware function. From

the LLVM IR, the hierarchical CDFG model is constructed (see Section 6.4). This model is then optimised for hardware synthesis using non-LLVM passes (see Section 6.5). The optimised CDFG is scheduled (see Section 6.6) using either ASAP or modulo scheduling[Rau94] and exported as RTL Verilog for logic synthesis.

In addition to this high-level hardware synthesis, the executable for the target system is generated by patching in the hardware/software interface operations using the LLVM code generation back-end.

6.2 C Input Annotation

The first step of the high level synthesis is to select which parts of the application should be implemented in hardware. For that, Nymble currently supports a semi-automatic partitioning of the application into hardware and software parts. The part of the application which is to be implemented as a hardware kernel must be selected in the C source by using the `#pragma HARDWARE on/off` and `#pragma SOFTWARE SERVICE on/off` statements.

In the example in Figure 6.2, the calculation in the method `foo` is selected as the hardware kernel. In high level synthesis, hardware kernels generally do not support user IO. However, Nymble provides a method to include any code that is not supported in hardware in the kernel by using hardware-to-software calls (see Section 3.1.6.3). With the `#pragma SOFTWARE SERVICE on/off`, the compiler is ordered to include such a hardware-to-software call into the hardware kernel. Nymble also automatically detects hardware unsupported code and places it into a hardware-to-software call. The developer can, however, decide to include more of the code around the unsupported code into a single call by manually placing the pragma.

6.3 Partitioning

For the automated hardware-software partitioning of the annotated source code, the clang front-end was modified to parse and insert the partition borders specified by the `#pragma HARDWARE on/off` and `#pragma SOFTWARE SERVICE on/off` statements. The pragmas are transformed into special marker instructions, placed at the region entry and all of its exits, while generating the LLVM-IR for the following newly written partitioning pass.

This pass traverses the CFG in reverse post-order and collects all BBs starting from the entry marker until reaching an exit marker. Markers which are not at the beginning of a BB invoke a split of the BB. Calls to other functions from within the marked blocks are either inlined or declared as a hardware function (see Section 3.3.6.2) or hardware-to-software call (see Section 3.1.6.3).

```
int foo(int* a[N]) {
  int i,sum;
  printf("executing␣foo\n");
  #pragma HARDWARE on
  sum = 0;
  for (i = 0; i < N; i++) {
    int *pointer = a[i];
    if(pointer == 0) {
      #pragma SOFTWARESERVICE on
      printf("NULL␣pointer␣encountered␣at␣i:%d\n",i);
      #pragma SOFTWARESERVICE off
      return ERR;
    }
    sum += *pointer;
  }
  #pragma HARDWARE off
  printf("finished␣foo\n");
  return sum;
}
```

Figure 6.2: C source annotated with partitioning pragmas

LLVM's CodeExtractor utility class is then used to extract these blocks into a new *accelerator function*. The SSA property in the LLVM-IR makes it easy to find values that cross the boundaries of the hardware part. Values originating from outside the accelerator are passed as arguments to the hardware function. Values that have uses outside of the accelerator become "out" arguments that point to a stack slot written inside of the hardware function.

6.4 CDFG Construction

After partitioning the CFG into software and hardware parts, the Nymble IR can be constructed for the hardware part of the application.

Before the CDFG is constructed out of the LLVM IR, the transformation passes shown in Table 6.1 are applied. These guarantee that the IR is in a normalized form for the CDFG construction. After the normalization, the hierarchically loop model is created by constructing a CDFG for each loop, starting with the most deeply nested loops.

The construction algorithm's description is split into three parts. The general construction (Section 6.4.1) of the data-flow in the CDFG and the more complicated construction of condition- (Section 6.4.2) and memory-dependencies (Section 6.4.3).

Name	Description
-simplifycfg	Removes dead or unnecessary basic blocks.
-lowerswitch	Transforms switch instructions to a sequence of branches.
-loop-simplify	Guarantees that natural loops have a preheader block, their header block dominates all loop exits, and they have exactly one backedge.
-sccp	Sparse conditional constant propagation.
-instcombine	Algebraic simplifications.
-dce	Dead code elimination.
-mergereturn	Transform function to have at most one return instruction.
-basicaa, -scev-aa	Alias information for load and store instructions based on program independent facts and scalar evolution analysis.
-loops	Natural loop detection. Identifies loop header, pre-header, exit, and ledge blocks.
-domtree	Build dominance relation for basic blocks.

Table 6.1: LLVM transforms/analysis passes used for normalization

6.4.1 General Construction

Each CDFG is constructed out of the SSA-CFG using a visitor-pattern. Because of the SSA form, each target value is created by only a single instruction in the CFG. When an instruction is added to the CDFG, the instruction is referenced in a symbol table indexed by the target value.

Starting with traversing all BBs for each instruction in the BB, it is first checked whether an instruction in the CDFG was already created by checking the symbol table for the target value. If it was not yet created, the visitor pattern is used to create the appropriate CDFG instruction. For instructions with parameters it is furthermore checked whether the source value of each parameter is already created. If it was not yet created, the visitor pattern is again used to create the appropriate source instruction. Then, the instructions are connected by adding a data-dependency edge from the source to the target instruction.

If an instruction is encountered, which has side-effects or where the result depends on actual control-flow in the CFG (ϕ, memory accesses) then, in addition to the control-dependencies, a control-dependency has to be added to the CDFG instruction. The condition construction is described in section 6.4.2.

After all CDFGs have been constructed, a second pass is done over them to add the dependencies between memory operations. The memory dependency construction is described in section 6.4.3.

6.4.2 Condition Construction

To understand the construction of the control-dependencies in the CDFG, it is necessary to go back to how control is handled in the CFG. In the CFG and its corresponding DFA execution model, the decision whether a BB and its operations are executed depends on all previous branch decisions in a iteration. But the result of all decisions is encoded in the state as the currently executed BB. So for each new branch instruction, it is enough to look at this single branch instruction as all previous branches are covered by the fact that this branch instructions are executed. If a previous branch decision had resulted in not executing this branch instruction, it would not have been executed at all.

As there is no such state in the pipeline model, it is always necessary to cover all branch instructions on the path from the start node to the target BB. The resulting control-dependency is a combination of the conditions from all encountered branch instructions.

Two kinds of control conditions in a CFG are used in this work, edge and block conditions. An *edge condition* is the condition under which an edge (A,B) is used to traverse the CFG from BB_A to BB_B. A *block condition* is the condition under which a specific BB is executed.

To construct the control-dependencies in the CDFG a combination of the control conditions is used. While the core of the construction algorithm is similar, the star point depends on type of instruction the control-dependency is created for.

For ϕ-instructions, which are transformed to ψ-instructions (see Section 2.1.5 for definition of ψ-instruction), the output depends on over which edge the BB is reached in the CFG. Because of that, the dependency construction starts with an edge condition.

For all other instructions, the control-dependency controls when they are executed. In the CFG all instructions inside a BB are executed when the control flow reaches that BB. Because of that, the dependency construction starts with a block condition.

The construction algorithms is split into the functions shown in Algorithm 1, Algorithm 2, and Algorithm 3. Starting from the input edge or block, the algorithm recursively travels backwards along the CFG. While it travels along the BBs and edges, a control-dependency is constructed incorporating all encountered conditions. The base case is when the loop header is reached. This means that this instruction only depends on the execution of the loop itself which is always fulfilled when the loop is executed. Whether the loop is executed is a control-dependency for loop instruction in the outer loop. In the control-dependency this is represented by a condition which is always fulfilled. It can be omitted if there are other conditions in the control-dependency.

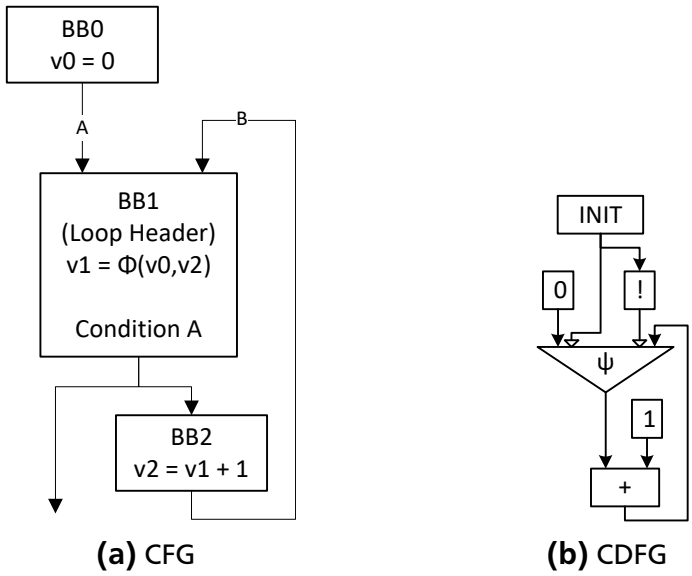


Figure 6.3: Special case: loop initialization condition construction

Only for the construction of the control-dependencies for ψ-instructions from ϕ-instructions in the loop header exists a special case. These ϕ-instructions exist for all variables which are possibly modified during the loop execution. This case is indicated by, that one of the two edges to the loop header (this exact number of edges is guaranteed by the earlier loop normalization pass in section 6.4) is coming from the outer loop. An example for that can be seen in Figure 6.3. In that case, for both edges to the loop header, their edge condition is constructed as a dependency from the `INIT` instruction. This special instruction is `true` only once per loop execution. This way the value coming from the outer loop is used only in the initial iteration. In all following iterations the value calculated in the previous iteration is transferred into the next iteration.

The cause for using the control-dependencies was already discussed in Section 2.1.5. What was not yet discussed is how the control-flow in the CFG affects the control-dependencies. This will be made clear with the following example. The control flow always moves along the CFG edges, starting at initial BB. If a branch is reached, the branch condition decides which edge is taken. To make it clear which impact these conditions have on the control-dependencies at each block or edge, the combined conditions are annotated to each block and edge.

The example in Figure 6.4 begins with first branch in `BB0`. Both outgoing edges are annotated with the value for condition A. Over the `false` edge, `BB1` is reached, which is thus also annotated with the condition A. Continuing with the branch in `BB1`, `BB2` and `BB3` are reached depending on condition B. From both BBs, `BB4` is reached, merging the two paths. At this point an important step is happening. Note how both edges from `BB2` and `BB3` to `BB4` are annotated with different

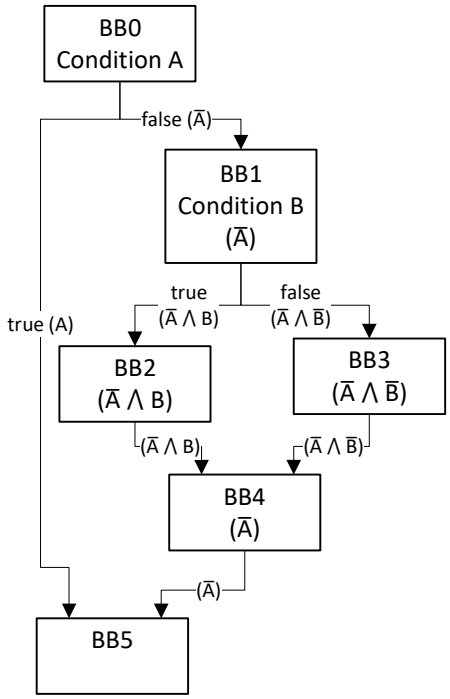


Figure 6.4: Condition example

variants for condition B. If these conditions would be combined, an unnecessary condition $(\bar{A} \wedge B) \vee (\bar{A} \wedge \bar{B})$ is created as the dependency on condition B can be completely removed. The cause for that is that the paths started by condition B in `BB1` merge at this point. While not directly harmful, such unnecessary conditions in the dependency can lead reduced performance of the created data-path. The same step is also happening at `BB5`, which is the merge for condition A in `BB0`.

In the construction algorithm, this case is detected by using the dominator tree for the CFG. As the algorithm moves backwards along the CFG, the check is to see if its possible to skip some part of the CFG. The part that can be skipped is defined by the branch and corresponding merge point in the CFG. The earliest branch reached going backwards along the CFG for a given BB X, is the immediate dominator (IDOM) of BB X. If the tested BB X is also the merge point for that branch, it is the immediate post-dominator (IPDOM) of the IDOM of X. It is then possible to skip ahead to the IDOM BB (shown in Algorithm 2).

But there is a case where there is such a IDOM IPDOM relationship which does not correctly identify the branch merge relationship. This case is shown in figure 6.5. It can be easily seen that `BB1` is not a merge point for condition A. If the previous tests are applied it can be seen that the IDOM of `BB1` is `BB0`. But also that `BB1` is an IPDOM of `BB0`. This IPDOM relationship is not so obvious at first because it uses the back-edge from `BB2` to `BB0`. If this back-edge is used it becomes clear that exit node is only reached going through `BB1`. But as this crosses the border

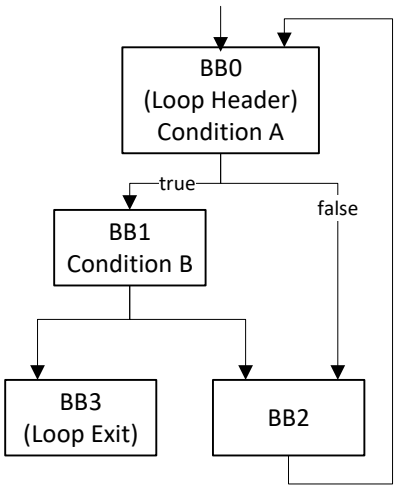


Figure 6.5: Special case for loops

between loop iterations through the back-edge, and the construction algorithm is only for intra-loop dependencies, it is necessary to detect and recover from this case. To detect this back-edge, it is checked if at least one successor of the IDOM of BB X is post dominated by IDOM of BB X. In the example the IDOM of `BB1` is `BB0`. While `BB1` as an successor of `BB0` is not post dominated by `BB0`, the second successor `BB2` is post dominated by `BB0`. Thus the algorithm decides that it cannot skip ahead from `BB1` to `BB0`. The details for this decision is shown in Algorithm 3.

```
input: Edge BB_pred → BB_x
if BB_pred outside current loop then          // Edge from outside of current loop
    return condition_init                     // -> Use init condition from current loop
else
    condition_pred = block condition(BB_pred)
    if number of successors(BB_pred) = 1 then
        return condition_pred
    end
    switch Type of termination statement do   // if or switch statement
        case branch do
            if BB_x = target_true then        // true or false edge
                condition_edge = condition_branch
            else
                condition_edge = not(condition_branch)
            end
        endcase
        case switch do
            conditions_targets = ∅
            conditions_all = ∅
            // collect conditions of all switch branches and create
            // set of conditions only reaching the target block
            foreach (BB_target, condition_target) ∈ targets do
                conditions_all = conditions_all ∪ condition_target if
                 BB_target = BB_x then
                    conditions_targets = conditions_targets ∪ condition_target
                end
            end
            if BB_x == BB_defaulttarget then
                // Block is reached by default case
                // no case conditions must be fulfilled
                condition_edge = not(or(conditions_all))
            else
                // Block is reached by some cases other than default
                condition_edge = or(condition_targets)
            end
        endcase
    end
    return and(condition_pred, condition_edge)
end
```

Algorithm 1: Edge condition

```
input: Basic block BB_x
if can use condition of IDOM(BB_x) then          // Skip intermediate blocks
    return block condition(IDOM(BB_x))
else
    // The block can be reached by any incoming edge
    // One of the edge conditions must fulfilled
    conditions_pred = ∅
    foreach BB_pred ∈ predecessor(BB_x) do
        conditions_pred = conditions_pred ∪ edge condition(BB_pred, BB_x)
    end
    return OR(conditions_pred)
end
```

Algorithm 2: Block condition

```
input: Basic block BB_x
if not BB_x post dominates IDOM(BB_x) then
    // Cannot skip if not all edges from the immediate dominator
    // reach this block
    return false
end
foreach BB_succ ∈ successor(IDOM(BB_x)) do
    // Cannot skip from a loop exit to loop header
    // See Figure 6.5, BB3 and BB0
    if IDOM(BB_x) post dominates BB_succ then
        return false
    end
end
return true
```

Algorithm 3: Can use condition of IDOM(BB_x)

6.4.3 Memory Dependency Graph Construction

As the model uses nested CDFGs, it is necessary to construct a MDG for each CDFG. First the MDG for the CFG is created by using the Steensgard alias analysis implementation provided by LLVM.

The MDG for the CFG is then mapped to the MDGs of each CDFG. In general each CDFG MDG is a subset of the CFG MDG, where only the edges between the instructions that are in the same CDFG are in the corresponding MDG.

In addition to these dependencies a number of other cases require additional dependencies to be added to the CDFG MDGs. The biggest part of these additional dependencies are made up by dependencies between instructions in different CDFGs.

Depending on the configuration of the execution model (static or dynamic II) the constructions of these additional dependencies differs. The static II model re-uses the same ports by guaranteeing that only one instruction is accessing a port at a time. All accesses to the same port have to be sequentialized through the dependencies, even across nested CDFGs. This means that it is not only important which instructions are accessing the same memory location but also which are using the same memory port.

The dynamic II model re-uses the same port with help of a dynamic global arbiter. The requirement for that arbiter is only that the dependencies from the basic CFG MDG have to be fulfilled, including dependencies across nested CDFGs.

6.4.4 CDFG Construction Example

As mentioned previously, Nymble uses a hierarchical CDFG model to represent the application.

Nymble traverses BBs of the CFG in post-order to build the CDFG. If the necessary operand of an operation has not been constructed yet, it will be constructed before the operation itself. Because of the post-order this happens only for back-edges and constants. Already constructed operations, representing a SSA value, are entered into a symbol table. Because the CFG is in SSA form, each value is represented by exactly one operation. Each operation has a type which specifies the bit-size and whether it is an integer, fixed-point or floating-point value. Integer values are further distinguished between signed and unsigned values.

The BB names are in the format as they are created by LLVM, so the format is BB(name). LLVM uses these names in the code generation for jump labels. Values are labelled in the format `%value.version`.

In the following example (Figures 6.8 and 6.9) for the construction of the CDFG for the application in Figure 6.6 (CFG in Figure 6.7) the following colours will be used to highlight parts of the CDFG. Green will show the main calculation of

```
int func(int op, char *X, int N) {
  int j;
#pragma HARDWARE on
  for (j = 0; j < N; j++) {
    char tmp = X[j];
    if (op)
      tmp++;
    else
      tmp--;
    X[j] = tmp;
  }
#pragma HARDWARE off
  return j;
}
```

Figure 6.6: Sample function with partitioning directives

the application, blue will show the address calculation for the memory accesses and red will show calculation of the control dependencies. Specific points in the figures will be highlighted in the text and figure with ①, ②, etc.

In Figure 6.9a the construction was interrupted after creating the operations for BB(`for.cond`). The `add` instruction ③ together with the ψ instruction ② implement the incrementation of the loop counter. This corresponds to the statements `%j.0 = phi i32 [ 0, %entry ], [ %inc2, %for.inc ]` for the ψ and `%inc2 = add nsw i32 %j.0, 1` for the `add`. The ψ here is used for a loop carried data dependency. This means that only for the first iteration of the loop the initial value, here a 0, is used. For all other iterations the value calculated inside the loop, here `inc2`, is used.

One important point that can be seen is that the comparison statement `%cmp = icmp slt i32 %j.0, %N` is treated as a normal operation ① which just calculates a value (here `%cmp`).

The next interruption shown in Figure 6.9b is after creating the operations for BB(`for.body`). The left `add` operation ⑤ was created for the statement `%arrayidx1 = getelementptr inbounds i8* %X, i32 %j.0`. Its purpose is to add the loop counter `j.0` to the base address `X`. Note that as the base address `X` comes from outside the loop a special transfer operation ④ for it has been created. The control dependency for the `load` ⑥ is constructed with the algorithm explained in Section 6.4.2.

The state of the CDFG after the BBs BB(`if.then`), BB(`if.else`) and BB(`if.end`) is shown in Figure 6.9c. Here the operations for the increase or decrease of the value `%0`, which was loaded from memory, were constructed. Note that the loop counter was omitted.

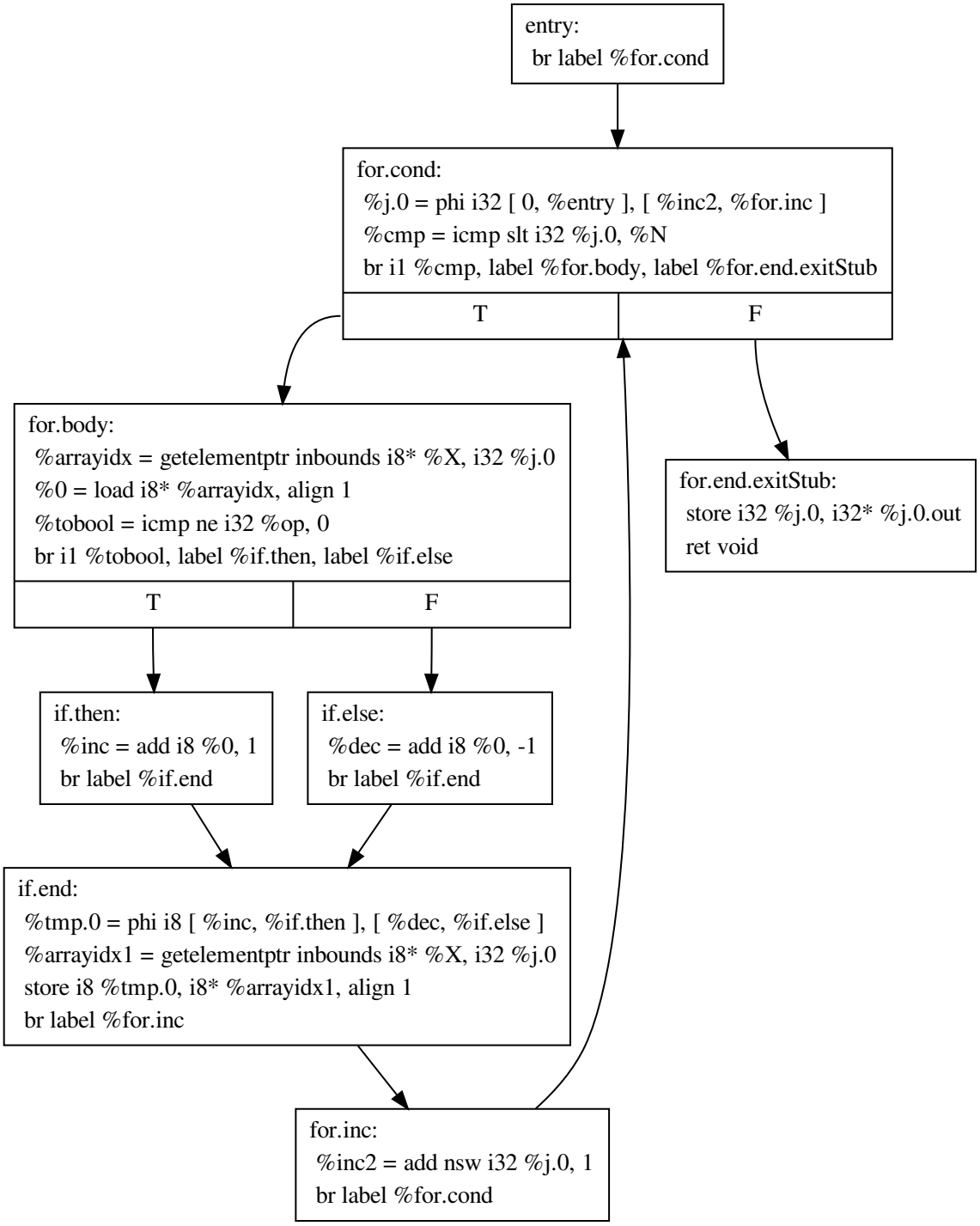


Figure 6.7: CFG of the hardware function from Figure 6.6

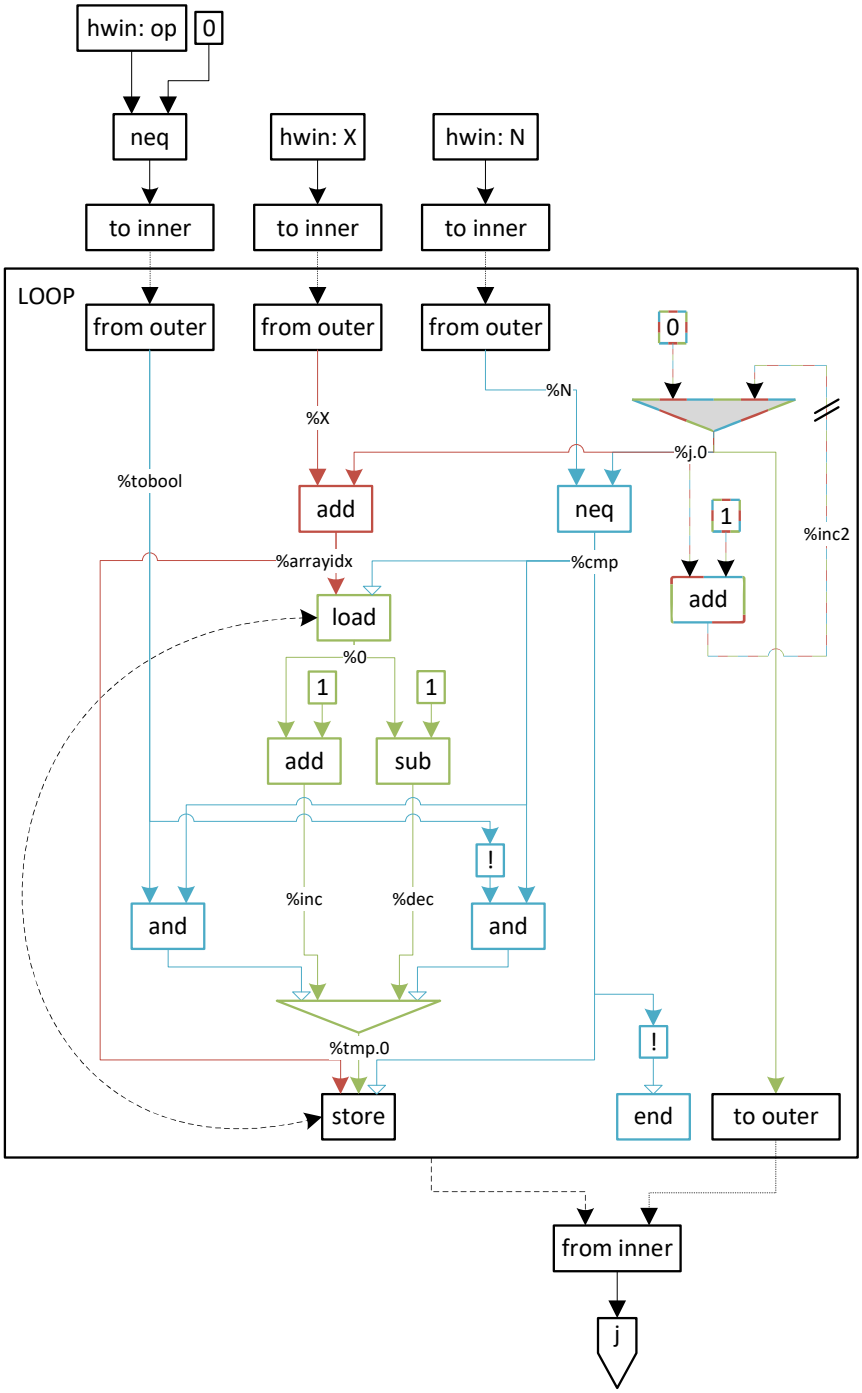


Figure 6.8: CDFG of the hardware function from Figure 6.6

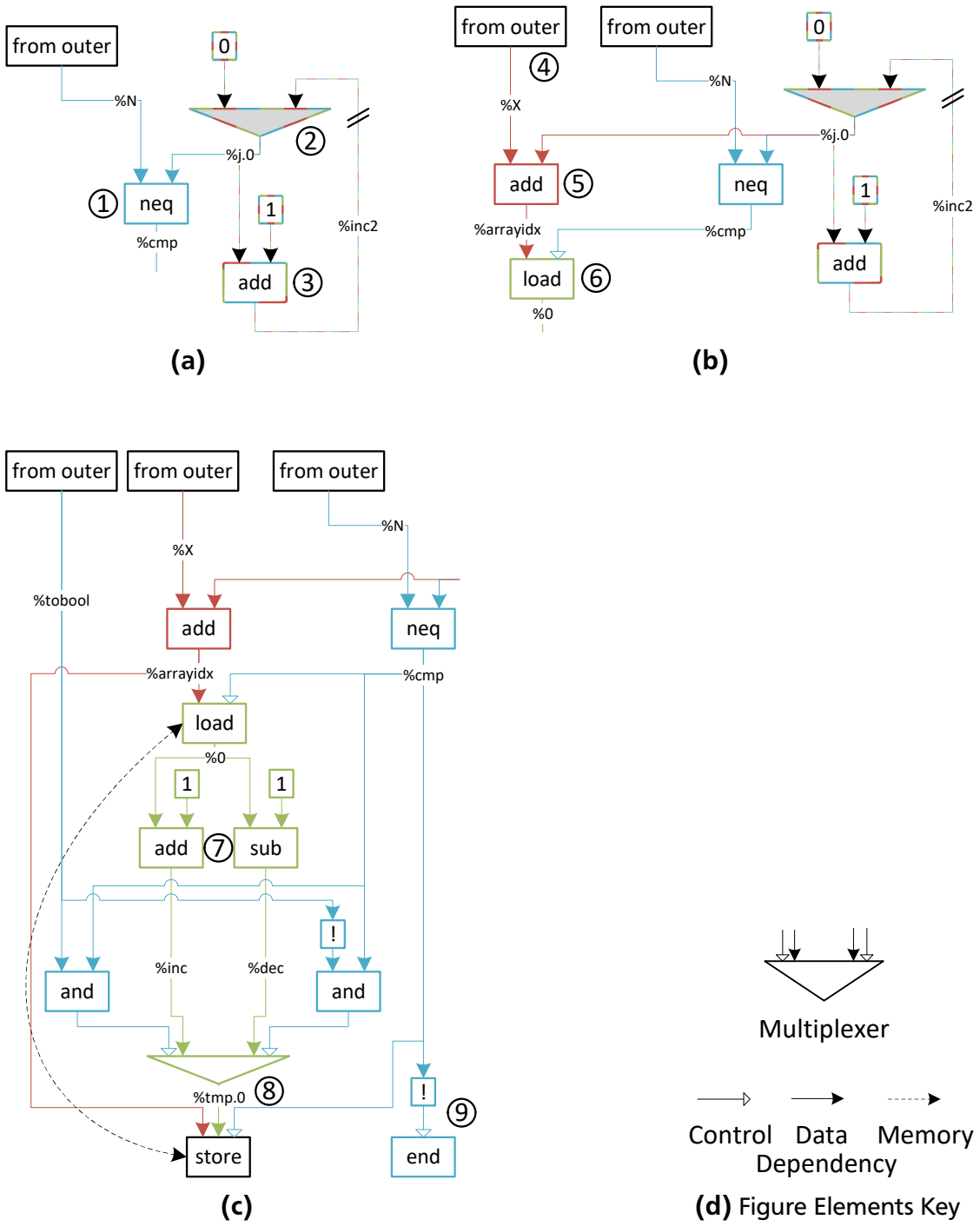


Figure 6.9: CDFG example construction steps

In the example, both values `inc` and `dec` ⑦ are always calculated. The ϕ instruction `%tmp.0 = phi i8 [ %inc, %if.then ], [ %dec, %if.else ]` is transformed to a ψ instruction ⑧. The control dependency for each input of the multiplexer is constructed with the algorithm explained in Section 6.4.2.

Also, the `store` for the statement `store i8 %tmp.0, i8* %arrayidx1, align 1` has been created at this step. Like the ψ and `load` it has a control dependency. Additionally, the memory dependency between the load and store is added.

The termination condition of the loop is added to the special operation `end` ⑨. This condition is calculated by `or`-combining the conditions of all edges leaving the loop. Here this condition is the edge from `BB(for.cond)` to `BB(for.end.exitStub)` in Figure 6.7.

Finally, as the loop has been constructed, the surrounding loop can be constructed as well. In the example this is just the function body. To exchange values between the function body and the loop, the necessary transfer operations are inserted. As this function body is the top level function of the example, the construction of the hierarchical CDFG has been finished.

6.5 CDFG Optimizations

After the hierarchical CDFGs for the application are constructed, a number of execution model independent optimizations are applied to reduce the size or improve the throughput of the generated accelerator.

6.5.1 Chaining

During the scheduling, each operation is assigned to a stage. This assignment can have a direct impact on the performance by the resulting II of each loop's pipeline. As written earlier, operations can be executed combinational as long as their combined delay in a stage is not higher than the time available. This time is defined by clock frequency, as each stage is to be executed in a single clock cycle.

For the combinational chaining, the delay of each single cycle operation was analysed and entered into a table. The compiler now tries to chain as many operations as possible to minimize the number of stages. MCOs or VLOs constitute a natural border for the chains as they are never included in a chain.

The compiler greedily takes any operation whose predecessors were already visited and tries to chain as many operations as possible to it. This then continues until all operations are visited.

6.5.2 Constant Propagation

In addition to the constant propagation provided by LLVM, the Nymble has to apply an additional constant propagation pass on the CDFGs. This is necessary because the CDFG construction adds additional operations, especially for the control-dependencies. So like a constant propagation on the CFG, the compiler tries to statically evaluate the conditions during compile time.

6.6 Scheduling

As both the static and dynamic II model require the assignment of each operation to a stage, Nymble provides a number of scheduling algorithms. The simple greedy algorithm is mostly used for initial testing of new features as it generates a schedule usually much faster. The other more sophisticated algorithms are used to achieve smaller IIs.

This section will shortly explain the provided scheduling algorithms. All schedulers support limited resources which is generally necessary for the static multiplexing of memory accesses. The dynamic model handles the resource sharing dynamically and uses scheduling only for its stage based execution.

6.6.1 ASAP Scheduling

The simplest scheduling algorithm provided is a greedy "As Soon As Possible" scheduler. Each operation is scheduled at the earliest stage, when all other operations the operation has dependencies on are already scheduled. This only takes intra-iteration dependencies into account and the schedule is corrected afterwards by moving all operations violating them into a later schedule. Using the ASAP scheduler can lead to worse II than modulo scheduling algorithm, resulting in worse performance of the generated accelerator. To comply with the resource limits, the schedule moves the currently scheduled operation to later stages until the resource limits are not violated.

6.6.2 Modulo Scheduling

The second scheduling algorithm uses modulo scheduling [Rau94] to achieve possible better II than the simple ASAP scheduler. For a set II the algorithm tries to iteratively find a valid schedule. From an initial, not necessary valid schedule, it tries to re-schedule all conflicting operations (and dependant operations) using a priority scheme. If the algorithm can not find a valid solution after set amount of tries, the II is increased and restarted with that II.

7 Target Systems

This chapter will describe the details of each target system currently supported by Nymble and the generated accelerators. While the overall behaviour is similar on all systems, it differs in the following points: Hardware-Software interface including shared memory organisation, cached memory system and hardware invocation protocol.

7.1 ACE M5

The initial target system was the ACE M5 adaptive computing system (shown in Figure 7.1), which was used in [Hut+10b; THK11b; Thi+11; THK11a]. Hardware-wise, the ACE M5 consists of a Xilinx ML507 Virtex-5 FX development board. The on-chip PowerPC 440 processor acts as the CPU component of the heterogeneous system while the remainder of the Virtex-5 FPGA fabric are used for the accelerator. The main memory shared between CPU and accelerator is implemented off-chip as DDR2-SDRAM DIMMs. However, a number of extensions have been made over the original Xilinx-provided environment.

First, memory accesses are performed using the MARC2 memory access system [LWK11], allowing the accelerator direct access to the DDR2-SDRAM memory controller without having to go over the comparatively slow PLB bus. MARC2 provides each memory operation in the data-path with a dedicated cache port. The system can be configured by indicating which accesses occur to non-overlapping memory regions (determined by Alias/Points-To analysis in the Nymble compiler) to generate an application-specific sparse cache coherency communication network. Each coherency cluster may have an arbitrary number of read ports, but only a single write port. If more writes occur to address ranges potentially overlapping those of the read ports, multiple writes need to be sequentialized and issued through the single write port. An arbitrary number of parallel writes may be performed to non-overlapping ranges. In that case, the writes can be assigned to different coherency clusters. This arrangement allows the issuing of multiple memory operations in parallel per cycle.

Second, the ACE M5 platform runs under a heavily modified full-scale version of the Linux operating system that allows fast accelerator-CPU signaling (up to 23x faster than even a kernel with real-time patches) as well as the use of virtual memory by the accelerator using the AISLE technique [LK10]. The latter is important for allowing the CPU and accelerator to freely pass pointers across the hardware-

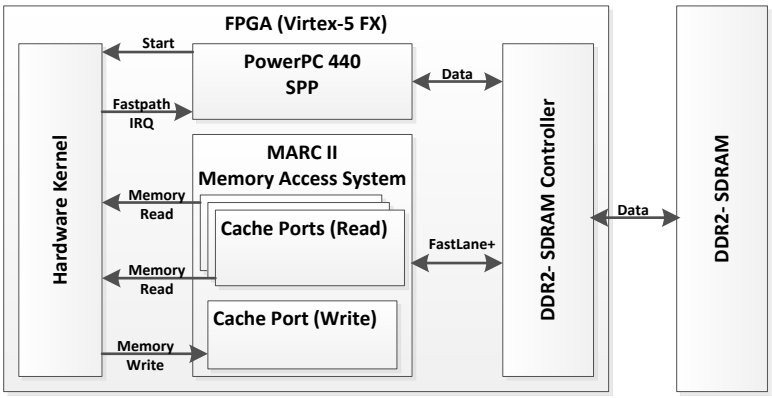


Figure 7.1: ACE M5 system architecture

software boundary, a crucial capability when compiling from a pointer-intensive language such as C.

On the ACE M5 platform, hardware communication registers are memory mapped through the operating system modifications. The shared memory is placed in the RAM directly on the board. Both CPU and FPGA use the same physical address space to access the memory.

While both the CPU and FPGA use the same physical address space to access the memory over the same memory controller, the cache in the CPU and FPGA application are not kept in coherency. So upon a context switch between the software and hardware, it is necessary to flush and invalidate the appropriate cache lines in both caches.

The hardware invocation protocol on the ML507 platform is identical to the one shown in Section 3.1.6.4 because it is the original platform which required the additional flush and invalidate steps.

7.2 DINI

As the FPGA resources of the initial system proved to be inadequate, a second system was developed. As in the ACE M5 system, the software part of the application is executed a standard Central Processing Unit (CPU) (Intel i7-3770K). However, the connection of the shared memory and accelerator is quite different here. In ACE M5 both the FPGA and the CPU (which are even integrated into a single chip) are connected to the same DDR2 memory. The DINI system (Figure 7.2), however, uses a standard PC as host which is connected via PCIe to a DNV7F1A Virtex-7 development board as the accelerator. In addition to the Xilinx Virtex 7 VX690T FPGA, this board contains its own set of DDR3 memory. To share this memory between the software and hardware parts of the application, some modifications in

the operating system were necessary. Before explaining these changes, the system architecture is shown.

The connection between the host PC and the accelerator board uses a PCIe 2.0 interface with four lanes. On the board the interface connects to the manufacturer provided *config* FPGA. The config FPGA then communicates with *user* FPGA. The DDR3 memory controller is implemented in the user FPGA together with the application logic. This is then the memory which is shared between the hardware and software parts through the operating system modifications.

Herein lies the biggest difference between ACE M5 and this system. In contrast to ACE M5, both the host PC's CPU and the hardware kernel access the shared memory over the same cached memory system. Upon a context switch between the hardware and software parts, this cache does not have to be flushed unlike in ACE M5. However the CPU cache still must be flushed or invalidated.

Like with the ACE M5 system, the shared memory is accessed in the whole application (hardware and software parts) with virtual addresses. This means that pointers can be used and shared across the hardware-software "border". In contrary to ACE M5, where the shared memory can be directly accessed (because it is in the main memory), the shared memory located on the accelerator board has to be mapped into the physical address space of the host PC. This uses the designated methods for PCIe memory mapping provided by the standard linux kernel. After this mapping, the CPU can address the shared memory as in ACE M5. Using the same PCIe memory mapping, the register to control and communicate with the accelerator are also mapped into the physical address space (which is also done in ACE M5). Currently only 32MB are used from the shared memory because of limits imposed by the config FPGA.

The operating system recognises that an application is a hardware-software application by means of a special flag in the executable. Additionally, the name of the executable is prefixed with the first slots and the number of slots to be used. The operating system then allocates the requested slots in the PCIe mapped shared memory. All segments, besides the code segment, are then loaded into that memory region. Also, all additional memory allocated dynamically during runtime (malloc) is only allocated in the initially requested slots. Upon termination of the application the allocated slots are freed for reuse. Currently, the 32mb are split into 8 slots with 4mb each.

For multi-threaded execution each thread's application uses its own independent and continuous set of slots. This in turn means that the DINI system currently only supports 8 parallel executions of a multi-threaded application. This is done to allow the total separation of the memory of each thread and thus avoid thrashing or other memory dependency issues in the memory system between different threads.

In reality, however, it seems that all current Intel CPUs have a problem with caching PCIe-mapped memory. This leads to instantaneous crashes when access-

ing the shared memory on the accelerator. Because of this, the cache has to be disabled when accessing the shared memory. In turn this makes the software part of the application extremely slow.

As the focus of this work is the multi-threaded execution model of the hardware part of the application, the disabled cache has been accepted because of limited resources to integrate another target system. To prove that the kernels generated by Nymble work in a real system, it was sufficient to have a working system where the generated kernels can executed together with the memory system for the shared memory.

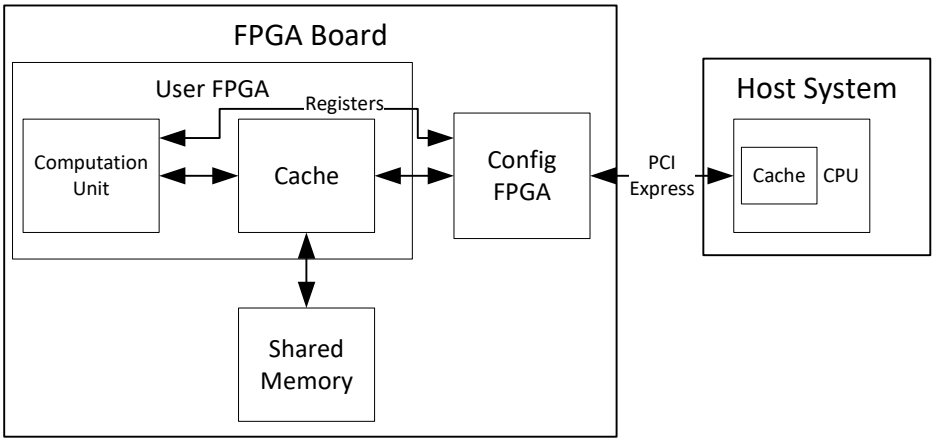


Figure 7.2: DINI system architecture

7.3 Convey

As an additional target system the Convey HC1ex platform [Con10] was examined because of availability. The system (shown in Figure 7.3) consists of a host pc and the attached Convey coprocessor. The host system is build as a multiprocessor system using a standard CPU. Interestingly, the coprocessor is integrated into the host system as one of the CPUs. The coprocessor board is placed on top of the host and is connected directly to one of CPU sockets. The board contains multiple Xilinx Virtex 6 LX760 FPGAs which are called Application Engine (AE). (Only one is currently used by this work). Each of these AEs is connected to a shared memory system which can also access the host systems memory over the host-interface. In the same manner the host system can access the memory on the coprocessor board.

However, the memory system provided for the FPGA (Xilinx Virtex 6 LX760) on the accelerator board is targeted to streaming memory accesses. The memory system provides a high throughput with a relatively high latency (90 clock cycles) for individual memory accesses. As the kernels currently generated by Nymble do not use streaming memory accesses, this means that while the kernels can be exe-

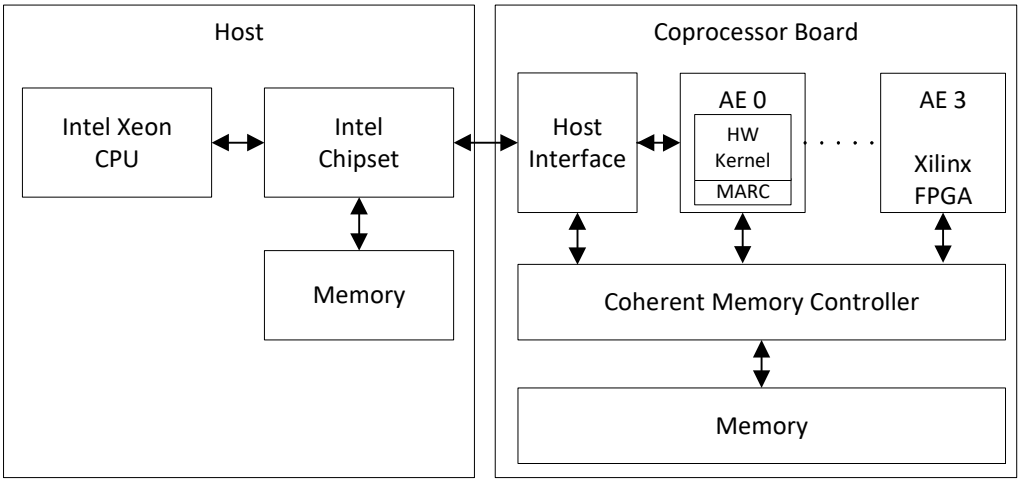


Figure 7.3: Convey HC1ex system architecture

cuted on the Convey system, the performance is handicapped by this architecture mismatch.

The host system uses a Linux operating system provided by Convey. The communication with the coprocessor utilizes an extended instruction set with additional communication instructions. The communication protocol for Nymble is identical to the one shown in Section 3.1.6.4. While the Convey memory system provides coherent cache from all ports flushes and invalidates are not necessary. However, because MARC2 is used as a cache on top of the Convey memory system, flushes and invalidates of MARC2 are required.

These issues with the streaming accesses and necessary cache flush/invalidate, despite using a coherent shared memory, result from the mismatch of the expected and actual memory architecture. Some problems could be alleviated by supporting streaming memory accesses with Nymble. With the pipelined model it would be possible that multiple iterations of the same thread wait in the input queue of memory operations. An improved interface would allow the "streaming" of these accesses to the Convey memory system reducing the overall latency. These improvements however are out of the scope of this work.

8 Related Work

In this chapter a number of compiler frameworks will be discussed. After a general discussion, selected academic high-level-synthesis tools will be explained in greater detail and their difference to Nymble examined.

8.1 General

High-level synthesis tools translating different subsets of C into synthesizable HDL code are under active development from many commercial vendors and academic groups alike. Commercial tools include Xilinx Vivado HLS [Xil14], Y Explorations eXCite [Y E], and Synopsis Synphony C Compiler [Syn11]. These tools, however, do not perform co-compilation into hybrid hardware/software-executables, which is still the domain of a small number of academic projects such as LegUp [Can+13a], ROCCC [Vil+10], Comrade [GL11], Bambu [PF12] and DWARV [Nan+12]. The topic of exploiting multi-threaded execution in the generated hardware is even more rarely addressed.

In [HN13], the CHAT compiler is introduced as a variant of ROCCC capable of generating multi-threaded accelerators that allow for a very quick context switch to alleviate the impact of memory latencies. Like ROCCC, the CHAT compiler is focused on generating hardware for highly specialized classes of input programs, such as sparse matrix multiplication. According to the authors, CHAT can translate only regular **`for`**-loops with a single index variable. A similar approach with multiple processing elements is used by Bambu [Min+16].

In [Tan+14] Tan et.al. presented a multi-threading model very similar to that of this work. Their work is based on a commercial HLS tool which itself is based on LLVM. They are using a context buffer for switching and reordering threads similar to the TCS and HTS unit combined. Their local memory based approach for the context buffer is similar to the TCS (Nymble uses registers, local memory or distributed ram for queues depending on their size). Compared to Nymble, they are providing a method to optimise the scheduling to reduce the number of the context buffers. With regard to the multi-threading model of Nymble, this means to reduce the number of multi-threaded stages by moving VLOs which are only a few stages apart to the same stage if possible. In addition, they use a heuristic to reduce the size of the context buffers itself. This heuristic tries to move non VLOs before or after multi-threading stages in such a way that the number of data going through the multi-threaded stage is reduced. This then reduces the bit-size of the

data to be stored in the TCS. This approach is a good idea to also be included in this work, possibly allowing an increase of the number of threads in the system.

Nymble, Tan et. al. [Tan+14] and CHAT [HN13] share the general idea that is beneficial to hide memory access latencies by switching execution to another ready thread. While Nymble and Tan et. el. use a similar context buffer model applicable to a wide range of applications, CHAT's multi-threading can only be applied to a small domain of problems because the for-loops used as the multi-threaded kernels must follow a strict coding scheme where only the innermost loop can contain any logic besides the loop variable.

In contrast, LegUp pursues a different approach more similar to software multi-threading [CBA13]. LegUp accepts a parallel program that uses the *pthreads* and *OpenMPI* APIs and generates a dedicated hardware accelerator instance for each (software) thread or for each parallel loop respectively. This is fundamentally different from Nymble which aims to increase the utilization of a *single* accelerator instance by extending it for simultaneous multi-threaded execution and allowing the processing of data from parallel threads.

As an example for another completely different approach to multi-threaded accelerators, Convey Computer recently added support for a concept called *Hybrid Threading* (HT) to the tool chain for their FPGA-accelerated computing systems [Con14]. The HT flow accepts a low-level description encapsulated in idiomatic C (basically an FSM, with each state representing a clock cycle, extended with message-based I/O) for efficiently describing computation, but without support for pointers or variable-bound/non-unit stride loops. These descriptions are then compiled into synthesizable HDL and linked to a vendor-supplied HW/SW framework that allows the starting of threads on the hardware accelerators and provides the context switching mechanism. Thread switching requires a single clock cycle and is used to effectively hide memory latencies. Despite being limited to an idiomatic programming style, the abstraction level of the HT C code is significantly higher than low-level HDL programming, with the actual multi-threading hardware being added automatically by the tools. The main difference between HT and Nymble is that the latter accepts true untimed programs, while HT relies on a manually scheduled/chained program with explicit message-based communication to host and memories.

8.2 Compiler Frameworks

Before discussing some of the High-Level-Synthesis (HLS) tools in more detail, the underlying compiler frameworks used by these tools will be explained briefly.

8.2.1 GCC

GCC [SE16] is the GNU Compiler Collection. It is an open-source compiler framework with support for many languages and target systems. GCC's middle end is based on the GIMPLE IR which is a textual representation of the application which only uses Three Address Code (TAC) instructions. Also, the control flow is represented by a combination of conditional statements and *goto* instructions. Figure 8.1b shows an example of GIMPLE with the corresponding C statements from the source code in Figure 8.1a.

GCC is used by some academic HLS tools (Section 8.3.2) as their front-end and middle-end (for target system independent optimizations and static code analysis). Their own back-end is then used to transform the GIMPLE IR into a hardware kernel.

```
int func(int op, char *X, int N) {
  int j;
  for (j = 0; j < N; j++) {
    char tmp = X[j];
    if (op)
      tmp++;
    else
      tmp--;
    X[j] = tmp;
  }
  return j;
}
```

(a) C Code

```
char tmp = X[j];      { j.0 = (sizetype) j;
                        D.1492 = X + j.0;
                        tmp = *D.1492;
if (op) {...} else {...}  { if (op != 0) goto <D.1493>;
                            else goto <D.1494>;
                          <D.1493>:
tmp++;                { tmp.1 = tmp;
                        tmp.2 = (unsigned char) tmp.1;
                        D.1497 = tmp.2 + 1;
                        tmp = (char) D.1497;
                      goto <D.1498>;
                      <D.1494>:
tmp--;                { tmp.3 = tmp;
                        tmp.4 = (unsigned char) tmp.3;
                        D.1501 = tmp.4 + 255;
                        tmp = (char) D.1501;
                      <D.1498>:
X[j] = tmp;           { j.5 = (sizetype) j;
                        D.1503 = X + j.5;
                        *D.1503 = tmp;
```

(b) C code with corresponding GIMPLE statements for the loop body

Figure 8.1: GCC IR example

8.2.2 LLVM

LLVM [LA04] is an open-source compiler framework. The name LLVM came from the acronym of the name "Low Level Virtual Machine", but today it stands for the entire project. The virtual machine is no longer the focus but only a part of the LLVM project. The development started as a research project at the University of Illinois. Today LLVM encompasses a wide number of projects where many of them are used in productive commercial and open-source projects. It is also widely spread in academic use.

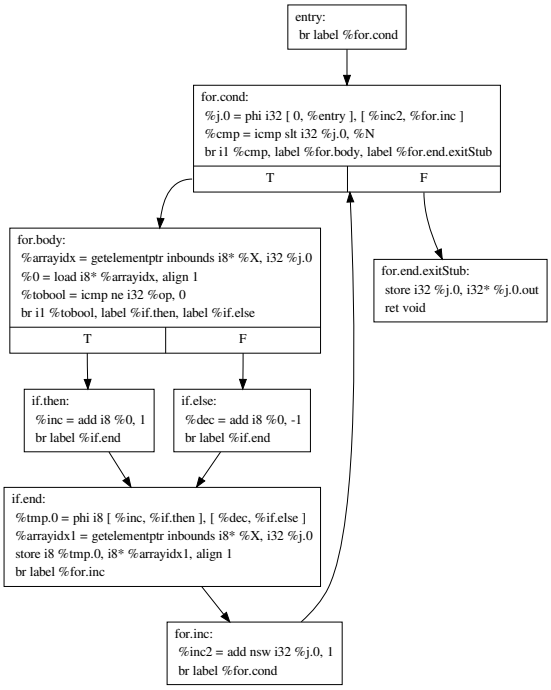
LLVM provides a complete compilation tool-chain, split into different components. Each component was designed to be compatible with existing tools generally used in Linux.

LLVM uses a CFG in SSA form with low level statements for its IR. Figure 8.2c shows which C statements from the example in Figure 8.2a correspond to which LLVM statements. Note that control statements like `if ... else ...` or loops are transformed into basic branch statements like `br`. Figure 8.2b shows the IR as a CFG.

LLVM also provides a wide array of target system independent optimizations. For Nymble the most important optimizations were shown in Section 6.4. For these optimizations and additional target specific purposes LLVM also provides static code analysis methods. For the compilation for heterogeneous systems, the application can easily be partitioned with LLVM by using built-in code extraction methods.

```
int func(int op, char *X, int N) {
  int j;
  for (j = 0; j < N; j++) {
    char tmp = X[j];
    if (op)
      tmp++;
    else
      tmp--;
    X[j] = tmp;
  }
  return j;
}
```

(a) C Code



(b) IR

```
char tmp = X[j];      { arrayidx = getelementptr inbounds i8* %X, i32 %j.0
                      { %0 = load i8* %arrayidx, align 1
if (op) {...} else {...}  { %tobool = icmp ne i32 %op, 0
                          { br i1 %tobool, label %if.then, label %if.else
                          if.then:
tmp++;                { %inc = add i8 %0, 1
                      br label %if.end
                      if.else:
tmp--;                { %dec = add i8 %0, -1
                      br label %if.end
                      if.end:
                      %tmp.0 = phi i8 [ %inc, %if.then ], [ %dec, %if.else ]
X[j] = tmp;           { %arrayidx1 = getelementptr inbounds i8* %X, i32 %j.0
                      { store i8 %tmp.0, i8* %arrayidx1, align 1
```

(c) C Code with corresponding LLVM statements for the loop body

Figure 8.2: LLVM IR example

8.2.3 CoSy

CoSy [ACE03] is a commercial compiler framework which provides support for C and C++ and was initially released in 1994. CoSy has been used for a wide array of target architectures. It provides many analyses optimizations to improve the resulting applications. Because CoSy is closed source, no example for its basic IR can be shown. CoSy is used by some academic HLS tools (Section 8.3.3) and other compilers.

8.3 High Level Synthesis Compilers

In this section, a number of HLS compilers is presented with details of their execution model, target platform and memory model. LegUp, Bambu and DWARV were chosen because they are current compilers used in recent research. Besides LegUp and DWARV, ROCCC with its extension CHAT was chosen as an additional compiler which can generate multi-threaded kernels.

8.3.1 LegUp

LegUp [Can+13a] is a HLS compiler based on LLVM, first released in 2011. While LegUp can also target heterogeneous systems consisting of a CPU and accelerator, most of its published evaluation was done on pure hardware-only system. The current implementation is for an Altera DE2 board with a MIPS softcore processor and accelerator connected by an Altera Avalon interconnect. In the heterogeneous system, both the CPU and the accelerator use an on-FPGA data-cache connected to of chip-memory, as shown as in Figure 8.3. Currently, dynamic memory allocation (malloc) is not supported even in the software part of the heterogeneous system (tested in public LegUp 4.0 release). However, pointer based data-structures are supported.

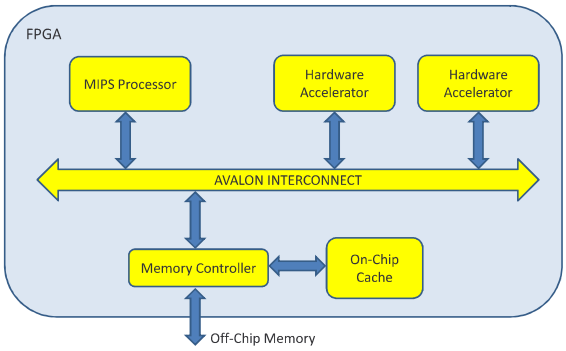


Figure 8.3: LegUp target system architecture (taken from [Can+13b])

LegUp statically allocates all memory to either global or local memory. While global memory can be accessed from all locations in the data-path, local memory can only be accessed from the module in which the local memory is instantiated. However, multiple local memories can be accessed simultaneously, while the global memory can only be accessed sequentially. To determine which values should be placed in global or local memory, LegUp uses static memory analyses. Even in the heterogeneus system, all memories have a latency of two clock cycles as the softcore processor is also placed on the FPGA and uses the same memory resources as the local memories for the global memory.

The main execution model of LegUp is a basic block FSM (see Section 2.2.1) without pipelining. In its latest release the support for loop pipelining was added. Figure 8.4 shows a schematic interpretation of the LegUp pipeline controller. Here it can be seen that the execution model is quite similar to the static II model (compare to the controller in Section 3.2.2).

The biggest difference of LegUp's pipeline model implementation is in the handling of which stages are activated. While Nymble uses a pipeline of registers (one register for each stage), LegUp uses an additional register which stores the currently active set of stages. This *II State* encodes the active stages into a single value between 0 and II-1. Each stage N, whose remainder of N divided by the II is equal to the value of this II state register, is activated. To handle the pro- and epilogue, i.e. where the pipeline is started or finishing, the II state is combined with a set of *Valid* registers. The combination of both is very similar to the stage register used in this work. However, Nymble used the pipeline model from the beginning, before it was integrated into LegUp.

LegUp supports both local and shared global memory. Local memory is used when LegUp can determine that an array is only used in the hardware accelerator. In its hybrid execution mode, LegUp automatically integrates the hardware-software interface.

LegUp supports multi-threading using n copies of the same data-path [CBA13]. These copies are instantiated according to pthreads or OpenMP (see Section 8.5) functions. Locks are supported using pseudo memory accesses that block until the lock is resolved. The same idea is planned for Nymble (see Section 10.2).

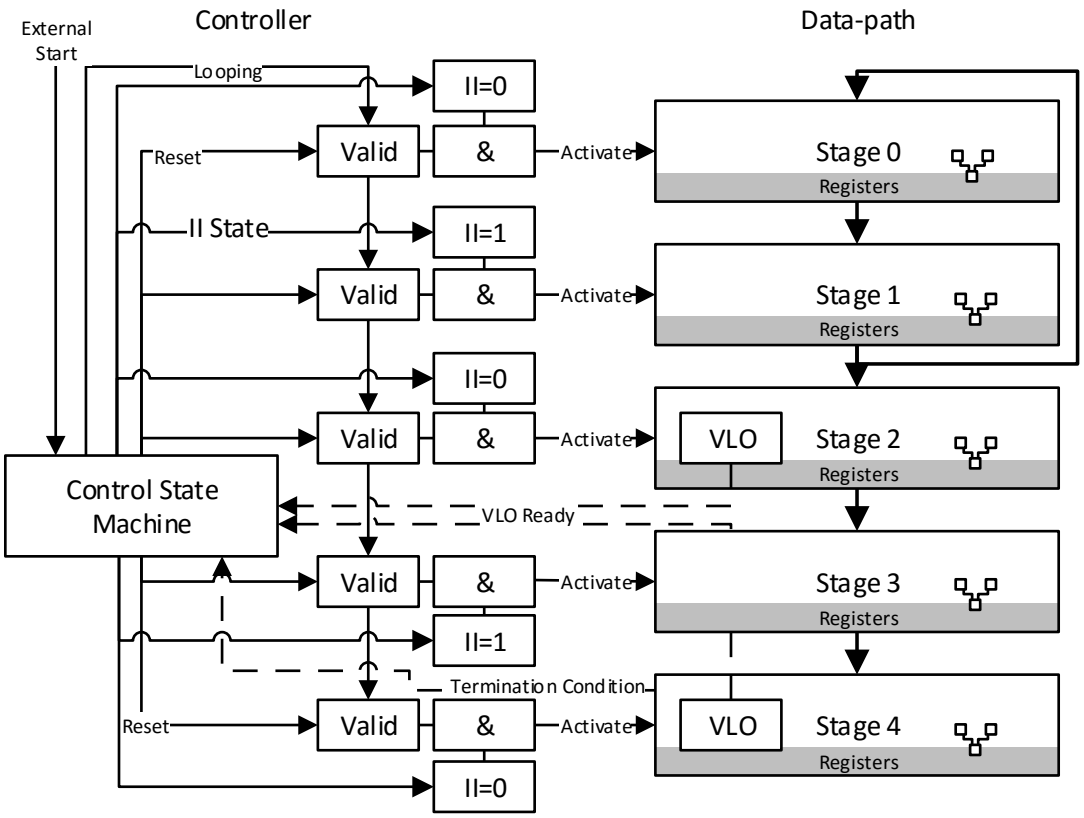


Figure 8.4: Schematic of pipeline controller used by LegUp (II=2)

8.3.2 Bambu

Bambu [PF12] is a HLS compiler based on GCC [SE16]. It can compile all benchmarks of the CHStone suite. Floating point operations are supported by using FloPoCo [DP11]. Bambu has a similar support as Nymble of hardware functions [Min+15] (see Section 3.3.6.2), which even supports function pointers (limited to non-recursive functions, as indicated by the explicit mentioning of only non-recursive benchmarks and future work). The different publications based on Bambu do not specify a target platform but Bambu is generally targeting generic FPGA based solutions.

The default execution model of Bambu is a basic block FSM (see Section 2.2.1) without pipelining. Other executions models (distributed controller, task based multi-threading with duplicated processing units) are currently research based on Bambu (both explained shortly). Like LegUp, Bambu supports local and shared (in the whole hardware kernel) memory. Similarly, Bambu assigns variables to these memories depending on static analyses. However, the generated hardware kernels currently only support physical addresses as a heterogeneous system is not the main target of Bambu. Generated kernels have to be manually integrated into a solution.

Based on Bambu, [Cas+15] presents a task-parallelism using a token based controller to control which tasks use which functional unit in the data-path. The token based distributed controller is similar to the one presented by [GL11] (here it controls single operations), which was one of the initial ideas for the work on the multi-threaded execution model and the dynamic II model. The distributed controller uses *Resource Managers* (RM) to handle concurrent access to same resource by multiple tasks. This manager is similar to the handling of VLOs with shared resources (Section 4.3.7.2) in the multi-threaded model. Besides the RM, the n copies are not interacting with each other, unlike the multi-threaded model presented in this work (multiple simultaneous threads in a single data-path).

Another task parallelism multi-threading model based on Bambu [Min+16] uses multiple replicate processing elements and a dynamic scheduler to distribute the execution of parallel loop bodies. In Figure 8.5 a schematic taken from [Min+16] shows the three basic components of their proposed solution. The *Kernel Pool* is the set of duplicated processing units (PU). The *Dynamic Task Scheduler* (DTS) uses a queue to keep track which tasks are ready and also stores their input parameters. In the DTS a dispatcher queries the status register and dynamically assigns the tasks to available PUs. The status register stores which PUs are available. Finally, the *Termination Logic* checks whether the number of completed tasks is equal to the number of spawned tasks and then asserts the done signal.

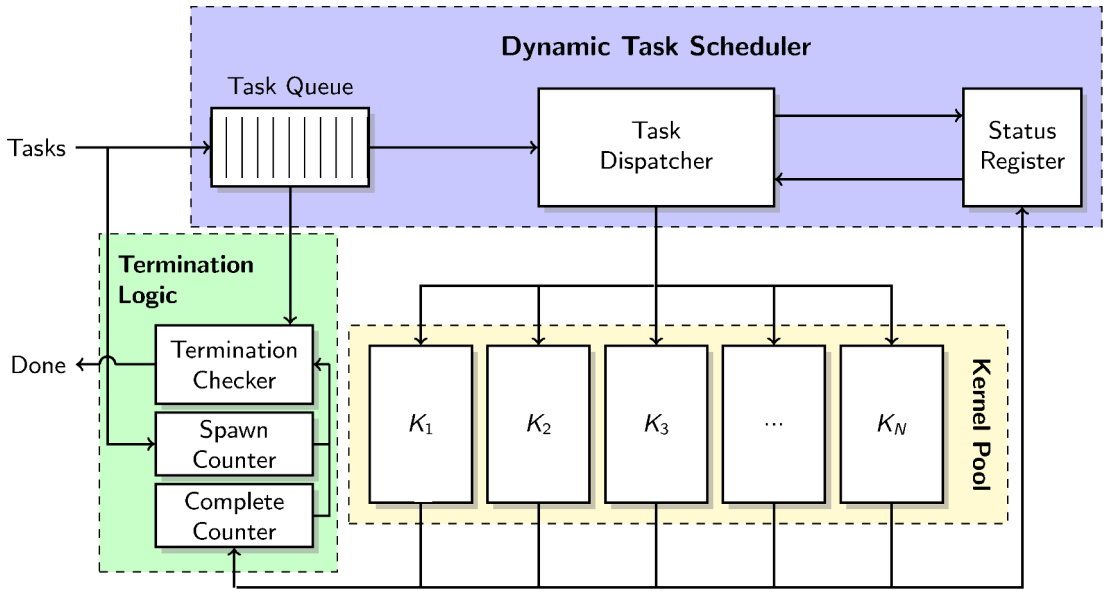


Figure 8.5: Schematic representation of task based multi-threading model of Bambu using a dynamic task scheduler (taken from [Min+16])

8.3.3 DWARV

DWARV 2.0 [Nan+12] is a HLS compiler based on CoSy [ACE03], released in 2012. It can compile all benchmarks of the CHStone suite. However, DWARV does not support global variables. Though this is the only language restriction compared to LegUp, Bambu, and Nymble. As global variables are required for CHStone, the benchmarks were modified to pass these variables as pointer parameters to the accelerated function [Nan+16].

DWARV uses a basic block FSM execution model for its generated kernels (Section 2.2.1).

According to [Nan+14] DWARV 2.0 targets similar heterogeneous systems as Nymble, as they support the Convey HC-2^{ex} and ML510 platform which are comparable to the platforms in Section 7.3 and Section 7.1, respectively.

In the same publication, it is described that DWARV supports multiple kernels which can share their local memory using a Network on Chip (NoC). While DWARV supports memory accesses with an arbitrary but fixed latency, it currently does not support such accesses with a variable latency, i.e. cached memory accesses.

8.3.4 ROCCC

ROCCC [Vil+10] is an open-source project that provides a HLS compiler for a subset of the C language. It supports only `for`-loops without arbitrary pointer support.

ROCCC targets no particular system as it only generated kernels with a generic vendor independent interface (streaming or random address memory accesses or simple registers). The generated kernels have to be manually integrated into a solution and ROCCC has no support for any kind of automated partitioning or integration of hardware software co-execution

ROCCC uses a distinction between *modules* and *systems*. A module is a concrete hardware block with a fixed number of inputs and outputs (Figure 8.6a). Modules can be included and combined in other modules (Figure 8.6b). Modules can only be written as straight non-looping code. A system then contains the main loops of the application (Figure 8.6c). These loops have to be perfectly nested loops, i.e. only the inner most loop can contain any ”work“ code. Finally, the whole system is connected to the outside using the generic memory interface.

An example for the memory interface can be seen in Figure 8.7. It shows how the data-path is connected to a controller, n memory input-, and m memory output-interfaces. In ROCCC, all memory have to be an array access and cannot be indexed by non loop index variables. Also in ROCCC, unlike its extension CHAT (Section 8.3.5), all memory accesses must be regular, i.e. continuous addresses. This way it is always possible to scalarize the memory accesses and to use a streaming memory access. The scalarization is done inside the input-interface with its associated *smart buffer*. The size of the smart buffer is configured by ROCCC. For the output-interface FIFO buffers are used to prevent stalls in the memory system propagating backwards into the data-path. In addition to that, all memory accesses in ROCCC must be independent as it has no support for loop-carried dependencies.

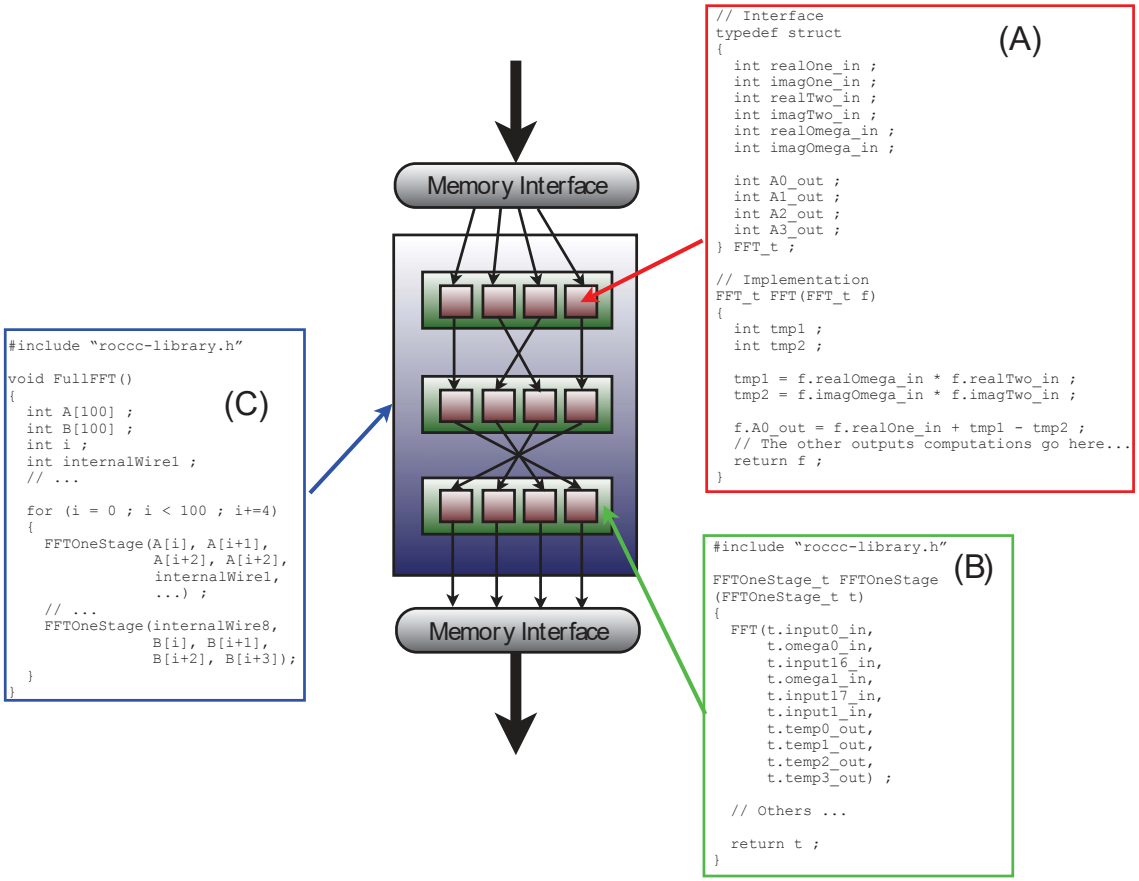


Figure 8.6: ROCCC schematic with input and output interface (taken from [Vil+10])

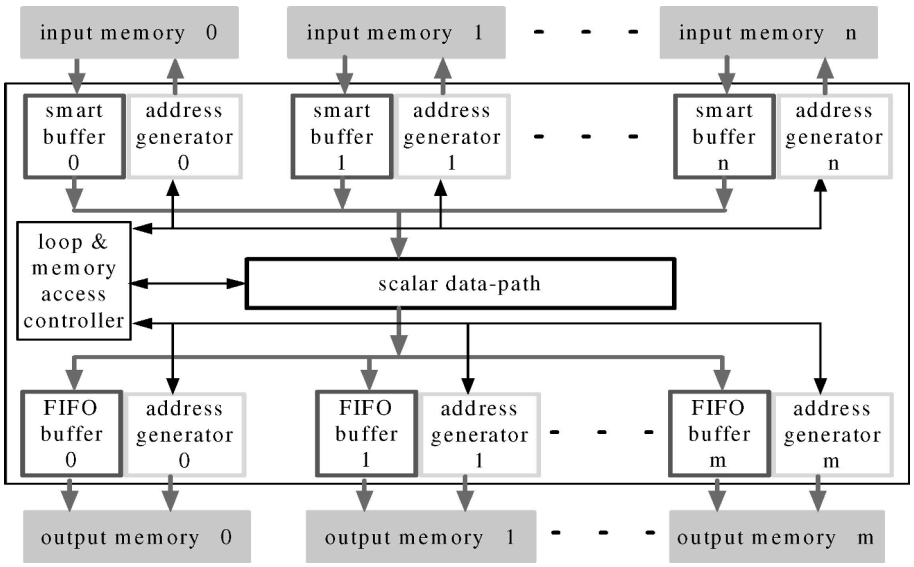


Figure 8.7: ROCCC memory interface schematic (taken from [GNB08])

8.3.5 CHAT

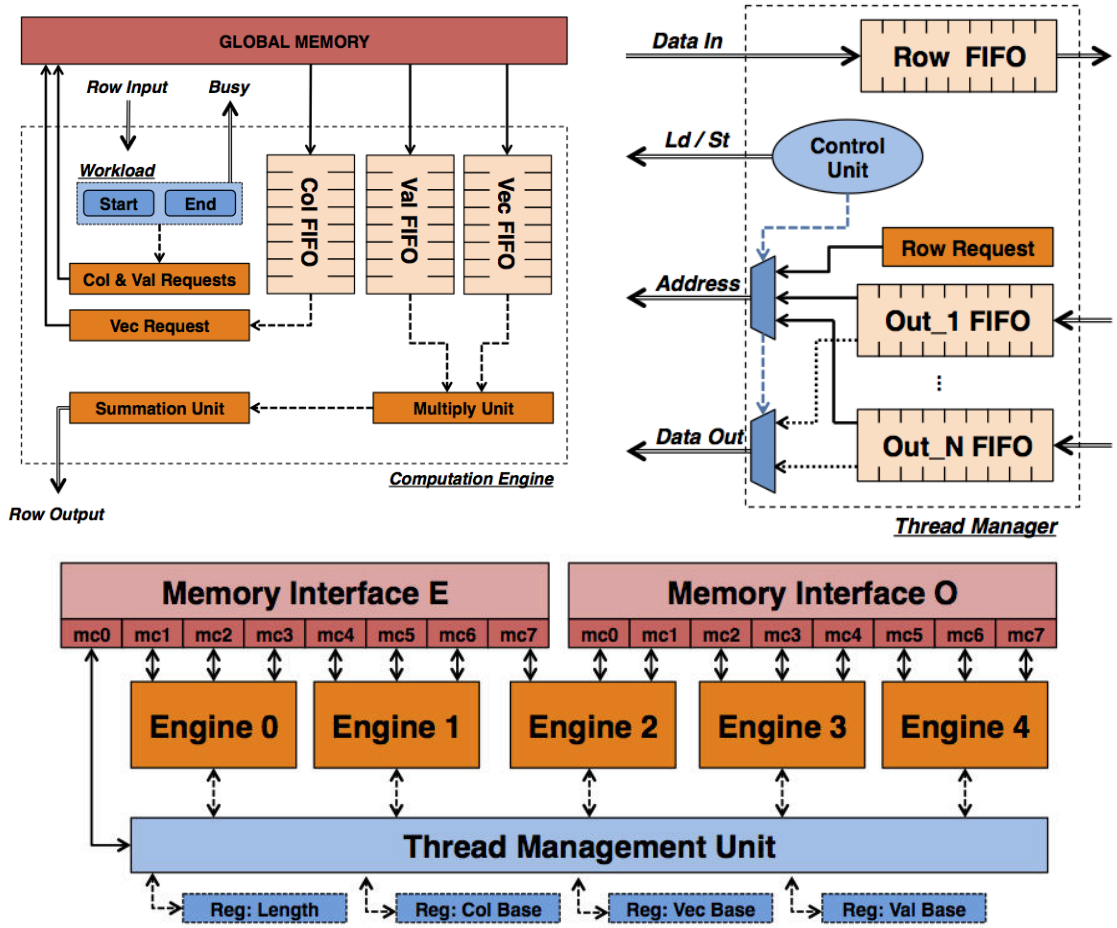
CHAT is an extension to ROCCC. “It uses the same underlying tools as ROCCC but targets irregular applications” [HN13]. CHAT provides multi-threading support for applications with irregular memory accesses. It slightly improves the handling of loops, which means that now some code is allowed around the inner most loop. This allows a dynamic workload for each thread specified by the inner loop, as each invocation of the inner loop is a new thread.

To support irregular applications, CHAT breaks the input controller into multiple parts. A single thread manager controls the invocations of the inner loop. The inner loop is executed as one of multiple computation engines (CE). The thread manager distributes the threads to the CEs and handles accesses to shared memory resources.

The example in Figure 8.8 shows a sparse matrix vector multiplication (SPMV) multi-threaded kernel generated by CHAT. It uses multiple identical *Computation Engines* (CE) which handle the inner loop of the SPMV. The CE here works in two steps, first reading from `col` the index of the irregular access to `vec`. After that both `val` and `vec` are accessed to add their product to the row sum.

The workload of each CE is determined by the *Thread Managment Unit* or *Thread Manager* which handles the outer loop. Here it uses two adjacent elements from `row` to set start and end positions for each CE. While the threads are distributed in-order to the CEs, the workload size can differ. Thus, the `out` array can be written to memory out-of-order. The kernel writes whenever a thread finishes.

All values besides the irregular access to `vec` are handled by streaming memory interfaces. Only `vec` uses a random address memory interface.




```
void spmv_csr (int *row, int *val, int *col, int *vec,
 int *out, int length) {
  int r, c, tmp;

  for(r=0;r<length;++r) {
    for(c=row[r]; c < row[r+1];++c) {
      tmp = tmp + val[c] * vec[ col[c] ];
    }
    out[r] = tmp;
  }
}
```

Figure 8.8: CHAT multi-threading with Computation Engine and Thread Manager (taken from [HN13])

8.4 Summary

This section gives an overview over the presented compilers in terms of execution model, memory model, and multi-threading support. In Table 8.1 shows a summary of the compared features in Section 8.3.

Compiler	Execution Model	Memory Model	Multi-threading
LegUp	FSM + Pipeline	Shared Memory	Duplicated CUs
Nymble	Pipeline + FSM	Shared Memory	Parallel Threads in CU
Bambu	FSM + others	Exclusive Memory	Duplicated CUs
DWARV	FSM	Shared Memory	Duplicated CUs
ROCCC	FSM	Streaming Memory	-
CHAT	FSM	"" + Irregular Accesses	Duplicated CUs

Table 8.1: Summary of HLS compiler features

For the execution model, the main model of each compiler is shown. Additionally, if available, the most often mentioned secondary model is also shown. Most compilers use a variant of the execution models that were described in Section 2.2.

For the memory model, the models are classified into the following three groups:

- *Shared Memory* between hardware and software parts of application provides the most flexibility as pointered data structures can be exchanged between the software and hardware.
- With *Exclusive Memory* only the hardware kernel can directly access the memory. Before the kernel can be executed the data has to be ”manually“ transferred to this memory.
- *Streaming Memory* means that the generated kernels can access the memory in sequential order only.

For multi-threading support most compilers are using *Duplicated Computing Units (CU)*, while only Nymble provides the support for parallel threads in a single CU. However, there is [Tan+14] which provides a similar approach as Nymble. But as they are using an unnamed commercial compiler as their base and because of a lack of further information, their work was not included in the detailed analysis.

8.5 OpenMP based Multi-threading

Software multi-threading is generally handled through frameworks which abstract the system specific implementation details. A well known framework that is used in recent research is OpenMP [DM98]. OpenMP provides a pragma to specify explicit tasks in an application.

This section will give an overview of works using OpenMP for HLS that utilize this pragma to create task specific hardware kernels.

fpBLYSK [Pod14] creates hardware kernels with n copies of a task specific core. Each core uses a FSM based execution model. An on-chip soft-core processor is used to create tasks and assign them to the cores. It also has to transfer all data from and to the cores, as the cores lack the ability to interact with memory.

While not requiring task pragmas, Harmonic [Luk+09] (based on Haydn HLS [CJL05]) partitions an application into tasks which are then executed on CPU, DSPs or custom kernels generated with Haydn HLS. Each task uses its own local memory which is filled upon task creation.

OmpSs@Zynq [Fil+14] is based on the Xilinx Zynq platform and uses Vivado HLS to generate task specific kernels. The system uses an abstraction layer to distribute tasks between the generated hardware kernels and other targets like CPUs and GPUs. Multiple tasks can be issued to a single generated hardware kernel using the handshaking generated by Vivado HLS. Compared to Nymble these kernels do not allow the reordering of tasks or threads.

Nymble-OMP [SOK16], which is based on Nymble, uses a statically scheduled non-pipelined execution model and creates n copies of the data-path for m threads. It generally uses fewer copies than threads, as a distributed thread context storage is used to switch stalled threads with another thread that is ready. It is mentioned that $n = 2$ copies for $m = 8$ threads achieves the best throughput-per area efficiency.

In addition to the task based methods, OpenMP also supports loop based multi-threading where the work in the loop nest is distributed among multiple threads. The following works use this work-share loop method.

Earlier works translate the OpenMP annotated loops into SystemC [Dzi+06] or HandleC [LNW06] implementations. These works do not provide a co-execution model of hardware and software.

The multi-threading implementation in LegUp [CBA13] is also based on work-share loop of OpenMP. For more details see Section 8.3.1.

9 Evaluation

The evaluation of the proposed multi-threaded execution will consider multiple aspects. First the area and runtime efficiency compared to the presented single-threaded models will be evaluated. Then the impact of different optimization for using the optional multi-threaded stages will be examined. Following this, hardware functions and the experimental basic block models will be evaluated. Afterwards, Nymble will be compared with other academic HLS compilers. Finally, some benchmarks will be analysed in depth to see which additional optimizations could improve the runtime.

But before all that, it will be explained exactly what was measured for the evaluations.

9.1 Measurement

All run-time evaluations were performed using a simulated memory system. To prove that the presented execution model actually works on real hardware, the benchmarks were synthesized and executed on the DINI system (Section 7.2). To evaluate the resources requirements, the benchmarks were synthesized for a Xilinx Virtex 7 VX690T FPGA (which is the one used in the DINI system). The details of both the simulation and synthesis are shown in Section 9.1.2 and Section 9.1.1, respectively.

To be able to compare the results between different HLS compilers, it is first necessary to define what is being measured. For example in [Nan+16], it is said that "*For each benchmark kernel, some data is kept local to the kernel (i.e. in BRAMs instantiated within the module), whereas other data is considered "global", kept outside the kernel and accessed via a memory controller*". As local memory is generally much faster than using global memory, it is really important to know which data is put into local memory, especially for benchmarks like CHStone. In CHStone, all input data, which is normally read from a file, is kept in relativly small constant arrays. The size of the input data and constant tables is shown in Table 9.1. These sizes were collected by searching for all `const` arrays defined in each benchmark's source code. All of these constant arrays, which where not clearly marked as *input data*, were counted as *tables*. MachSuite provides the memory footprint for each benchmark (shown in Figure 9.1). This footprint is defined as uniquely addressed bytes by each benchmark.

Benchmark	Input data	Tables
adpcm	400	820
blowfish	5200	4312
dfadd	736	-
dfmul	320	-
dfdiv	352	-
dfsin	576	-
gsm	320	256
mips	208	-

Table 9.1: Size of input data and constant tables in bytes for CHStone benchmarks

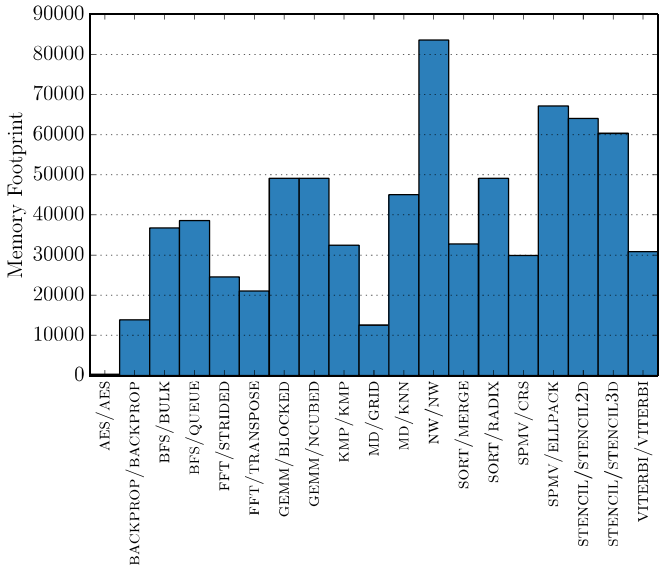


Figure 9.1: Memory footprint for MachSuite benchmarks (taken from [Rea+14])

```
... read input ...

#pragma HARDWARE on
run_benchmark( input );
#pragma HARDWARE off

... check output ...
```

(a) MachSuite

```
for (i = 0; i < N; i++)
  so[i] = inData[i];

#pragma HARDWARE on
Gsm_LPC_Analysis (so, LARc);
#pragma HARDWARE off

for (i = 0; i < N; i++)
  main_result += (so[i] != outData[i]);
for (i = 0; i < M; i++)
  main_result += (LARc[i] != outLARc[i]);
```

(b) CHStone (gsm)

Figure 9.2: Placement of partitioning pragmas in benchmarks

```
for (i = 0; i < N; i++)
{
  float64 result;
  x1 = a_input[i];
  x2 = b_input[i];
  #pragma HARDWARE on
  result = float64_add(x1, x2);
  #pragma HARDWARE off
  main_result +=
    (result != z_output[i]);
}
```

(a) dfadd

```
#pragma HARDWARE on
for (i = 0; i < N; i++)
{
  x1 = a_input[i];
  x2 = b_input[i];
  results[i] = float64_add(x1, x2);
}
#pragma HARDWARE off

for (i = 0; i < N; i++)
  main_result +=
    (results[i] != z_output[i]);
```

(b) dfadd_mod

Figure 9.3: Placement of partitioning pragmas in `df...` benchmarks

The benchmarks are taken from the CHStone [Yuk+08] and MachSuite [Rea+14] benchmark suites. For CHStone, the hardware partitioning pragmas were set around the topmost code in each benchmark that was possible without including the initialisation and result checking. MachSuite uses a universal harness to execute all benchmarks, so the partitioning pragmas (see Section 6.2) were set around the execution of the first benchmark specific function call. Figure 9.2 shows examples for the pragmas for both benchmark suites.

CHStone contains some benchmarks (all beginning with df...), where the original code prevented a pragma placement to include any loop in the hardware. This lead to some performance issues and thus these benchmarks were slightly modified by moving the result checking out of the main loop. These modified versions are marked by the suffix _mod. Examples for both can be seen in Figure 9.3, where a) shows the original and b) the modified version.

In all figures the benchmarks are always in the following order: Non-looping benchmarks from CHStone followed by the remaining CHStone benchmarks. Then all benchmarks from MachSuite without floating point operations followed by all benchmarks from MachSuite with floating point operations. An overview of all benchmarks with a short description and the section were each benchmark is first mentioned can be seen Table 9.2.

For previous publications of this work, it was always assumed that the input data was not held in local memory, as real applications work on much bigger data sets. It would be, however, allowed to place static tables local memories. Also, if a compiler can insert automatic or pragma guided transfers to a local memory for computation, the latency of these transfers has to be taken into account.

However, it seems that in the general community the approach to compare performance is to compare an accelerator working purely on local data. The main reason for that is that most HLS compiler and their memory models do not support a global shared memory. As the part of Nymble presented in this work has no support for local memory, the simulated memory model, which will be explained in Section 9.1.2, was adapted to emulate local memory. The adaptation details will be explained in Section 9.1.3.

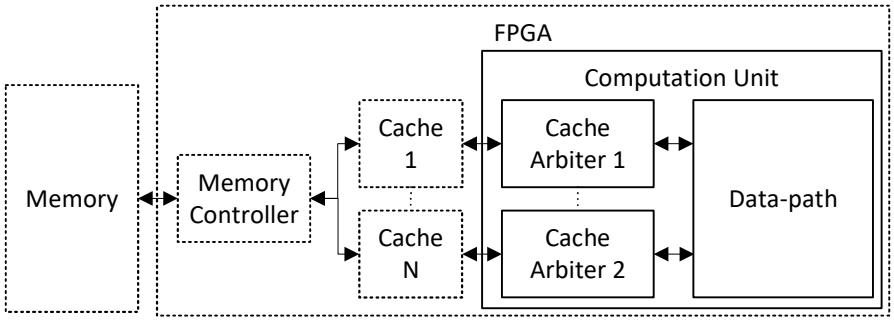
	Benchmark	Description	See Sec.	
CHStone	adpcm	adpcm encoder and decoder	9.1	Integer
	blowfish	data encryption	9.1	
	df_add	low level floating point addition	9.1	
	df_div	low level floating point division	9.1	
	df_mul	low level floating point multiplication	9.1	
	df_sin	low level floating point sinus	9.1	
	df_add_mod	modified to reduce context switches	9.1	
	df_div_mod	modified to reduce context switches	9.1	
	df_mul_mod	modified to reduce context switches	9.1	
	df_sin_mod	modified to reduce context switches	9.1	
	gsm	linear predictive coding analysis	9.1	
	gsm_mod	modified to reduce false dependencies	9.8.1	
	mips	simplified MIPS processor	9.1	
	sha	secure hash algorithm	9.1	
MachSuite	aes	advanced encryption standard	9.1	
	bfs_bulk	Data-oriented breadth-first search	9.1	
	bfs_queue	expanding-horizon breadth-first search	9.1	
	gemm_blocked	blocked version of matrix multiplication	9.1	
	gemm_blocked_mod	modified to reduce false dependencies	9.8.2	
	gemm_ncubed	naive, O(n3) dense matrix multiplication	9.1	
	gemm_ncubed_mod	inner loop unrolled eight times	9.8.3	
	kmp	Knuth-Morris-Pratt string matching algorithm	9.1	
	nw	optimal sequence alignment with dynamic programming	9.1	
	sort_merge	mergesort algorithm	9.1	
	sort_radix	sorts an array by comparing 4-bits blocks at a time	9.1	
	stencil2d	2d, 9-point square stencil computation	9.1	
	stencil3d	3d, 7-point von Neumann stencil computation	9.1	
	fft_strided	recursive Fast Fourier Transformation	9.1	Floating Point
	md_grid	molecular dynamics, using spatial decomposition	9.1	
	md_knn	molecular dynamics, using k-nearest neighbours	9.1	
	spmv_crs	sparse matrix-vector multiplication, using variable-length neighbor lists.	9.1	
	spmv_ellpack	sparse matrix-vector multiplication, using fixed-size neighbor lists	9.1	
	spmv_ellpack_mod	expression tree reordered for height reduction	9.8.4	
	viterbi	dynamic programing algorithm for computing probabilities on a Hidden Markov model	9.1	

Table 9.2: Benchmark Overview

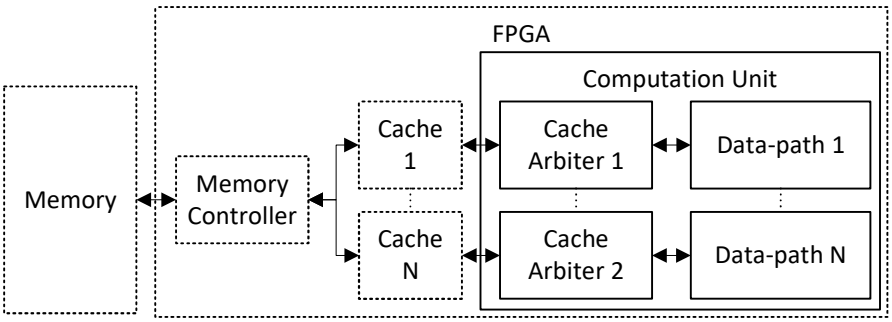
9.1.1 Synthesis

Figure 9.4 shows the parts of the DINI target system which are synthesized for the area efficiency evaluation as solid lines. To minimize the impact of the synthesis tool optimizations on the area efficiency evaluation, the single-threaded data-path was duplicated into N instances which where all synthesized together. This allows the synthesis tool to use the same LUT and Slice packaging methods for both the single- and multi-threaded data-paths. If the result of a single single-threaded instance would have just been multiplied by N, these optimizations would not have been included in the results.

For the functional test on real hardware, the whole system (including the dashed parts) was synthesized. The resulting bit-stream was then uploaded to the FPGA and each application was executed with four parallel threads.



(a) Multi-threaded data-path



(b) N copies of single-threaded data-path

Figure 9.4: Schematic of synthesized parts (parts with solid lines are synthesized)

9.1.2 Simulation

For the evaluation of the run-time a simulation model as shown in Figure 9.5 was used. In the ModelSim simulator, the computation unit was simulated together with a software model of the cache system. For each of the N threads, an exclusive

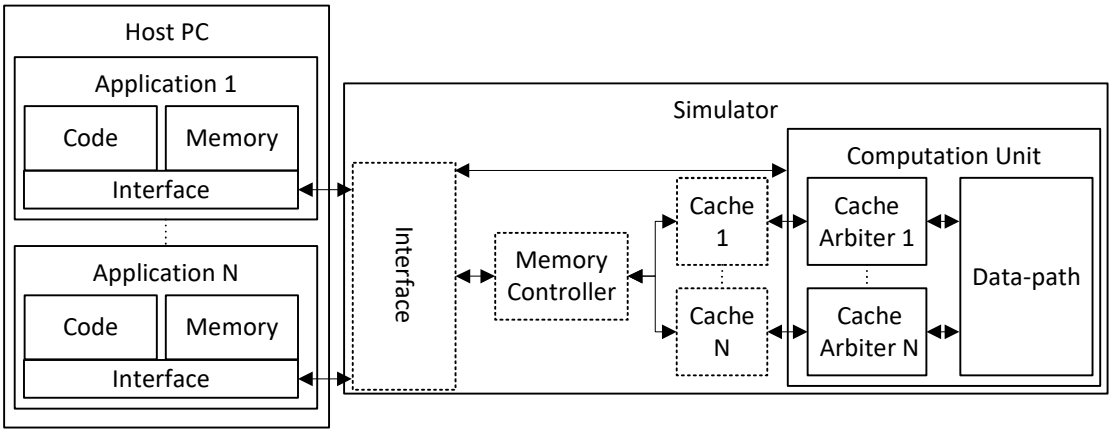


Figure 9.5: Schematic of simulation for run-time evaluation (parts done in software simulation is shown dashed)

cache is instantiated which provides two ports for each thread, one for reading and one for writing. A cache port arbiter then controls the order in which the memory read and write VLOs are allowed to access the cache (see Section 3.3.5).

The applications executed on the host communicate with simulator over a Linux socket based interface which provides bi-directional communication between each application and the simulator. In case a cache has to access the shared memory, it requests this access at the memory controller. The memory controller then requests the access from the simulator interface which selects corresponding application based on the address of the memory access. The application's interface then accesses the memory of application to resolve the memory access.

Each cache is configured with a latency of one clock cycle for a cache hit and 20 clock cycles for a cache miss. It provides 1024 cache-lines with eight 32 bit entries each. While the cache can be configured with a different number of ports for reads and writes, a detailed evaluation of the impact of the memory configuration is out of the scope of this work.

9.1.2.1 Removing Memory Impact

To evaluate the execution models' baseline without the impact of any memory latencies, all memory accesses in the data-path are replaced with an unlimited number of parallel single cycle memory accesses. These single cycle memory accesses only work in simulation and cannot be synthesized. However, the results in Figure 9.6 show that both pipelined models have similar runtime for a single thread. This ensures that differences in the runtime with non-zero latency memory accesses are resulting from the different handling of stalling and not, for example, differently scheduled pipelines. However, for a few benchmarks the slight differ-

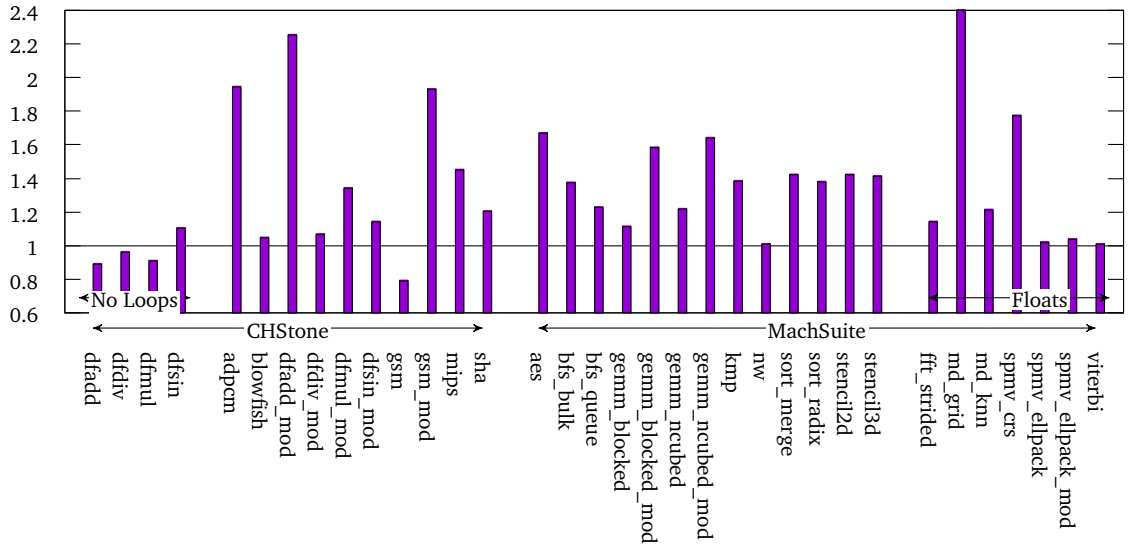


Figure 9.6: Normalized runtime between static and dynamic model with unlimited single cycle memory accesses (value > 1, dynamic model is better)

ences in the handling of nested loops has a significant impact of the runtime in favour of the dynamic model.

In the static II model the whole outer loop is stalled when an inner loop is executed, preventing other iterations from continuing. In the dynamic II model, however, the outer loop can let other iterations keep running. Both adpcm and md_grid have multiple loops nested in the same parent loop. In the static model these nested loops are executed sequentially, while the dynamic model can execute multiple nested loops at once (for different iterations). This difference has a big impact on the runtime, especially when an unlimited number of memory accesses can be executed whose sequential execution through the multiplexed cache ports would have otherwise limited the runtime.

For a completely fair comparison between the single and multi-threaded model, the best runtime from either the single-threaded static or dynamic model is used.

9.1.3 Local Memory Emulation

In the evaluation of related work it became clear that the measurements of these HLS compilers were performed using only on-chip memory. While limited in size, these memory blocks can be accessed in only one or two clock cycles. This means that the measurements of Nymble as previously published cannot be compared to these other systems. All target systems of Nymble are a heterogeneous system executing both a software- and hardware-part. These parts share the same virtual address space and access the shared memory through a cached memory system.

Because of that, the memory and cache latency can have a huge impact on the runtime.

To be able to compare measurements across tools using a similar configuration of fast on-chip memory, the simulated memory model was adapted to emulate such a configuration. As in reality the number of parallel accesses to these memory resources is still limited by number of available memory ports (typical block-rams have two ports, but multiple rams can be used for independent data arrays), the simulated memory only allows a specific number of parallel accesses in a single clock cycle before stalling all other accesses until the next cycle.

9.1.4 Runtime Measurement

The basic measurement of the runtime is the number of clock cycles between the first and last register transfer of a single hardware execution. For the comparison of the different execution models and the efficiency of the multi-threaded model, all data-paths were generated with the same target frequency of 100 MHz. The target frequency changes how many operations can be chained together in a single clock cycle. Because of the equal target frequency for all benchmarks and test configurations, the results can be compared without having to synthesize and evaluate the real maximum frequency. All benchmarks achieve at least 100 MHz in the synthesis for the execution on the DINI system (see Section 9.6). Figure 9.7 shows the invocation protocol of a typical application. For all parts in green, the clock counter for the runtime is active and are thus included in the overall runtime measurement. The counter is not reset when an application is using the accelerator multiple times.

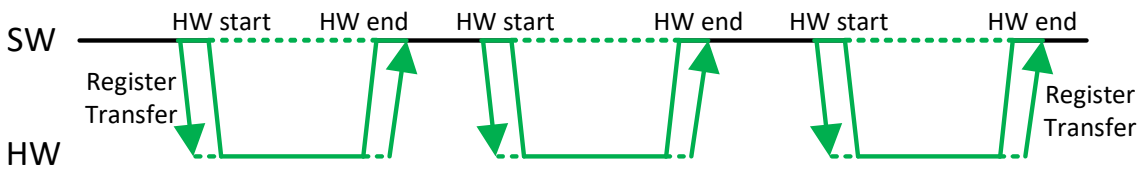


Figure 9.7: Invocation protocol with measured parts highlighted green

9.2 Efficiency of Multi-Threaded Model

To evaluate the efficiency of the multi-threaded model, the benchmarks are compared by executing with a single-threaded static II (Section 3.2), a single-threaded dynamic II (Section 3.3), and a multi-threaded implementation (Chapter 4) with 4 threads. Because the dynamic II was developed with multi-threading in mind, the single-threaded implementation still contains the TID which, however, should

be optimised away by the synthesis tool. As mentioned earlier, the target for all syntheses was the DINI system (Section 7.2).

The multi-threaded model has two general parameters, using *queues* to improve performance and using *all optional multi-threaded stages* (see Section 4.3.9). These performance queues (see Section 4.3.8) should not be mistaken with the queues used to correctly handle MCOs (see Section 3.3.4.1). Only the use of the performance queues can be enabled and disabled. A finer granularity for the usage of optional multi-threaded stages will be discussed in Section 9.3. Both options will be shown as *queues* and *opt. MT* in evaluation's graphs, respectively.

The efficiency is measured with two metrics, area (Section 9.2.1) and runtime (Section 9.2.2) efficiency. For the runtime efficiency the lowest runtime of the single-threaded implementations (static or dynamic) is multiplied by the number of threads (four in all measurements if not noted otherwise[1]) to compare the throughput of sequential against multi-threaded execution (for details see Section 9.1.2). For the area efficiency four copies of the single-threaded implementation are synthesized as well as a single four-threaded implementation (for details see Section 9.1.1). The resulting area usage measured in slices is then compared.

The efficiency is then shown as a factor between single-threaded and the multi-threaded implementation (for each value V it was calculated as $\frac{V_{ST} \times 4}{V_{MT}}$). Here a value greater than one means that the multi-threaded model is more efficient than the single-threaded implementation. The following sections will discuss the efficiency measurements in detail, starting with the area efficiency followed by the runtime efficiency.

9.2.1 Area Efficiency

Figure 9.8 shows the area efficiency for each implementation of the kernel relative to four copies of the static single-threaded model. This figure uses the best results obtained from using different optimization for choosing which optional multi-threaded stages are used (see Section 9.3). For comparison Figure 9.9 shows the efficiency without applying any optimizations. From this it can be seen that without using area optimizations about half the benchmarks become area inefficient. In Section 9.3 it will be shown that these optimizations can be applied with only a very small impact on the runtime efficiency.

While the multi-threaded model performs well for many benchmarks, there are also benchmarks which are less efficient than the single-threaded model. To understand why some applications are more suitable for the multi-threaded model

[1] The hardware software co-simulation has a problem with some bigger (input data) benchmarks when executed with eight threads. A few benchmarks were executed on the DINI system with eight threads.

than others, the efficient and inefficient applications are analysed in order of the benchmarks (see Section 9.1).

df... Benchmarks

As these benchmarks contain no loops, most features of the multi-threaded model are not used. In general, they are more efficient with the multi-threaded model than the single-threaded model. However, Nymble currently handles non-loops as loops with a single iteration. Because of that, they contain some overhead only really necessary for loops. Optimizing the back-end to handle non-loops more effective would also be useful for improving the multi-threaded basic block model as each basic block is generally implemented as such a non-loop. Additionally, because basic blocks are generally much smaller than in the df... benchmarks, the overhead has an even bigger impact.

The modified versions of these benchmarks have a lowered area efficiency because of the additional loop and its multi-threading overhead. However, the overall runtime of modified versions is much lower than the unmodified one because of reduction of the hardware software interface latency by doing less transitions through the interface.

Remaining CHStone and integer MachSuite Benchmarks

These benchmarks are mostly just a bit more efficient with the multi-threaded model. The cause for this is that the relative multi-threading overhead increased without bigger operations like floating point ones. This is proven by the fact that most benchmarks with floating point operations are more efficient than all non floating point benchmarks.

While queues have a negligible impact on the area efficiency, enabling optional reordering stages can have a significant impact on area efficiency. Both features also have an impact on the runtime efficiency which is analysed in Section 9.2.2.

Another important point for the efficiency is the use of complex operators. Without a relative high amount of complex operators as in most benchmarks without floating operations, the efficiency is not as good as for benchmarks with many complex operators. For simple operators like integer addition and logic, the multi-threading logic overhead is much bigger compared to complex operators. For example, if a simple one bit comparator result has to be stored in TCS, it uses at least two additional bits for addressing (with four threads) in addition to the multiplexing logic. When a 32 bit float MCO is made multiplexed, at best the same two additional bits addressing are needed. Of course, in many cases the operator requires additional queues, reducing the efficiency as shown in Sections 3.2.4.1 and 4.3.7.1. dfdiv and spmv_crs are the most extreme outliers. dfdiv has a single integer divider (complex operation) compared to dfadd and dfmul. On the other

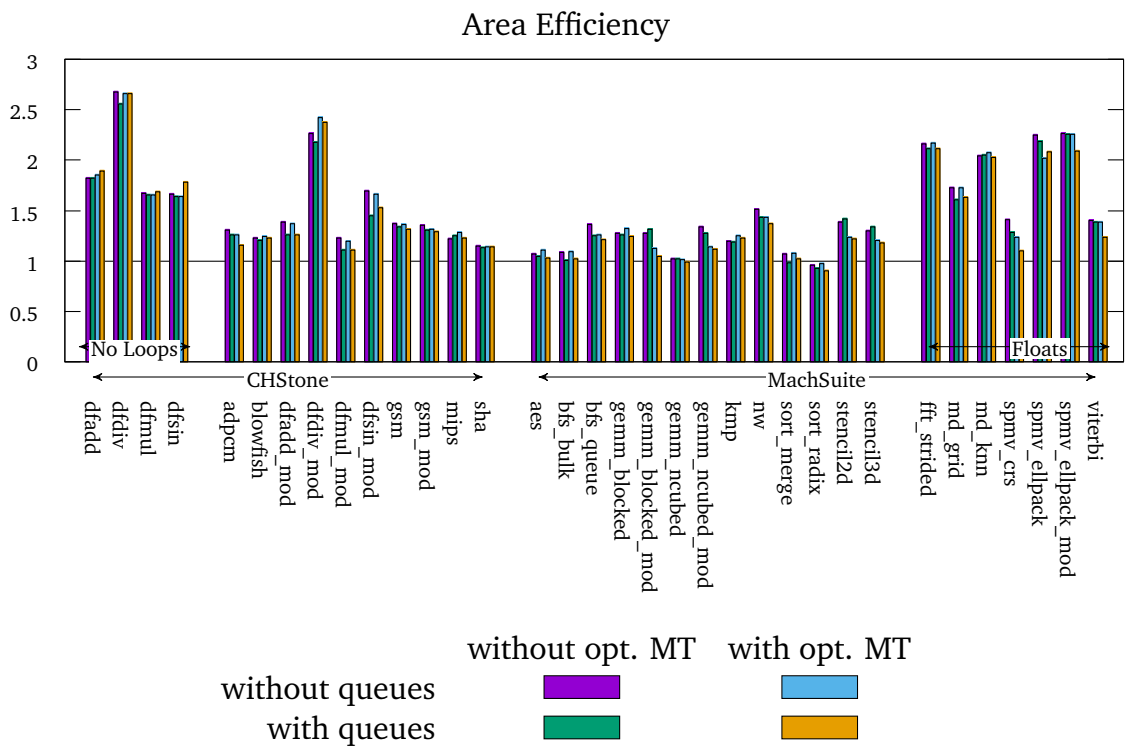


Figure 9.8: Area efficiency of multi-threaded model compared to four copies of static single-threaded model with optimizations (minimum values out of all optimizations, see Section 9.3). The multi-threaded model is more efficient for values greater than one

hand, spmv_csr has only two floating point operations which provide not enough potential for the multi-threaded model.

Overall, it can be said the multi-threaded model is more area-efficient for benchmarks with complex operators.

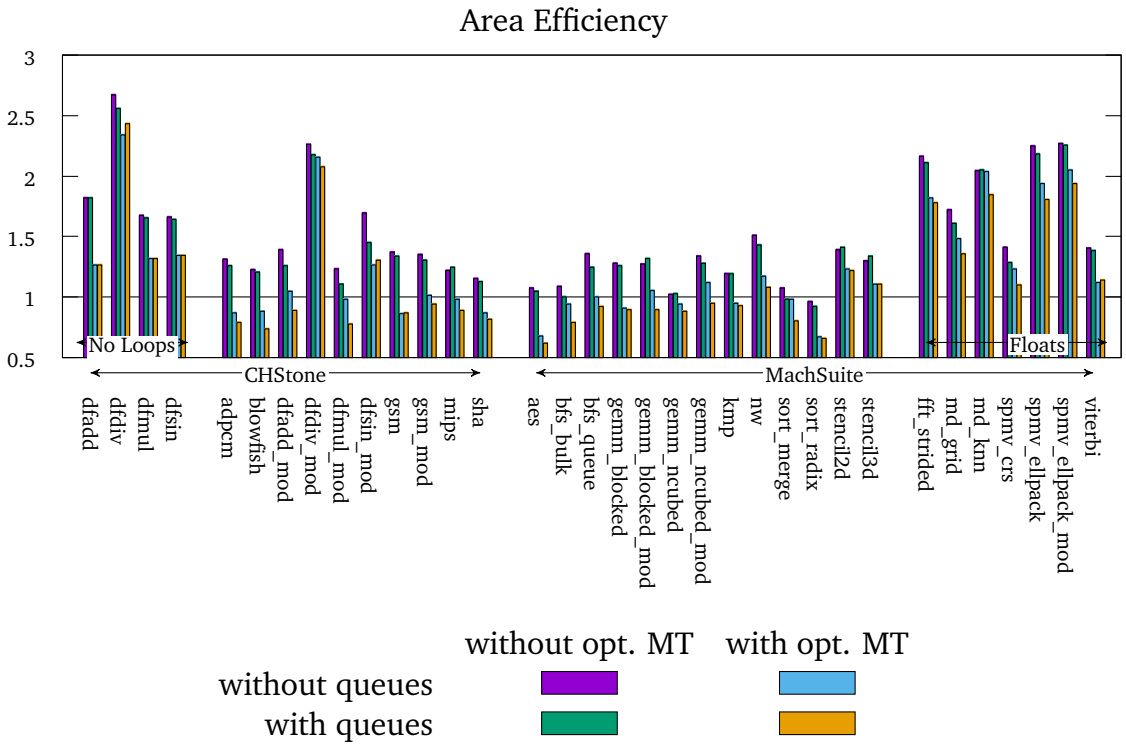


Figure 9.9: Area efficiency of multi-threaded model compared to four copies of static single-threaded model without optimizations. The multi-threaded model is more efficient for values greater than one

9.2.2 Runtime Efficiency

Similar to the area efficiency, Figure 9.10 shows the runtime efficiency for each implementation of the kernel relative to four consecutive sequential executions of the single-threaded models (static or dynamic). Thus, the multi-threaded and single-threaded versions perform the same number of computations.

Again, the impact of the different features on the unmodified df... benchmarks is almost not existent as they contain no loops (see discussion in Section 9.2.1). Currently, the hardware threads are all issued by the software. The context switch latency between the hardware and software is higher than the runtime of a single thread in the small df... data-paths. Because of that there is almost never a time where multiple threads are at the same point in the pipeline, requiring thread reordering. That is why there is no almost difference in the runtime efficiency between the different parameter options.

Overall the impact of queues and using optional multi-threaded stages on the runtime efficiency of all CHStone benchmarks is limited. The reason for that is that the threads spread out enough so that reordering at memory accesses is not necessary most of the time. This effect can be seen more clearly in the optimizations aiming to improve area efficiency by reducing the number of optional multi-threaded stages in use (see Section 9.3). For CHStone, these optimizations have almost no impact on the runtime efficiency.

The reduced efficiency of df_sin_mod results from a starved thread. This means that a thread is waiting at multi-threaded stages much longer than the other threads to be selected to move into the next stage. This starvation is caused by combination of the static reordering priorities and a specific placement of multi-threaded stages. In the modified version of df_sin, the new outer loop has a small $II = 7$ with a pipeline length of 39. However, df_sin is the only benchmark out of all df... which actually contains an inner loop (which is not long enough to hide the previously discussed context switching latency). However, this inner loop is located almost at the end of the new outer loop. Additionally, the stage directly before the inner loops VLO contains a memory access VLO. Because of the small II compared to length and the enabled queues, many iterations of the outer loop are started before this memory access (a cache miss) is completed once. This leads to the situation that next iteration of the highest priority thread is already in the input queue of the memory access and thus immediately issues the next request (a cache hit).

By itself this does not starve the other threads because only cache misses require the single shared memory controller. However, before the second highest priority thread has finished its first access (a cache miss), the ninth access of the highest priority threads results again in a cache miss because it already read all data from the previously loaded cache-line and has to wait for other accesses to finish. After

that, these two threads now alternate between reading from the cache and having to access the shared memory controller. This starves all threads with lower priority until the highest priority thread is finished and the same behaviour continues with second and third highest priority thread. In fact, this continues until only a single thread is left. An improved prioritization in the memory system (a fair arbiter based on the wait time) would solve this by making sure that no thread is starved. But in general the simple static priority works for most benchmarks.

For many benchmarks of MachSuite, however, the impact of using queues and optional multi-threaded stages can be clearly seen. Using them increases the runtime efficiency of almost all MachSuite benchmarks. But this also decreases the area efficiency which can be seen especially for spmv_crs in Section 9.2.1. This higher impact of thread reordering comes from the higher number of instances where threads are blocking each other at memory accesses in the pipeline. However, unlike the problem with df_sin_mod, in these benchmarks the reordering in the optional multi-threaded stages improves the runtime efficiency because while there are multiple threads in a stage with memory access VLOs, no single thread is starved by the other threads. The timing of the cache hits and misses leaves enough room for the reordering to be useful.

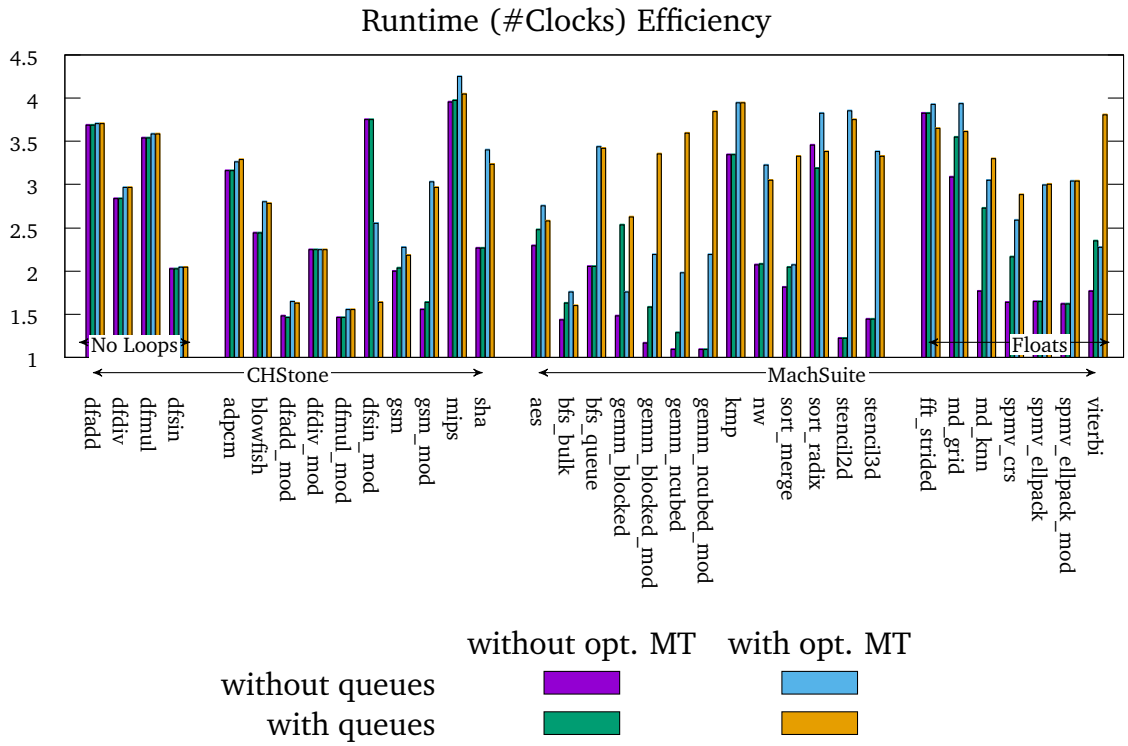


Figure 9.10: Runtime efficiency of multi-threaded model compared to four consecutive runs of single-threaded model without optimizations. The multi-threaded model is more efficient for values greater one

9.3 Optimised usage of optional Multi-Threaded Stages

Comparing Figure 9.8 and Figure 9.9 it can easily be seen that without an optimized placement of the multi-threaded stages the area efficiency is often worse than simple copies of single-threaded implementations. Because of that this section will discuss and evaluate different strategies to place multi-threaded stages. As described in Section 4.3.9, multi-threaded stages are mandatory only to encapsulate entire loops, which are treated as VLOs from the perspective of the surrounding loop. Again, without multi-threading support, only a single thread could enter a loop (while the rest would be blocked in a prior stage) which would prevent multi-threading especially in those parts of the program that could profit from them most. In all other places, multi-threaded stages are just optional. For the benchmark stencil3d (shown in Figure 9.11) of the MachSuite benchmark collection, the structure of optional and mandatory multi-threaded stages is shown in Table 9.3. The levels indicate the loop nesting levels, with 0 indicating the main function itself.

It is thus promising to explore if (and which) optional multi-threaded stages could be removed from the accelerator with limited (or ideally even no) loss in performance. To this end, Nymble inserts per-thread performance counters into each pipeline stage. These track the number of cycles this stage would stall if only mandatory multi-threaded stages were used. For stencil3d, this is shown in Table 9.4.

Most of the activity occurs in the inner loop (at Level 3). Obviously, the optional multi-threaded Stage 4 would be useful (as its lack causes a large back-pressure of stalls, due to the inability to reorder threads in the *seven* memory read operators located at that stage). On the other hand, a multi-threaded Stage 9 would not be that useful (only a relatively small number of stalls occurs in the single write operator). The results of these studies have led to two proposed optimization heuristics.

In the next two sub-sections the impact of the optimization parameters will be analysed with the following metrics. As all optimizations try to improve the area efficiency by reducing the number of multi-threaded stages, the runtime is evaluated relative to an implementation using all optional multi-threaded stages. The runtime is also shown as an efficiency factor compared to four sequential single-threaded executions, similar to Section 9.2.2.

The area efficiency is then evaluated by comparing it with four copies of the static single-threaded model, similar to Section 9.2.1. For convenience, it will also be shown as the relative improvement compared to using all optional multi-threaded stages. As the area efficiency depends on non-deterministic synthesis algorithms (Place & Route), as an alternative the same comparison is also done in terms of registers and LUTs.

```
// Stencil computation
loop_height : for(i = 1; i < height_size - 1; i++){          // Level 1
  loop_col : for(j = 1; j < col_size - 1; j++){              // Level 2
    loop_row : for(k = 1; k < row_size - 1; k++){            // Level 3
      // most activity in Stage 4 due to the seven memory reads
      // stage 0-3 contains address and loop counter calculation
      sum0 = orig[INDX(row_size, col_size, k, j, i)];        // Stage 4
      sum1 = orig[INDX(row_size, col_size, k, j, i + 1)] + // Stage 4
             orig[INDX(row_size, col_size, k, j, i - 1)] + // Stage 4
             orig[INDX(row_size, col_size, k, j + 1, i)] + // Stage 4
             orig[INDX(row_size, col_size, k, j - 1, i)] + // Stage 4
             orig[INDX(row_size, col_size, k + 1, j, i)] + // Stage 4
             orig[INDX(row_size, col_size, k - 1, j, i)];  // Stage 4
      mul0 = sum0 * C[0];
      mul1 = sum1 * C[1];
      // only a single memory write in Stage 9
      sol[INDX(row_size, col_size, k, j, i)] = mul0 + mul1; // Stage 9
    }
  }
}
```

Figure 9.11: Source code for stencil3d's main loop

9.3.1 Backwards Removal

When examining the profiling results for larger examples, it becomes clear that the lack of optional multi-threaded stages in the later stages of the inner loops causes fewer stalls than having optional multi-threaded stages missing from the earlier stages. This is already visible in the small stencil3d example of Table 9.4, where in the Level 3 loop the lack of multi-threading at Stage 4 is much more severe than on Stage 9.

This observation can be explained by considering the nature of data-path execution in the presence of VLOs. At the beginning of the loop (which itself is a VLO), all threads start at the same time, causing significant demand for shared resources (such a memory accesses), and thus being most likely to stall. The variable-latency of the VLOs then causes a spreading-out-in-time of threads, as they get deeper in the pipeline (since the threads are often subject to different latencies). Thus, there is less potential for conflict among threads and, correspondingly, less need for re-ordering by a multi-threaded stage.

A very simple heuristic can thus, for each loop level, attempt to omit the last N optional multi-threaded stages and only implement the earlier ones in each pipeline. The impact of this approach is shown in Figure 9.12, relative to an accelerator using *all* optional multi-threaded stages. For each benchmark, each

Stage	VLO
0	
1	
2	R:1
3	R:1
4	
5	L:1
6	
7	

(a) Level 0

Stage	VLO
0	
1	
2	
3	L:1
4	
5	
6	

(b) Level 1

Stage	VLO
0	
1	
2	
3	L:1
4	
5	
6	

(c) Level 2

Stage	VLO
0	
1	
2	
3	
4	R:7
5	
6	
7	
8	
9	W:1
10	
11	

(d) Level 3

Table 9.3: VLOs and multi-threaded stages for stencil3d. Column VLO shows number of R=read, W=write, L=loop VLOs per stage; boxed=mandatory, grey=optional multi-threaded stage

Stage	Stall
0	0
1	20
2	20
3	0
4	0
5	6434k
6	0
7	0

(a) Level 0

Stage	Stall
0	0
1	0
2	0
3	6433k
4	0
5	0
6	0

(b) Level 1

Stage	Stall
0	0
1	0
2	0
3	6428k
4	0
5	0
6	0

(c) Level 2

Stage	Stall
0	11700
1	1531116
2	1575000
3	1575000
4	1575000
5	0
6	0
7	0
8	0
9	450934
10	0
11	0

(d) Level 3

Table 9.4: Per-stage stall counters for stencil3d, with only mandatory multi-threaded stages (shown only for a single thread)

bar indicates the last $N = 0 \ldots 4$ multi-threaded stages being dropped from each loop level. Figure 9.12 gives **a)** the run-time efficiency in clock cycles compared to four consecutive executions of a single threaded kernel (note: the clock frequency itself was not negatively affected by the multi-threaded stage removal), **b)** the area-efficiency compared to four copies of the static single-threaded model, and **c)** the relative number of LUTs in the accelerator core (for details see Section 9.1.1 and 9.1.2).

The results shown in Figure 9.12b are already promising: Even dropping only the last multi-threaded stage from a loop level can result in LUT savings of up to 14% (e.g., for aes), while maintaining the same performance of the fully-multi-threaded-stage-populated version. For many benchmarks, even the four last multi-threaded stages can be dropped without adversely affecting performance (e.g.,

fft_strided, spmv_ellpack), leading to LUT savings of up to 34% for aes. However, there is too much of a good thing: The performance of stencil_2d and stencil_3d begins to deteriorate if more than one optional multi-threaded stage is dropped, while gemm_ncubed cannot afford even the removal of a single multi-threaded stage[2] Obviously, a more targeted optimization strategy is required for consistent results.

9.3.2 Profile-Guided Placement

Multi-threaded stages can be placed more selectively by taking the actual run-time behaviour of the accelerator into account. This is achieved by relying on the performance counters described in Section 9.3 to collect a stall profile when executing the accelerator (in simulation or hardware), with only mandatory multi-threaded stages present, on representative input data.

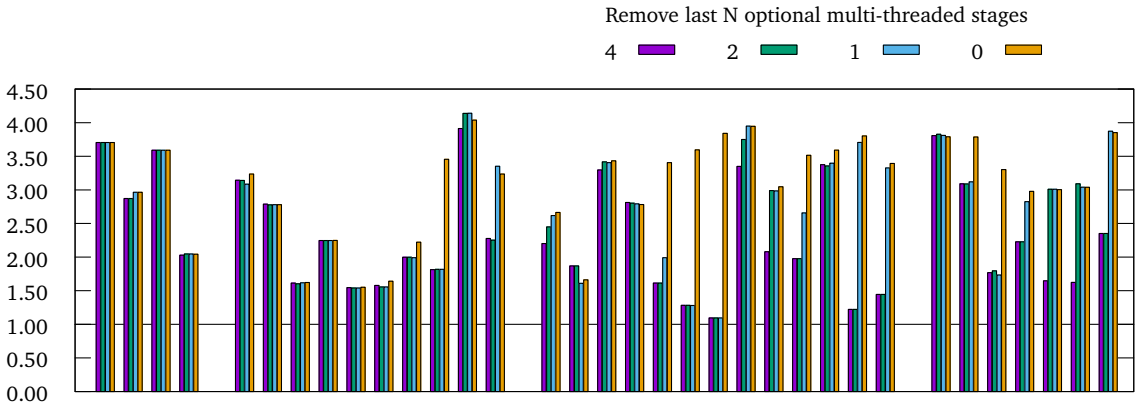
The key idea here is to selectively insert only those optional multi-threaded stages that are responsible for stalls that make up a significant fraction Q of the entire execution clock cycles C_{total}. An optional multi-threaded stage will be used if $\frac{1}{Q} \times C_{total} < C_{stage}$, i.e for $Q = 100$ the stage has to stall for at least 1% of the entire execution clock cycles.

As the choice of Q is crucial for this heuristic, the performance was evaluated over a wide range of values to examine the robustness of the algorithm. Note that this algorithm still relies on the already restricted places for multi-threaded stages in general (see Section 4.3.9). For stencil3d, this leads to the multi-threading-ineligible (not containing loops or VLOs) stages 1...3 in Loop Level 3 (Table 9.4.d) being disregarded for multi-threaded stage placement (despite their large stall counts), as the observed stalls are just the result of back-pressure that will be removed by a multi-threaded stage at Stage 4 (which, being optional, is actually eligible for multi-threaded stage placement).

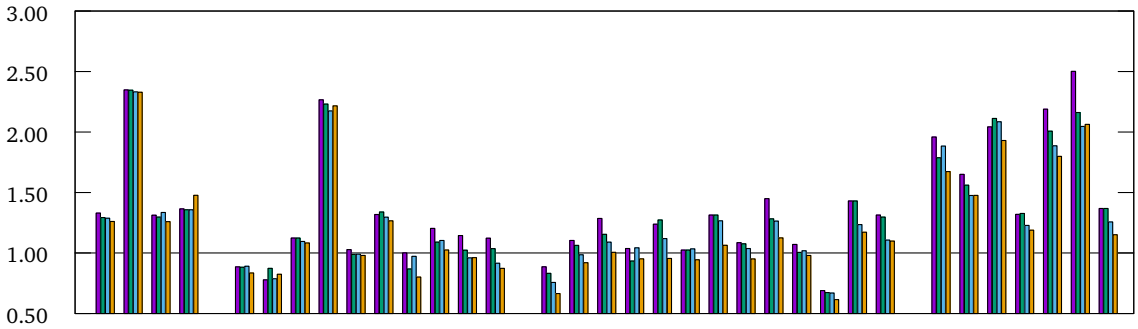
As in the previous section, Figure 9.13 gives **a)** the run-time efficiency in clock cycles compared to four consecutive executions of a single threaded kernel (note: the clock frequency itself was not negatively affected by the multi-threaded stage removal), **b)** the area-efficiency compared to four copies of the static single-threaded model, and **c)** the relative number of LUTs in the accelerator core (for details see Section 9.1.1 and 9.1.2).

Figure 9.13 shows that the approach is robust with regard to the choice of Q. It already gives good results for $Q = 10,000$ (stalling for 0.01% of the total run-time)(e.g., yielding 20% of LUT savings for aes), with only minor area improvements achievable for $Q = 1,000 \ldots 100$. Only for $Q = 10$ the insertion criterion

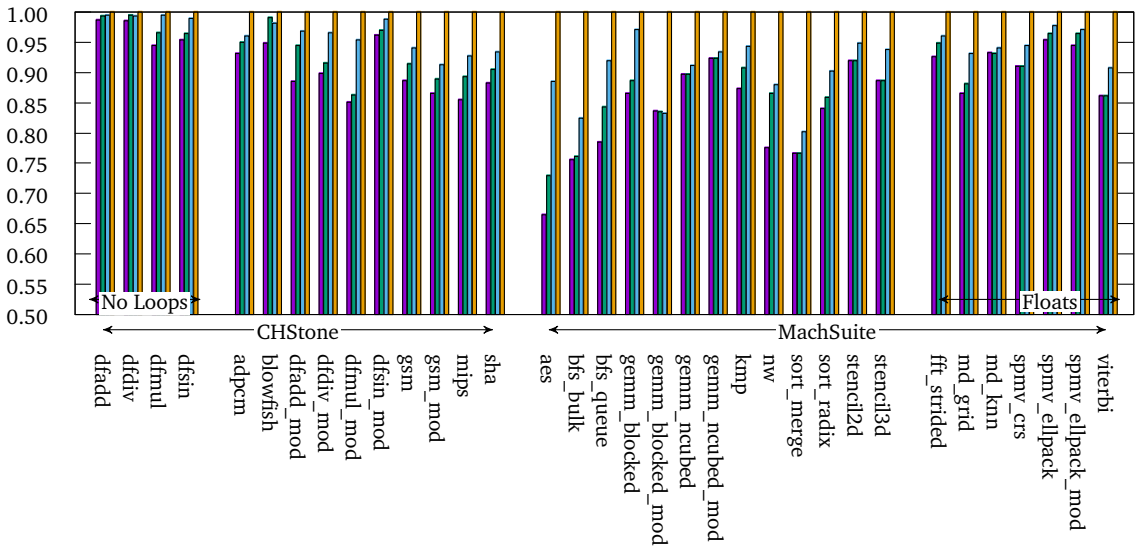
[2] The odd effect of *increasing* performance in fft_strided when dropping more stages appears to be a side-effect of also removing the priority-based scheduler in the HTS, which in itself might not be the best scheduling strategy for the specific benchmark.



(a) Runtime efficiency (#Clocks) compared to four consecutive runs of single-threaded model



(b) Area efficiency compared to four copies of single-threaded model



(c) Relative #LUTs

Figure 9.12: Optimization by removing the last N optional multi-threaded stages

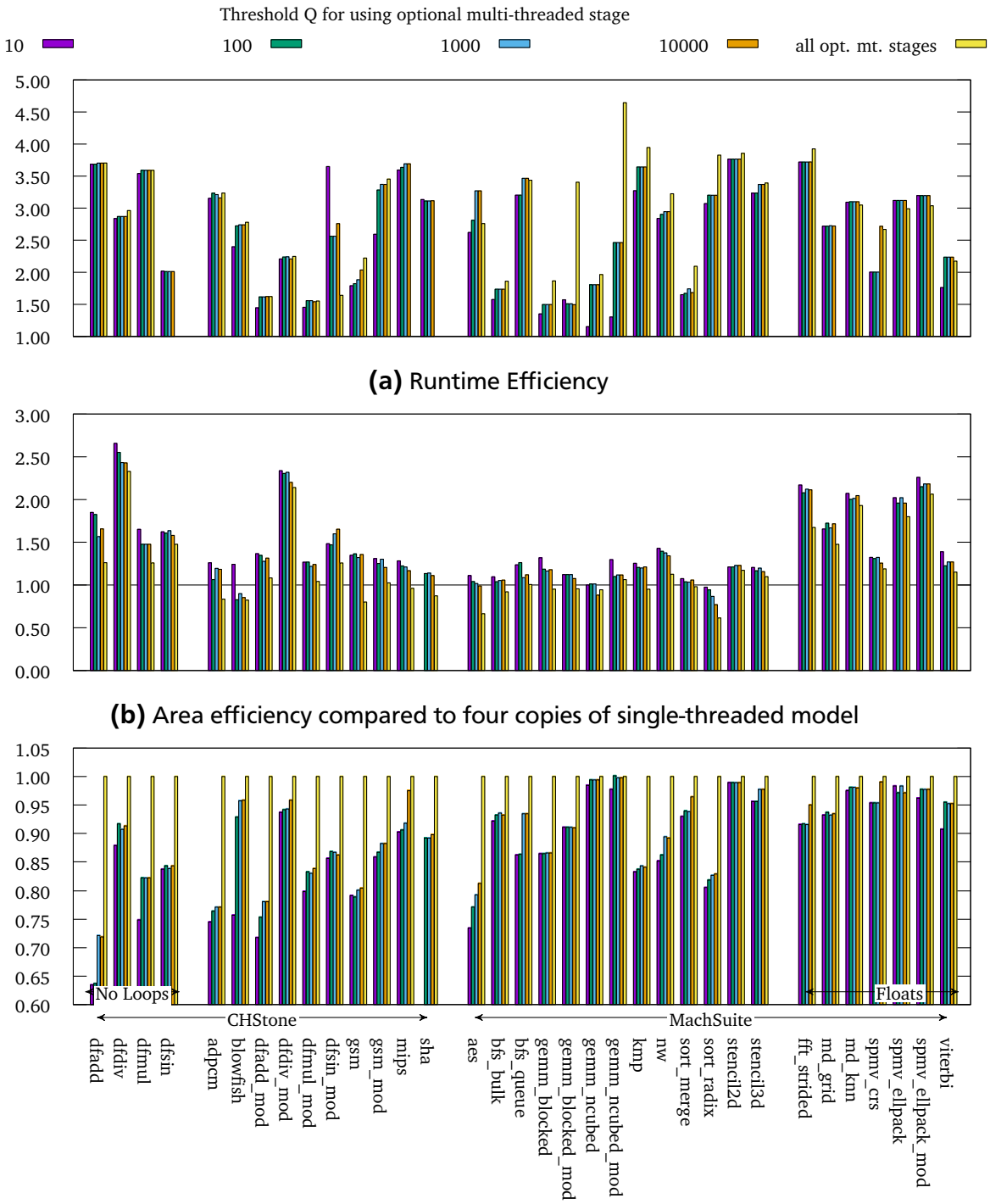


Figure 9.13: Profile-guided placement

becomes too selective: A multi-threaded stage would only be used on a stage with stalls exceeding more than 10% of the entire execution time. The benchmark gemm_ncubed is adversely affected by this, as too many useful multi-threaded stages (but having smaller stall counts) would be omitted.

The strange behaviour of dfsin_mod, where it achieves the highest runtime efficiency (i.e. fastest runtime) with the least amount of optional multi-threaded stages ($Q = 10$), is caused by the static priorities in the thread reordering. Some threads are starving because a new iteration of a higher priority thread is already available before the lower priority thread could continue. By removing prior reordering stages, the iterations are spread farther from each other so that the starving does not happen. A solution would be to improve the priorities for thread reordering.

Compared to the Backwards Deletion heuristic, the results of the profile-guided approach are much more predictable (e.g., choosing $Q = 1000$), while still realizing almost all the benefits (similar performance with less hardware area).

The key disadvantage of this approach is, of course, that the placement process now has to include the profiling (simulation or generation of actual hardware) of the accelerator. For the benchmarks presented here, simulation takes between 4 and 14 minutes on current x86 compute servers.

9.4 Basic Block Execution Model

The main point of this work lies in the multi-threaded pipelined model. However, as explained in Chapter 5 it is possible to emulate the basic block model by re-purposing some parts of multi-threaded model. It even is possible to still allow for multi-threading in the (emulated) basic block model.

But, as these re-purposed parts are not optimised for the basic block model, some compromises had to be taken into account. All basic blocks are currently executed as single-iteration loops. This means that they still contain some logic necessary for loops with more than one iteration. The overhead in terms of runtime and area currently reduces the usefulness of this basic model implementation to the point that it currently only shows some kind of runtime improvement for very few benchmarks.

However, with an optimised back-end for such non-loops, this overhead could surely be decreased to the point where the (emulated) basic block model can be mixed with the pipelined model for an increased efficiency of the entire accelerator.

In addition to the *emulation* (Chapter 5), another method of *extracting* basic blocks into nested single-iteration loops was shown in Section 3.3.6.1. While the emulation variant executes whole loops using a FSM as controller, the extraction variant only extracts some basic blocks selected on a heuristic into single-iteration

loops. This heuristic extracts a basic block when its execution count is at least N times smaller than the execution count of its parent loop. The emulation variant currently just executes the outer-most loops with the basic block model. This simple selection was done to show that both the pipeline model and the basic block model can be used together in a single accelerator. However, even with these simple rules for deciding which parts are executed as pipeline and which as basic blocks, the extraction achieved an improvement of the runtime for some benchmarks.

With blowfish from the CHStone suite, an example for an unbalanced pipeline can be shown. With an unlimited number of single-cycle memory accesses in the pipelined model, blowfish takes 320320 clock cycles to execute. However, with the basic block model it only takes 101659 clock cycles. The cause for this is again an unbalanced pipeline. In the code of the main loop of blowfish (Figure 9.14), the part shown in red is only executed once during the first iteration of the loop. However, in the pipelined model this part is included in the overall scheduling of the loop and leads to a pipeline with an $II = 58$. In the basic block model, this block has a length of 56 while the other remaining block only has a length of 6. Even with the overhead of the emulated basic block model, this huge imbalance between both blocks causes the $\sim$ 66% reduction in runtime for the basic block model compared to the pipelined model.

The basic block extraction achieves a similar result of 80860 clock cycles. The basic block extraction achieves a better runtime because it uses the pipelined model and just extracts the unbalanced (red) block.

The extraction variant uses a heuristic to decide which basic block should be extracted into its own single-iteration loop. The heuristic combines a profile based execution count for each basic block with a simple size estimation (number of instructions in the LLVM-IR). First, the heuristic checks if the basic block has at least N instructions or at least a single floating point MCO. Without this prerequisite, a basic block will never be extracted as it assumed that the resulting single iteration loop would not be shorter (scheduling wise) than the VLO representing the new loop. When the minimum size is exceeded, then the heuristic uses the execution count profile to determine whether the basic block is executed seldom enough to justify the extraction. This threshold is defined by Q where a basic block with the execution count EC_{block} is extracted when $EC_{block} \times Q \geq EC_{parent}$.

Figure 9.15 shows the runtime for different thresholds Q for the basic block extraction relative to not extracting any basic block. df_add_mod reaches the worst case of being 7.25 (cropped in the figure) times slower that the version without extracted basic blocks. For $Q = 1$ the extraction is to aggressive and introduces to many small basic blocks. The same happens with df_add, however, as the unmodified version has many context switches, the context switch overhead hides much of the difference to the other values of Q.

The curious case that, when using cached memory accesses (Figure 9.15a), some benchmarks achieve better relative performance than a single thread with single-cycle memory accesses can be explained again by the problem with the static reordering priority. In some benchmarks one of the threads is starved which is solved by accident through the basic block extraction.

Figure 9.16 shows the runtime of the basic block model compared to the dynamic II model. However, only the outer-most loop is implemented with basic block model to prove that a combination of both models works.

9.5 Hardware Functions

The purpose of the hardware functions presented in Section 3.3.6.2 is to reduce the resource requirements of the generated accelerator. As this influences the inlining of code (function calls are either inlined or created as hardware functions), it can be assumed that it not only has an impact on the size but also on the runtime. To measure this impact, the benchmarks were synthesized and executed with either no hardware functions at all or the currently most aggressive setting (all functions with at least two call-sites and 50 instructions in the LLVM IR) for creating hardware functions.

For each function in the source code, Nymble decides whether the function is inlined or executed as a hardware function. For that, Nymble analyses the number of call-sites in the hardware part and the number of instructions as a size estimate for each function. The size estimate is transitive. This means that functions calls itself are treated as the size of called function. If a function has at least two call-sites it is a candidate for a hardware function.

If a configurable minimum threshold for both number of call-sites and estimated size is exceeded, the function is implemented as a hardware function. Otherwise, it will be inlined at call-sites in the hardware part. This can lead to a degradation of pipeline performance as the stage in which the VLO for the hardware function is placed has to be stalled until the hardware function is finished.

Figure 9.17 shows the resulting area in LUTs and Slices and runtime when creating hardware functions for the earlier explained aggressive settings. The results are shown relative to using no hardware functions at all. The top of each column signifies the maximum number of call-sites for a single function.

Only for a small number of benchmarks in the CHStone suite were functions identified (functions with at least two call-sites and 50 instructions in the LLVM IR) by Nymble as suitable hardware functions. In MachSuite, most benchmarks are implemented as code without function calls. The only call-site was the call of the benchmark kernel surrounded by the hardware pragmas.

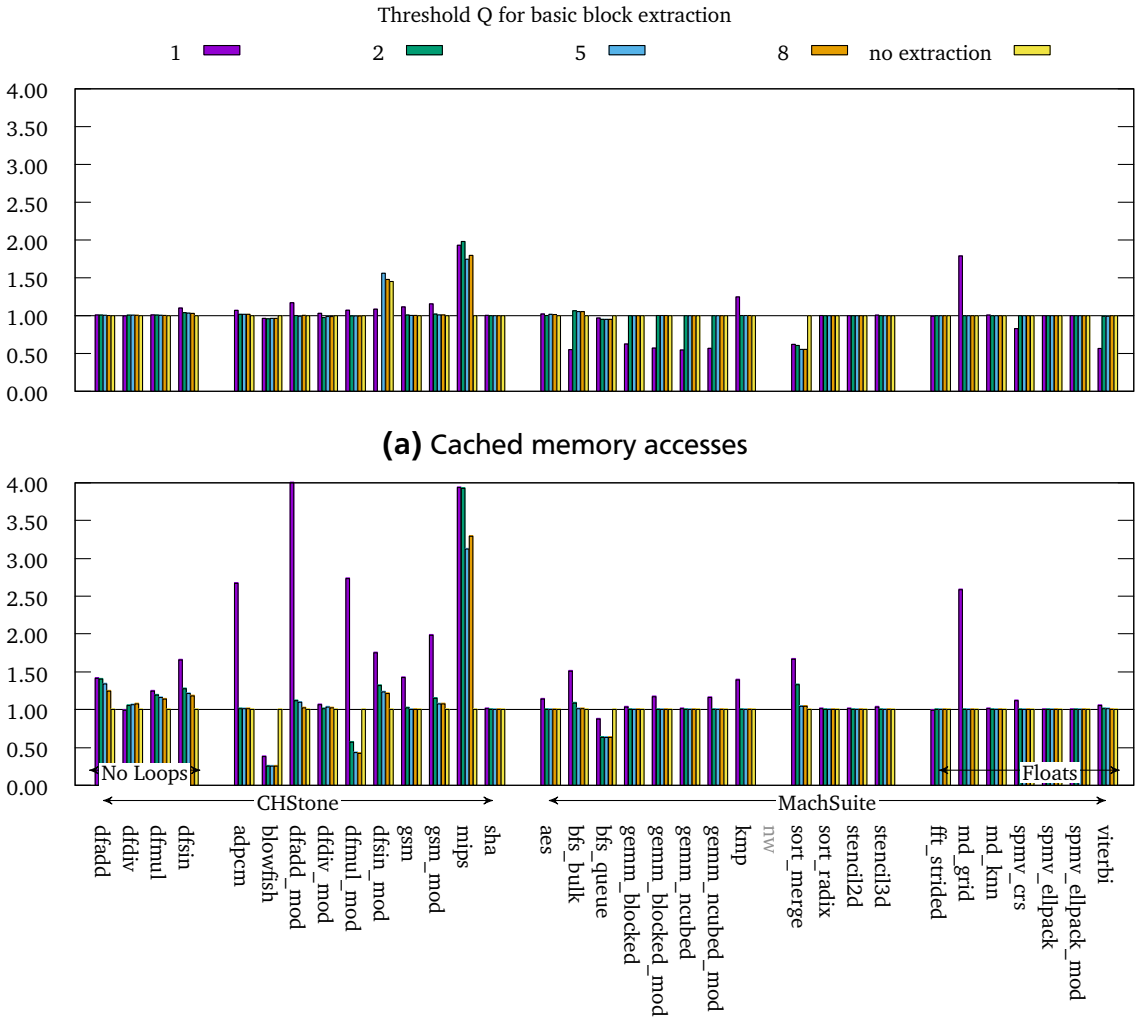
The cause for that is that only a few benchmarks have enough functions which are called from more than one call-site, which can be seen on the top of Fig-

```
void
BF_cfb64_encrypt (in, out, length, ivec, num, encrypt)
     unsigned char *in;
     unsigned char *out;
     long length;
     unsigned char *ivec;
     int *num;
     int encrypt;
{
  register BF_LONG v0, v1, t;
  register int n;
  register long l;
  BF_LONG ti[2];
  unsigned char *iv, c, cc;

  n = *num;
  l = length;
  iv = (unsigned char *) ivec;
  if (encrypt)
    {
      while (l--)
  {
    if (n == 0) // executed only once per accelerator invocation
      {         // unbalances pipeline
        n2l (iv, v0);
        ti[0] = v0;
        n2l (iv, v1);
        ti[1] = v1;
        BF_encrypt ((unsigned long *) ti, BF_ENCRYPT);
        iv = (unsigned char *) ivec;
        t = ti[0];
        l2n (t, iv);
        t = ti[1];
        l2n (t, iv);

        iv = (unsigned char *) ivec;
      }
    c = *(in++) ^ iv[n];
    *(out++) = c;
    iv[n] = c;
    n = (n + 1) & 0x07;
  }
    }
```

Figure 9.14: Source code for blowfish's main loop
(red part is executed only in the first loop iteration)



(a) Cached memory accesses

(b) Single thread with unlimited single-cycle memory accesses

Figure 9.15: Relative runtime compared to no basic block extraction (benchmarks with grey name are missing because of compilation errors)

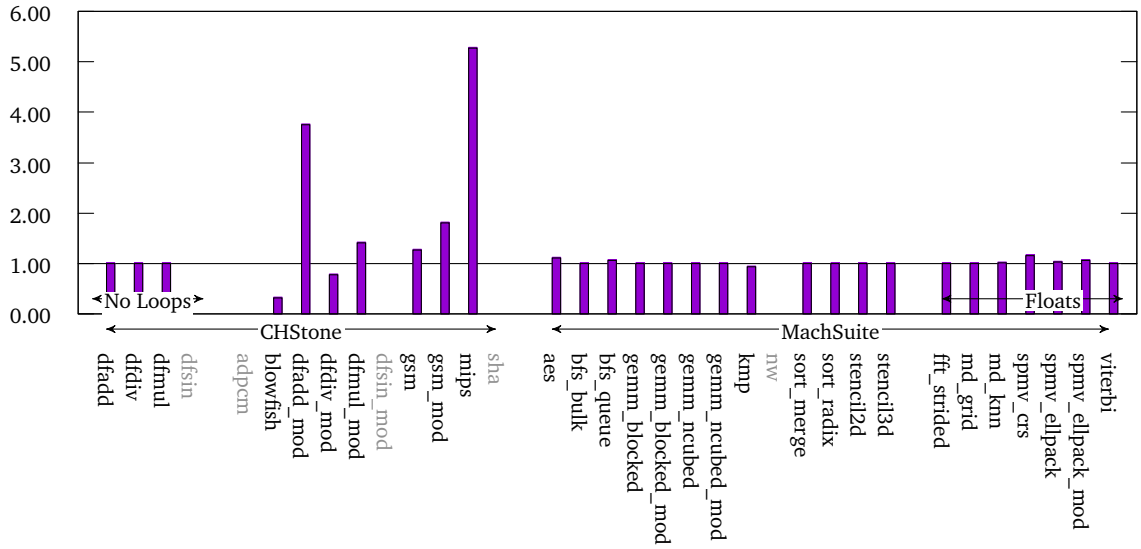


Figure 9.16: Relative runtime compared of basic block model (used for outer loop) to dynamic II model (single thread with unlimited memory accesses) (benchmarks with grey name are missing because of compilation errors)

ure 9.17. Only with at least two call-sites will a function be considered as a hardware function. The compiler currently cannot detect equivalent code to generate more fine-granular hardware functions. Additionally, the compiler only estimates the resulting size of the hardware function.

Even with these, most aggressive settings (all functions with at least two call-sites and and 50 instructions in the LLVM IR) for the hardware function selection, only a few benchmarks provide the opportunity for the use of hardware functions at all. These are adpcm, aes, and sha from CHStone and aes and sort_merge from MachSuite.

In most of these benchmarks, hardware functions reduce the area by 83% compared to the implementation without hardware functions, increasing the runtime only slightly (sha, aes, and sort_merge). However, for other benchmarks hardware functions increase the runtime by up to 75% with only a slight improvement of the area (dfsin_mod). The increase of the runtime results from the loss of pipelining as hardware functions (and inner loops in general) are currently not pipelined from the view of the parent loop. In extreme cases it can also happen that both the area and runtime becomes worse with hardware functions (adpcm). In addition to the loss of pipelining, the overhead of the hardware function multiplexing logic can become to great compared to logic of the function itself. This points to a suboptimal heuristic for the size estimation of the hardware functions.

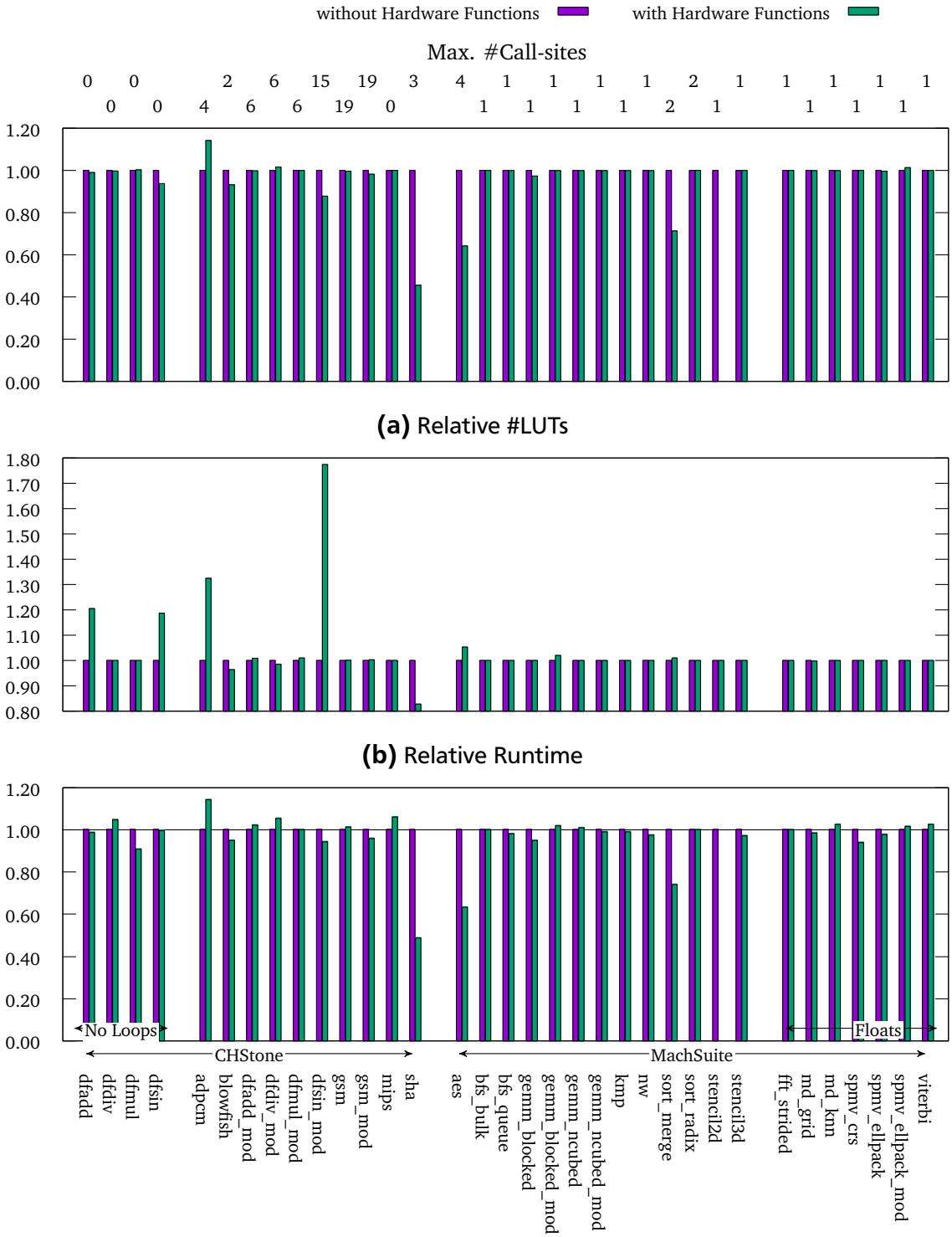


Figure 9.17: Hardware functions

9.6 Clock Frequency

To measure the maximum clock frequency for each benchmark, only the datapath and cache arbiters were synthesized as explained in Section 9.1.1 and shown in Figure 9.4a. To prove that the whole concept works, at least one variant (all optional multi-threaded stages with or without additional queues) of each of the presented benchmarks were executed in actual hardware on the DINI system (see Section 7.2). However, because current Intel CPUs have a problem with caching PCIe mapped memory, the CPU cache had to be disabled (see Section 7.2). Because of this, no performance evaluation of the entire DINI system was done.

The disabled cache slows down the software part of the application so much that each hardware thread is already finished before another thread could be started. To test the parallel execution of multiple threads, an extra module in hardware kernels stalls all threads on the very first pipeline stage until all threads have entered the hardware. This stall is then removed by staggering the first iteration of each thread by 16 clock cycles.

Initial starts of each thread are synchronized through an extra module in the hardware kernel.

All benchmarks executed on the DINI platform achieved at least 100 MHz, which currently is the only supported frequency of the DINI platform. Because of a bitstream only implementation of the PCIe core, it was not possible to change the frequency. A custom PCIe core is out-of-scope for this work.

The Convey platform has the same situation with two provided frequencies (125 MHz and 250 MHz). While some benchmarks could achieve 125 MHz, none were able to reach 250 MHz on the Convey platform. Because of the more complicated memory system, a more sophisticated configuration of place and route constraints would have been necessary to allow more benchmarks to achieve at least 125 MHz on the Convey platform.

For the actual measurement, each benchmark was synthesized with 100, 125, 150, 175, and 200 MHz target frequency. Table 9.6 shows for each benchmark the maximum frequency fully synthesized without violating any timing constraints.

9.7 Comparison with other HLS compilers

In this section Nymble will be compared to other HLS compilers. A study of different academic HLS compilers [Nan+16] provides resource and runtime measurements for the CHStone suite. These measurements were provided for Bambu, DWARV, and LegUp and a commercial HLS compiler. However, as was shown earlier, CHStone provides no benchmarks with floating point operations. Fortu-

Benchmark	Frequency
dfadd	125
dfdiv	125
dfmul	150
dfsin	100
dfadd_mod	125
dfdiv_mod	125
dfmul_mod	150
dfsin_mod	100
adpcm	100
blowfish	150
gsm	100
gsm_mod	125
mips	125
sha	125

(a) CHStone

	Benchmark	Frequency
integer only	aes	100
	bfs_bulk	175
	bfs_queue	175
	gemm_blocked	150
	gemm_blocked_mod[1]	125
	gemm_ncubed	150
	gemm_ncubed_mod[2]	125
	kmp	175
	nw	150
	sort_merge	150
	sort_radix	125
	stencil2d	125
	stencil3d	150
floating point	backprop	100
	fft_strided	175
	md_grid	125
	md_knn	150
	spmv_crs	200
	spmv_ellpack	175
	spmv_ellpack_mod[3]	175
	viterbi	200

[1] test harness modified to remove false inter-iteration dependencies
[2] inner loop unrolled eight times
[3] expression tree reordering (changes result because of FP normalization)

(a) MachSuite

Table 9.6: Maximum clock frequency for multi-threaded kernel-only implementations with four threads

nately, another set benchmark results for MachSuite (containing floating point operations) could be obtained from Bambu's source code distribution[3] [FLF16].

For the CHStone evaluation, the study measured the resources in #LUTs for Nymble, Bambu, and DWARV and in #ALMs for LegUp. For MachSuite, the resources were measured in #LUTs, #Registers, and #Slices. The runtime for both was measured in #clock cycles.

CHStone

Table 9.7 shows the resources used by kernels generated with Nymble (static single-threaded model, see Section 3.2) compared to Bambu, DWARV, and LegUp. The static single-threaded model is the baseline for the efficiency evaluation in the previous sections. While Nymble's kernels generally re-

[3] in file `examples/MachSuite/xilinx_synth.tex`

quire the most resources, there are some benchmarks (adpcm_encode, motion, and sobel) where Nymble uses less resources than at least one of the compared compilers. Overall, Nymble generally requires up to 2x more resources than the worst of the other compilers[4].

The overhead of the Nymble-created microarchitectures compared to the much smaller ones created by the competitive HLS systems is due to Nymble being able to handle variable-latency memory accesses. The other HLS systems assume only fixed memory access latencies which is far too simplistic for practical heterogeneous reconfigurable computing.

MachSuite

Table 9.8 shows a comparison of the area requirements of kernels generated by Bambu and Nymble's multi-threaded model. To fairly compare the four-threaded kernels of Nymble with the single-threaded kernels of Bambu, the resource requirements of Bambu were multiplied by four. For the runtime obtained using the simulated local memory (Section 9.1.3), different maximum limits (1, 2, 4, and unlimited) of parallel memory accesses were evaluated to show that, even limited to a small number of parallel accesses, the runtime generally does drastically increase. The limit does differentiate between writing and reading memory accesses.

In the results it can be seen that Nymble's four-threaded kernels can be more area efficient than four copies of kernels generated by Bambu (fft_strided ,md_knn, and spmv_ellpack). While not all benchmarks are more efficient with the multi-threaded model, the results are still in a comparable range of at most 3x. Again, remember that all Nymble-generated kernels carry the overhead of being able to interface with DRAM-based external memories.

The runtime of four threads compared to four consecutive sequential executions of the single-threaded implementation from Bambu is shown in Table 9.9. Also, Table 9.10 shows the runtime of a single-thread in Nymble's dynamic II model compared to a single execution of the kernel generated by Bambu. In both tables, the runtime is first shown as clock cycles and then as wall-clock time. The wall-clock time of Nymble was calculated by using the maximum frequency for the specific benchmark shown in Section 9.6, turning it into cycle time and multiplying it with the number of clock cycles. For Bambu, the maximum frequency shown in Table 9.8 was used. For Nymble, the runtime is shown for a different number of maximum allowed simultaneous memory accesses in the simulated local memory (Section 9.1.3). As these benchmarks also use single cycle memory accesses, the number specifies how many memory accesses can be executed in a single clock cycle. If

[4] The only outlier here is aes, where Nymble uses much more LUTs than all other compilers.

more memory operations are requesting a memory access, each additional access is stalled until the next cycle.

Note that all of these comparisons are somewhat "apples-to-oranges". Bambu, DWARV, and LegUp create hardware kernels that are much smaller and faster than Nymble. But they assume that all data reside in SRAM-like memories with very short (1-2 cycles) access latency. Nymble uses a far more complex microarchitecture to also handle realistic (DRAM-based, possibly cached) external memories. The price for this capability are overheads in execution time and area.

	Xilinx LUTs			Altera ALMs
Benchmark	Nymble	Bambu	DWARV	LegUp
adpcm_encode	16,727	19,931	5,626	2,490
aes_decrypt	70,484	8,747	12,733	4,297
aes_encrypt	64,427	8,485	15,699	4,263
bellmanford	1,290	1,046	633	493
blowfish	7,935	6,837	7,739	1,679
dfadd	8,853	7,250	7,334	2,812
dfadd_mod	8,516	7,250[1]	7,334[1]	2,812[1]
dfdiv	19,793	11,757	13,934	4,679
dfdiv_mod	20,554	11,757[1]	13,934[1]	4,679[1]
dfmul	4,811	3,430	14,157	1,464
dfmul_mod	5,646	3,430[1]	14,157[1]	1,464[1]
dfsin	40,751	21,892	30,616	9,099
dfsin_mod	41,253	21,892[1]	30,616[1]	9,099[1]
gsm	21,448	11,864	6,442	4,311
gsm_mod	17,085	11,864[1]	6,442[1]	4,311[1]
jpeg	110,847	46,757	NA	16,276
matrix	646	551	471	225
mips	6,282	2,501	3,904	1,319
satd	4,159	4,425	1,411	2,004
sha	20,016	4,213	10,012	6,398
sobel	2,651	3,106	1,160	1,241

[1] value copied from unmodified version, modifications should have no impact on basic block model.

Table 9.7: Nymble (static single-threaded model) compared with single-threaded area results from [Nan+16] (Table VI: Standard-Optimization Area Results)

Benchmark	LUTs Bambu 4x	LUTs Nymble MT	Regs Bambu 4x	Regs Nymble MT	Slices Bambu 4x	Slices Nymble MT	Clock Freq. Bambu
aes	30,416	56,675	14,676	29,839	9,028	23,430	180
bfs_bulk	2,496	10,557	2,384	5,907	1,084	3,503	254
bfs_queue	2,796	12,186	2,288	7,390	1,168	3,975	252
gemm_blocked	5,508	12,110	5,156	6,756	2,256	3,734	121
gemm_blocked_mod[1]	4,788	12,044	4,740	9,687	1,924	4,455	118
gemm_ncubed	1,256	5,666	1,288	3,357	568	2,035	300
gemm_ncubed_mod[2]	3,752	9,477	3,624	5,547	1,672	3,184	115
kmp	4,992	8,772	3,440	6,148	1,548	3,071	214
nw	5,556	16,656	4,200	9,516	1,920	5,524	173
sort_merge	7,916	14,682	4,280	7,064	2,432	4,894	173
sort_radix	20,856	43,338	17,780	23,012	7,372	13,653	133
stencil2d	8,336	10,428	8,292	9,153	3,376	3,888	117
stencil3d	2,448	7,552	1,976	6,170	872	2,914	124
fft_strided	26,224	18,728	10,408	15,100	7,948	6,315	101
md_grid	67,760	77,478	30,496	46,855	21,988	23,812	87
md_knn	62,220	41,474	25,824	31,218	19,504	13,391	83
spmv_crs	9,560	9,693	3,772	7,516	3,056	3,450	103
spmv_ellpack	47,036	30,388	23,432	32,185	15,372	11,124	102
spmv_ellpack_mod[3]	47,148	28,715	21,964	30,110	15,668	10,314	99
viterbi	2,944	7,120	2,068	6,336	1,164	2,728	109

[1] test harness modified to remove false inter-iteration dependencies
[2] inner loop unrolled eight times
[3] expression tree reordering (changes result because of FP normalization)

Table 9.8: Resource requirements comparison for four threads between Bambu (four accelerator instances) and Nymble (four threads in single instance)

	Benchmark	Runtime Nymble MT Max. # parallel mem. accesses 1	2	4	∞	Bambu 4x
#clock cycles	aes	10,858	7,802	6,002	3,713	8,632
	bfs_bulk	86,769	86,427	86,427	86,427	97,044
	bfs_queue	48,582	48,582	47,953	48,303	93,060
	fft_strided	257,315	239,865	235,769	236,059	356,440
	gemm_blocked	2,696,545	2,632,033	2,359,711	2,178,403	3,555,628
	gemm_blocked_mod[1]	2,845,409	1,569,275	932,049	404,197	3,572,016
	gemm_ncubed	4,651,482	2,922,288	2,922,288	2,922,288	2,130,188
	gemm_ncubed_mod[2]	2,365,872	1,317,168	804,198	439,216	1,229,072
	kmp	426,032	426,042	426,030	426,030	783,048
	md_grid	2,051,973	1,945,682	1,574,862	1,554,158	2,610,536
	md_knn	193,508	186,776	185,716	185,716	645,144
	nw	397,631	266,669	201,239	136,939	464,920
	sort_merge	1,291,522	1,133,560	1,133,560	1,133,560	1,245,228
	sort_radix	1,565,238	1,271,850	939,148	924,920	2,443,048
	spmv_crs	61,695	59,729	56,832	56,832	59,256
	spmv_ellpack	66,381	58,958	55,500	52,502	73,124
	spmv_ellpack_mod[3]	41,676	31,788	29,084	25,366	51,396
	stencil2d	593,816	313,700	190,066	63,556	255,544
	stencil3d	403,346	252,161	151,837	169,155	306,144
	viterbi	6,828,622	6,308,422	6,048,322	6,048,322	3,232,136
wall-clock (ms)[4]	aes	0.109	0.078	0.060	0.037	0.048
	bfs_bulk	0.496	0.494	0.494	0.494	0.383
	bfs_queue	0.278	0.278	0.274	0.276	0.369
	fft_strided	1.470	1.371	1.347	1.349	3.546
	gemm_blocked	17.977	17.547	15.731	14.523	29.458
	gemm_blocked_mod[1]	22.763	12.554	7.456	3.234	30.307
	gemm_ncubed	31.010	19.482	19.482	19.482	7.106
	gemm_ncubed_mod[2]	18.927	10.537	6.434	3.514	10.651
	kmp	2.434	2.435	2.434	2.434	3.661
	md_grid	16.416	15.565	12.599	12.433	29.982
	md_knn	1.290	1.245	1.238	1.238	7.777
	nw	2.651	1.778	1.342	0.913	2.691
	sort_merge	8.610	7.557	7.557	7.557	7.194
	sort_radix	12.522	10.175	7.513	7.399	18.258
	spmv_crs	0.308	0.299	0.284	0.284	0.573
	spmv_ellpack	0.379	0.337	0.317	0.300	0.717
	spmv_ellpack_mod[3]	0.278	0.212	0.194	0.169	0.515
	stencil2d	4.751	2.510	1.521	0.508	2.185
	stencil3d	2.689	1.681	1.012	1.128	2.462
	viterbi	34.143	31.542	30.242	30.242	29.571

[1] test harness modified to remove false inter-iteration dependencies
[2] inner loop unrolled eight times
[3] expression tree reordering (changes result because of FP normalization)
[4] for frequency see Table 9.6 and Table 9.8

Table 9.9: Runtime comparison for four threads between Bambu (four consecutive executions) and Nymble (four parallel threads in single instance)

	Benchmark	Runtime Nymble ST Max. # parallel mem. accesses 1	2	4	∞	Bambu
#clock cycles	aes	7,904	5,964	5,040	3,157	2,158
	bfs_bulk	31,763	31,755	31,755	31,755	24,261
	bfs_queue	48,031	48,031	47,809	47,809	23,265
	fft_strided	257,171	239,759	237,293	235,663	89,110
	gemm_blocked	1,368,689	1,335,921	1,204,849	1,106,545	888,907
	gemm_blocked_mod[1]	909,941	516,725	320,117	189,045	893,004
	gemm_ncubed	1,606,208	1,344,064	1,344,064	1,344,064	532,547
	gemm_ncubed_mod[2]	696,896	434,752	303,680	205,376	307,268
	kmp	425,765	425,765	425,763	425,763	195,762
	md_grid	1,808,589	1,659,018	1,433,708	1,415,838	652,634
	md_knn	94,788	93,764	93,252	93,252	161,286
	nw	201,553	168,785	152,401	136,017	116,230
	sort_merge	581,806	532,654	532,654	532,654	311,307
	sort_radix	1,409,412	1,173,924	899,460	885,124	608,262
	spmv_crs	30,927	29,941	28,494	28,494	14,814
	spmv_ellpack	66,262	58,852	55,394	51,936	18,281
	spmv_ellpack_mod[3]	39,092	31,682	28,718	25,460	12,849
	stencil2d	189,052	118,504	87,256	56,008	63,886
	stencil3d	171,247	133,747	108,247	95,647	76,536
	viterbi	3,441,218	3,179,074	3,048,002	3,048,002	808,034
wall-clock (ms)[4]	aes	0.079	0.060	0.050	0.032	0.012
	bfs_bulk	0.182	0.181	0.181	0.181	0.096
	bfs_queue	0.274	0.274	0.273	0.273	0.092
	fft_strided	1.470	1.370	1.356	1.347	0.887
	gemm_blocked	9.125	8.906	8.032	7.377	7.365
	gemm_blocked_mod[1]	7.280	4.134	2.561	1.512	7.579
	gemm_ncubed	10.708	8.960	8.960	8.960	1.777
	gemm_ncubed_mod[2]	5.575	3.478	2.429	1.643	2.663
	kmp	2.433	2.433	2.433	2.433	0.915
	md_grid	14.469	13.272	11.470	11.327	7.496
	md_knn	0.632	0.625	0.622	0.622	1.944
	nw	1.344	1.125	1.016	0.907	0.673
	sort_merge	3.879	3.551	3.551	3.551	1.798
	sort_radix	11.275	9.391	7.196	7.081	4.564
	spmv_crs	0.155	0.150	0.142	0.142	0.143
	spmv_ellpack	0.379	0.336	0.317	0.297	0.179
	spmv_ellpack_mod[3]	0.261	0.211	0.191	0.170	0.129
	stencil2d	1.512	0.948	0.698	0.448	0.546
	stencil3d	1.142	0.892	0.722	0.638	0.615
	viterbi	17.206	15.895	15.240	15.240	7.393

[1] test harness modified to remove false inter-iteration dependencies
[2] inner loop unrolled eight times
[3] expression tree reordering (changes result because of FP normalization)
[4] for frequency see Table 9.6 and Table 9.8

Table 9.10: Runtime comparison between Bambu and Nymble with a single thread

9.8 In-Depth Analysis

During the evaluation it was noticed that the basic block model used in other compilers sometimes performs better than the pipelined model of Nymble. A rather high II for the main loops implies an issue with the source code of these benchmarks being not particularly suitable for pipelined execution. The closer the II is to the length of the main loops, the less useful pipelining becomes.

The main cause for a high II are inter-iteration dependencies. A new iteration can only be started after all dependencies are fulfilled. Sometimes the dependencies created by the compiler's dependency analysis can be a false dependency, i.e a dependency that never actually occurs during execution. In general, false dependencies are only generated for memory accesses because the analysis for memory accesses is not sophisticated enough. Compilers expend lots of effort to analyse these memory dependencies, but if the analysis cannot 100% prove that a dependency does not exist, the compiler has to conservatively assume that it exists. This results in a number of false dependencies. And these false dependencies can have a negative impact on the II.

So for these false dependencies, an example benchmark is analysed to examine where false dependencies occur and what improvements of the II can be gained by manual prevention of these dependencies.

9.8.1 gsm

In gsm, profiling discovered that the loop shown in Figure 9.18 has the highest portion of the runtime. In the original version (shown in black and red), the loop has an II of 36 with a pipeline of length 39. A manually optimised version however achieves an II of 5 with a length of 7.

In the original version the basic alias analysis of LLVM cannot prove that the accesses to `L_ACF[k]` are independent from each other for each k. This leads to the sequential execution of each `STEP`, where a single `STEP` is scheduled to 4 cycles.

To solve this, the handling of the input and output data was modified by manually creating local values for each of the nine output values (shown in black and blue). In the unmodified version, the output values in `L_ACF` are sequentially read and written in each iteration of the main loop from and to memory. The manual localization replaces this with *one* read and *one* write for each value, at the beginning and end of the function, respectively. This modification improves the performance in two aspects:

First, the localization and removal of unnecessary memory accesses. Second, removing the memory accesses in the main loop avoids the problem that the alias analysis conservatively assumes that all values in `L_ACF` alias each other, resulting

	Nymble	LegUp
gsm	9080	4763
gsm_mod	4180	4763

Table 9.11: #Clock cycles comparison for gsm

	Nymble	Bambu
gemm_blocked	1,106,545	888,907
gemm_blocked_mod	189,045	893,004

Table 9.12: #Clock cycles comparison for gemm_blocked

in chaining the operations in each STEP sequentially, instead of executing them in parallel. Thus, the overall runtime is improved by around 50% for Nymble.

In the basic block model, however, (evaluated by using a kernel generated by LegUp), these changes have no impact on the runtime. As can be seen in Table 9.11, the pipeline model's performance is currently heavily dependent on "compatible" source code.

9.8.2 gemm_blocked

Similarly, gemm_blocked of MachSuite also has a problem with false inter-iteration dependencies. MachSuite uses a single structure to pass the test data to the test-specific `run_benchmark` function (shown in Figure 9.19a). As this structure contains both the input and output data, the alias analysis assumes that this whole structure can be accessed (and written) by all memory operations. However, gemm_blocked only writes to the array prod, so most of these dependencies are false dependencies.

To make it clear for the analysis that all arrays are independent, the function was modified by creating a working copy for each array (shown in Figure 9.19b). This reduces the II of the main loop (shown in Figure 9.19c) from 32 to 5 and reduces the pipeline length 35 from to 7. This reduces the runtime (see Table 9.12) for the Nymble generated kernel to around 17% of the unmodified version. In the basic block model, here presented by Bambu, the runtime does not change (the small difference comes from Bambu specifics and were not further analysed). Note that the absolute runtime values should not be compared between Nymble and Bambu, as the runtime for Nymble was generated with unlimited single cycle memory accesses (see Table 9.10 and Section 9.7).

```
void
Autocorrelation (word * s /* [0..159]      IN/OUT  */ ,
longword * L_ACF /* [0..8]        OUT      */ )
/*
*  The goal is to compute the array L_ACF[k].  The signal s[i] must
*  be scaled in order to avoid an overflow situation.
*/
{
  register int k, i;

  longword L_ACF0,L_ACF1,L_ACF2,L_ACF3,L_ACF4,L_ACF5,L_ACF6,L_ACF7,
      L_ACF8; /* Temporary registers for L_ACF */
  word temp;
  word smax;
  word scalauto, n;
  word *sp;
  word sl;

  for (k = 8; k >= 0; k--)
    L_ACF[k] = 0;
  L_ACF0 = L_ACF1 = L_ACF2 = L_ACF3 = L_ACF4 = L_ACF5 = L_ACF6 = L_ACF7
      = L_ACF8 = 0;

  #define STEP(k)  L_ACF[k] += ((longword)sl * sp[ -(k) ]);
  #define STEP(k)  L_ACF##k += ((longword)sl * sp[ -(k) ]);

  #define NEXTI  sl = *++sp

  ...

      for (i = 8; i <= 159; i++)
      {

      NEXTI;

      STEP (0); STEP (1); STEP (2); STEP (3); STEP (4); STEP (5); STEP
          (6); STEP (7); STEP (8);
  }

  for (k = 8; k >= 0; k--)
    L_ACF[k] <<= 1;
  L_ACF[0] = L_ACF0 << 1;
  ...
  L_ACF[8] = L_ACF8 << 1;
```

Figure 9.18: Longest runtime loop in `gsm`

```
void run_benchmark( void *vargs ) {
  struct bench_args_t *args = (struct bench_args_t *)vargs;
  bbgemm( args->m1, args->m2, args->prod );
}
```

(a) Original harness

```
void run_benchmark( void *vargs ) {
  struct bench_args_t *args = (struct bench_args_t *)vargs;
  int i;
  TYPE *m1,*m2,*prod;
  m1 = malloc(N*sizeof(TYPE));
  m2 = malloc(N*sizeof(TYPE));
  prod = malloc(N*sizeof(TYPE));
  for(i=0;i<N;i++) {
    m1[i] = args->m1[i];
    m2[i] = args->m2[i];
    prod[i] = args->prod[i];
  }
  #pragma HARDWARE on
  bbgemm( m1, m2, prod );
  #pragma HARDWARE off
  for(i=0;i<N;i++) {
    args->prod[i] = prod[i];
  }
}
```

(b) Modified harness

```
void bbgemm(TYPE m1[N], TYPE m2[N], TYPE prod[N]){
  int i, k, j, jj, kk, temp_x;
  int i_row, k_row;
  TYPE mul;

  loopjj:for (jj = 0; jj < row_size; jj += block_size)
    loopkk:for (kk = 0; kk < row_size; kk += block_size)
      loopi:for ( i = 0; i < row_size; ++i)
        loopk:for (k = 0; k < block_size; ++k) {
          i_row = i * row_size;
          k_row = (k  + kk) * row_size;
          temp_x = m1[i_row + k + kk];
          loopj:for (j = 0; j < block_size; ++j) {
            mul = temp_x * m2[k_row + j + jj];
            prod[i_row + j + jj] += mul;
          }
        }
```

(c) Loop

Figure 9.19: MachSuite harness code specific for gemm_blocked

Benchmark	#Clock cycles Nymble ST Max. # parallel mem. accesses 1	2	4	∞	Bambu
gemm_ncubed	1,606,208	1,344,064	1,344,064	1,344,064	532,547
gemm_ncubed_mod	696,896	434,752	303,680	205,376	307,268

(a) Single-threaded (single-cycle memory accesses)

Benchmark	#Clock cycles
gemm_ncubed	3,213,923
gemm_ncubed_mod	2,078,725

(b) Multi-threaded (cached memory accesses)

Table 9.13: #Clock cycles comparison for gemm_ncubed

9.8.3 gemm_ncubed

For gemm_ncubed the alias analysis does not create any false dependencies with negative impact on the runtime. However, it is still possible to optimise runtime of this benchmark, especially when considering that it will be executed with a pipelined model.

By examining the source code it was noticed that the pipeline executes only two memory accesses, a multiplication and an addition each iteration. Additionally, the inner most loop only executes eight iterations at all. As the presented pipelined model is not quite optimised for very short pipelines, a method to increase the number operations per iterations was sought.

The simple solution is to unroll the inner loop eight times. The new inner most loop now executes 16 memory accesses which utilize the pipeline more efficiently. Both the original and the unrolled version can be seen in Figure 9.20. The modifications are shown in red and blue for removed and newly added code, respectively.

Comparing single-threaded runtime (see Table 9.13), it can be easily seen that unrolling is very beneficial for the pipelined model. While it also improves the runtime of the basic block model of Bambu, it has a much bigger impact for Nymble. The table shows the runtime for different amounts of parallel single-cycle memory accesses (see Table 9.13a). Even with only a single memory access per cycle the unrolled version performs much better. From the unmodified version it can also be seen that, in fact, it does not profit from more than two parallel memory accesses. This improvement of the runtime can also be seen in the multi-threaded model (see Table 9.13b).

```
#define STEP(x)  k_col##x = (k+x) * col_size; \
                  mult##x = m1[i_col + k + x] * m2[k_col##x + j];

void gemm( TYPE m1[row_size * col_size],
           TYPE m2[row_size * col_size],
           TYPE prod[row_size * col_size]) {

int i, j, k;
TYPE mult, k_col, i_col;
int k_col0, k_col1, k_col2, k_col3, k_col4, k_col5, k_col6, k_col7;
TYPE mult0, mult1, mult2, mult3, mult4, mult5, mult6, mult7;

mult = 0;
k_col = 0;
i_col = 0;

outter:for(i=0;i<row_size;i++) {
        middle:for(j=0;j<col_size;j++) {
            i_col = i * col_size;
            TYPE sum = prod[i_col + j];
            inner:for(k=0;k<row_size;k+=8) {
            inner:for(k=0;k<row_size;k++) {
                STEP(0);
                STEP(1);
                STEP(2);
                STEP(3);
                STEP(4);
                STEP(5);
                STEP(6);
                STEP(7);
                sum += mult0 + mult1 + mult2 + mult3 + mult4 + mult5 +
                    mult6 + mult7;
                k_col = k * col_size;
                mult = m1[i_col + k] * m2[k_col + j];
                sum += mult;
            }
            prod[i_col + j]  = sum;
        }
    }
}
```

Figure 9.20: Main loops of gemm_ncubed

9.8.4 spmv_ellpack

In spmv_ellpack (see Figure 9.21) the innermost loop is unrolled completely by the compiler because it only has ten iterations. The resulting expression tree for `sum`, however, is actually a chain of multiplications. The compiler cannot create a real multiplication tree, because the benchmark uses floating point values. Because IEEE 754 compatible floating point multiplications are not associative, that means their order cannot be changed without possibly changing their result.

However, if some error for this floating point operations is acceptable, the runtime of spmv_ellpack can be significantly reduced by creating an optimised expression tree for the unrolled multiplication.

The multiplication tree of the manually unrolled loop (shown in blue) was reduced to a height of 3 instead of 9. Note that the initial read of `sum = out[i]` was also removed because the `out` array is initialized to zero anyway.

This change improves to about 0.5x of the unmodified version as can be seen in Table 9.14. This table shows the runtime of both single-threaded variants which can also be seen in Table 9.10.

Benchmark	#Clock cycles
spmv_ellpack	51,936
spmv_ellpack_mod	25,460

Table 9.14: #Clock cycles comparison for spmv_ellpack

```
#define TYPE double
#define N 494
#define L 10

#define STEP(x) Si##x = nzval[x + i*L] * vec[cols[x + i*L]]

void ellpack(TYPE nzval[N*L], int cols[N*L], TYPE vec[N], TYPE out[N])
{
  int i, j;
  TYPE Si;

  ellpack_1 : for (i=0; i<N; i++) {
    TYPE sum = out[i];
    ellpack_2 : for (j=0; j<L; j++) {
      Si = nzval[j + i*L] * vec[cols[j + i*L]];
      sum += Si;
    }
    out[i] = sum;

    STEP(0); STEP(1); STEP(2); STEP(3); STEP(4);
    STEP(5); STEP(6); STEP(7); STEP(8); STEP(9);
    out[i] = ((Si0 + Si1) + (Si2 + Si3)) + ((Si4 + Si5) + (Si6 + Si7))
        + (Si8 + Si9);
  }
}
```

Figure 9.21: Main loops of spmv_ellpack

10 Future Work

Before the conclusion of this work, this chapter will present some ideas which can improve all executions models. By improving upon the ideas of the basic block model emulation (see Chapter 5) the efficiency of the accelerator will be improved. After that, a method to handle critical regions will be shown.

10.1 Mixed Execution Models

To further improve the efficiency of the generated data-paths, the compiler will be enhanced to use different execution models for each nested loop. It was shown that it is possible to mix pipelined and FSM based loop data-paths in the hardware kernel. The next step is to integrate single-threaded static and dynamically scheduled data-paths into the overall model of nested loops. It will be possible for the compiler to select the appropriate model for each part of an application.

Based on the extraction of unbalanced pipeline elements it should be possible to execute such extracted elements as a simple pipelined pseudo MCO using the statically scheduled execution model. As an example in Figure 10.1, a kernel consisting of three nested loops is generated with three different execution models.

The outer loop is implemented with the multi-threaded FSM model because the different paths through the loop are unbalanced. The FSM model is more runtime efficient for such a loop. The middle loop is implemented as a multi-threaded pipelined loop. Because of its balanced nature, it can be efficiently pipelined. Finally, the inner loop is implemented as a statically scheduled single-threaded loop. Because it contains no VLO and has a fixed number of iterations the features of the other more complex execution model are not required. Instead of unrolling this loop, the compiler decided to treat it as a nested loop. As the number of iterations is equal for all loop executions, the multi-threading is implemented by interleaving the threads with basic C-Slow method.

10.2 Locks, Semaphores and Critical Regions

To allow the handling of critical regions the multi-threading model has to support locks or semaphores. A possible solution for that would be to use stalling facilities for VLOs. A semaphore shared between the threads would be implemented by a pair of special operations indicating the beginning and the end of the critical region.

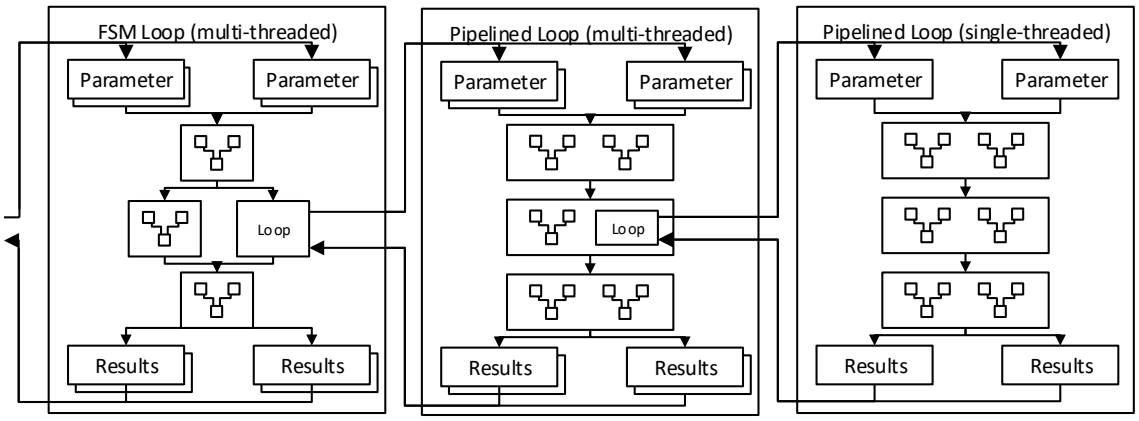


Figure 10.1: Different execution models for individual loops

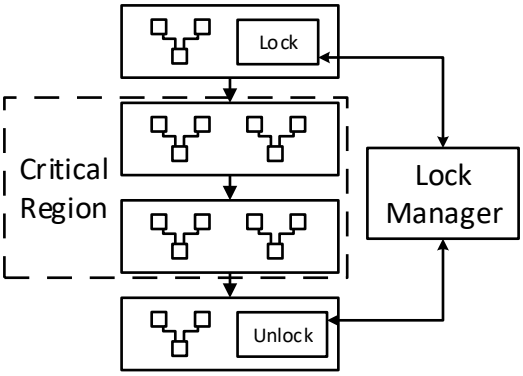


Figure 10.2: Implementing a critical region with VLOs as a lock

In Figure 10.2, just before the critical region, a *Lock* VLO tries to receive the exclusive access to critical region from a *Lock manager*. While this operation allows multiple threads to enter, only a single thread is able to continue into the next stage and thus the critical region. Another thread can only proceed when the *Unlock* VLO signals the *Lock manager* that the critical region is free again. This signal is sent when the stage after the critical region is activated by the thread currently in the critical region. All operations in the critical region are assigned by the compile-time scheduler to the stages between this pair of semaphore operations.

11 Conclusion

This work presented a novel simultaneous multi-threaded execution model for hardware accelerators. The goal of this model was to combine the following three points:

1. Temporal switching between threads to hide memory latency
2. Simultaneous execution of multiple threads to improve performance
3. Resource sharing between multiple threads to improve efficiency

Accelerators with this and other execution models can be generated by the presented Nymble HLS compiler.

It was shown that the multi-threaded model can improve resource efficiency by factor 2x compared to simply duplicating a single-threaded pipeline. At the same it can improve the throughput by factor of 3.5x compared to four consecutively executed threads.

During the extensive evaluation it was concluded, that the multi-threaded model is especially efficient for applications with floating point operations. This is due to reduced multi-threading overhead through increased operator complexity. In the worst case, a simple one bit logic operator could require two additional bits (four threads overall) for the thread identification, whereas a complex floating point operator also only requires these two bits for thread identification. From this example it can easily be seen that multi-threading can incur a higher overhead on simpler operators. Nevertheless, even for simple integer benchmarks it was shown that by selectively placing the multi-threaded stages into the accelerator pipeline, the efficiency can be improved up to a point where the multi-threaded model becomes viable even with less complex operators.

By using a concept similar to function calls in software, hardware functions allow reusing whole data-paths in the accelerator. Instead of inlining function calls, they are handled by the resource sharing method used for Variable Latency Operation (VLO). It was shown that this can improve resource efficiency with often only a small negative impact on the runtime. Only a single benchmark showed a significant degradation of the runtime with only an insignificant improvement of resource efficiency.

As was explained in the initial comparison of the most common execution models (basic block and pipeline), both have their advantages and disadvantages depending on the application. For Nymble, the focus was the pipelined execution

model. However, to show that it is possible to include the presented multi-threading into a basic block model, two different implementations of the basic block model were presented and evaluated. Because these models are not optimized, only the runtime was evaluated in comparison to the pipelined model. Only for a small set of benchmarks, the implemented basic block models achieved better performance than the pipelined model. The main problem here is, that the pipelined model which is used to execute the basic blocks is not well optimised for small basic blocks. Especially the minimum runtime of four to five clock cycles for each block is a big factor for the performance reduction.

By comparing Nymble with accelerators from other academic HLS compilers, it was shown that Nymble's single-threaded pipeline model can be as fast as the basic block model used by the other compilers. With the inclusion of multi-threading, which the model was optimised for, the accelerators can be 2x faster with two thirds of the slices when complex operators are used by the accelerator. For integer operations the multi-threading overhead is too big to compete with more optimised implementations of other works. However, the Nymble generated accelerators are within a range of 1.15x up to 3.5x more slices. It should be possible to close that gap with more optimizations for pipelines with simple operations.

Finally, multiple benchmarks were analysed in depth to determine what enhancements or optimizations the compiler requires to improve the performance of these benchmarks.

So as a final conclusion, it can be said, that the presented multi-threaded execution achieved the goals and can be further improved to achieve even better efficiency.

List of Abbreviations

ASIC Application-Specific Integrated Circuit

BB Basic Block

CDFG Control-Data-Flow-Graph

CFG Control-Flow-Graph

CPU Central Processing Unit

DFA Deterministic Finite Automaton

DFG Data-Flow-Graph

DPI Direct-Programming-Interface

DSP Digital Signal Processor

EBB Execution Basic Block

FPGA Field Programmable Gate Array

FSM Finite State Machine

HLS High-Level-Synthesis

HTS Hardware Thread Scheduler

II Initiation Interval

IR Intermediate Representation

MCO Multi-Cycle Operation

MDG Memory-Dependency-Graph

NoC Network on Chip

SSA Static Single Assignment

SWS Software Service

SWS Software Service

TAC Three Address Code

TCS Thread Context Storage

TID Thread Identifier

VLO Variable Latency Operation

Bibliography

[ACE03] ACE Associated Compiler Experts. "CoSy Compilers: Overview of Construction and Operation". In: *International Business* (2003).

[All70] Frances E. Allen. "Control flow analysis". In: *Proceedings of a symposium on Compiler optimization* -. New York, New York, USA: ACM Press, 1970, pp. 1–19.

[Bac78] John Backus. "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs". In: *Communications of the ACM* 21.8 (1978), pp. 613–641.

[Cam91] Raul Camposano. "Path-Based Scheduling for Synthesis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10.1 (1991), pp. 85–93.

[Can+13a] Andrew Canis et al. "From software to accelerators with LegUp high-level synthesis". In: *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. Vol. 13. 2. IEEE, 2013, pp. 1–9.

[Can+13b] Andrew Canis et al. "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems". In: *ACM Transactions on Embedded Computing Systems* 13.2 (2013), pp. 1–27.

[Cas+15] Vito Giovanni Castellana et al. "High level synthesis of RDF queries for graph analytics". In: *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2015, pp. 323–330.

[CBA13] Jongsok Choi, Stephen Brown, and Jason Anderson. "From Software Threads to Parallel Hardware in High-Level Synthesis for FPGAs". In: *2013 International Conference on Field-Programmable Technology (FPT)*. Ieee, 2013, pp. 270–277.

[Con10] Convey Computer Corp. *Convey's hybrid-core technology: the HC1 and HC1ex*. Tech. rep. 2010.

[Con14] Convey Computer Corp. *Hybrid Threading Reference Manual, Version 1.0*. 2014.

[CJL05] José Gabriel F. Coutinho, Jun Jiang, and Wayne Luk. “Interleaving behavioral and cycle-accurate descriptions for reconfigurable hardware compilation”. In: *Proceedings - 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2005* 2005 (2005), pp. 245–254.

[Cyt+91] Ron Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Transactions on Programming Languages and Systems* 13.4 (1991), pp. 451–490.

[DM98] Leonardo Dagum and Ramesh Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.

[Den+74] R.H. Dennard et al. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[Den80] Jack B. Dennis. “Data Flow Supercomputers.” In: *Computer* 13.11 (1980), pp. 48–56.

[DP11] Florent De Dinechin and Bogdan Pasca. “Custom arithmetic datapath design for FPGAs using the FloPoCo core generator”. In: *Design & Test of Computers, IEEE* (2011), pp. 1–6.

[Dzi+06] P. Dziurzanski et al. “A system for transforming an ANSI C code with OpenMP directives into a SystemC description”. In: *2006 IEEE Design and Diagnostics of Electronic Circuits and systems* 2006.April (2006), pp. 151–152.

[FLF16] Fabrizio Ferrandi, Marco Lattuada, and Pietro Fezzardi. *Panda Release 0.9.4*. 2016.

[Fil+14] Antonio Filgueras et al. “OmpSs@Zynq all-programmable SoC ecosystem”. In: *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays - FPGA ’14*. New York, New York, USA: ACM Press, 2014, pp. 137–146.

[GL11] Hagen Gädke-Lütjens. “Dynamic Scheduling in High-Level Compilation for Adaptive Computers”. Dissertation. Technical University Braunschweig, 2011, p. 242.

[GNB08] Zhi Guo, Walid Najjar, and Betul Buyukkurt. “Efficient hardware code generation for FPGAs”. In: *ACM Trans. Archit. Code Optim.* 5.1 (2008), 6:1–6:26.

[HN13] Robert J. Halstead and Walid Najjar. “Compiled multithreaded data paths on FPGAs for dynamic workloads”. In: *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE, 2013, pp. 1–10.

[HK15] Jens Huthmann and Andreas Koch. “Optimized high-level synthesis of SMT multi-threaded hardware accelerators”. In: *2015 International Conference on Field Programmable Technology (FPT)*. IEEE, 2015, pp. 176–183.

[HOK14] Jens Huthmann, Julian Oppermann, and Andreas Koch. “Automatic high-level synthesis of multi-threaded hardware accelerators”. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 2014, pp. 1–4.

[Hut+10a] Jens Huthmann et al. “Accelerating high-level engineering computations by automatic compilation of Geometric Algebra to hardware accelerators”. In: *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. IEEE, 2010, pp. 216–222.

[Hut+10b] Jens Huthmann et al. “Accelerating high-level engineering computations by automatic compilation of Geometric Algebra to hardware accelerators”. In: *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. IEEE, 2010, pp. 216–222.

[Hut+10c] Jens Huthmann et al. “Compiling Geometric Algebra Computations into Reconfigurable Hardware Accelerators”. In: *Dynamically Reconfigurable Architectures*. Ed. by Peter M Athanas et al. Dagstuhl Seminar Proceedings 10281. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010.

[Hut+13] Jens Huthmann et al. “Hardware/software co-compilation with the Nymble system”. In: *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, 2013, pp. 1–8.

[LK10] Holger Lange and Andreas Koch. “Architectures and Execution Models for Hardware/Software Compilation and Their System-Level Realization”. In: *IEEE Transactions on Computers* 59.10 (2010), pp. 1363–1377.

[LWK11] Holger Lange, Thorsten Wink, and Andreas Koch. “MARC II: A parametrized speculative multi-ported memory subsystem for reconfigurable computers”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. Marc I. IEEE, 2011, pp. 1–6.

[LA04] Chris Lattner and Vikram S Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Cgo* c (2004), pp. 75–86.

[LRS83] Charles E. Leiserson, Flavio M. Rose, and James B. Saxe. “Optimizing Synchronous Circuitry by Retiming”. In: *Third Caltech Conference on Very Large Scale Integration*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 87–116.

[LS91] Charles E Leiserson and James B Saxe. “Retiming synchronous circuitry”. In: *Algorithmica* 6.1 (1991), pp. 5–35.

[LNW06] Y. Y. Leow, C. Y. Ng, and W. F. Wong. “Generating hardware from OpenMP programs”. In: *Proceedings - 2006 IEEE International Conference on Field Programmable Technology, FPT 2006* (2006), pp. 73–80.

[LK16] Björn Liebig and Andreas Koch. “High-Level Synthesis of Resource-Shared Microarchitectures from Irregular Complex C-Code”. In: *International Conference on Field-Programmable Technology (FPT)*. 2016.

[Luk+09] W. Luk et al. “A high-level compilation toolchain for heterogeneous systems”. In: *Proceedings - IEEE International SOC Conference, SOCC 2009* (2009), pp. 9–18.

[Min+15] M Minutoli et al. “Inter-procedural resource sharing in High Level Synthesis through function proxies”. In: *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on* (2015), pp. 1–8.

[Min+16] Marco Minutoli et al. “Efficient synthesis of graph methods”. In: *Proceedings of the 35th International Conference on Computer-Aided Design - ICCAD ’16* (2016), pp. 1–8.

[Moo06] Gordon E Moore. “Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.” In: *IEEE Solid-State Circuits Newsletter* 20.3 (2006), pp. 33–35.

[Nan+12] Razvan Nane et al. “DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler”. In: *Proceedings - 22nd International Conference on Field Programmable Logic and Applications, FPL 2012*. 2012, pp. 619–622.

[Nan+14] Razvan Nane et al. “High-Level Synthesis in the Delft Workbench Hardware/Software Co-design Tool-Chain”. In: *Proceedings - 2014 International Conference on Embedded and Ubiquitous Computing, EUC 2014* (2014), pp. 138–145.

[Nan+16] Razvan Nane et al. “A Survey and Evaluation of FPGA High-Level Synthesis Tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* X.DECEMBER (2016), pp. 1–1.

[PF12] C Pilato and F Ferrandi. “Bambu: A Free Framework for the High Level Synthesis of Complex Applications”. In: *University Booth of DATE* 29.6 (2012), p. 2011.

[Pod14] Artur Podobas. “Accelerating parallel computations with OpenMP-driven System-on-Chip generation for FPGAs”. In: *Proceedings - 2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs, MCSoC 2014* (2014), pp. 149–156.

[Rau94] B. Ramakrishna Rau. “Iterative modulo scheduling”. In: *Proceedings of the 27th annual international symposium on Microarchitecture - MICRO 27*. New York, New York, USA: ACM Press, 1994, pp. 63–74.

[Rea+14] Brandon Reagen et al. “MachSuite: Benchmarks for Accelerator Design and Customized Architectures”. In: *IEEE International Symposium on Workload Characterization (IISWC)*. 2014.

[Rod69] Jorge E. Rodriguez. *A Graph Model for Parallel Computations*. 1969.

[SOK16] Lukas Sommer, Julian Oppermann, and Andreas Koch. “C-based Synthesis of Area-Efficient Accelerators for OpenMP Worksharing Loops”. In: *International Workshop on Heterogeneous High-performance Reconfigurable Computing*. 2016.

[SE16] Richard M. Stallman and Et.al. *Using the GNU Compiler Collection*. 2016.

[Syn11] Synopsis Inc. *Synphony C Compiler User Guide*. 2011.

[Tan+14] Mingxing Tan et al. “Multithreaded pipeline synthesis for data-parallel kernels”. In: *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2014), pp. 718–725.

[THK11a] Benjamin Thielmann, Jens Huthmann, and Andreas Koch. “Evaluation of speculative execution techniques for high-level language to hardware compilation”. In: *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)* (2011), pp. 1–8.

[THK11b] Benjamin Thielmann, Jens Huthmann, and Andreas Koch. “Precore - A Token-Based Speculation Architecture for High-Level Language to Hardware Compilation”. In: *2011 21st International Conference on Field Programmable Logic and Applications*. IEEE, 2011, pp. 123–129.

[THK12] Benjamin Thielmann, Jens Huthmann, and Andreas Koch. “Memory Latency Hiding by Load Value Speculation for Reconfigurable Computers”. In: *ACM Transactions on Reconfigurable Technology and Systems* 5.3 (2012), pp. 1–14.

[Thi+11] Benjamin Thielmann et al. “RAP: More Efficient Memory Access in Highly Speculative Execution on Reconfigurable Adaptive Computers”. In: *2011 International Conference on Reconfigurable Computing and FPGAs* (2011), pp. 434–441.

[Thi+12] Benjamin Thielmann et al. “Widening the Memory Bottleneck by Automatically-Compiled Application-Specific Speculation Mechanisms”. In: *Embedded Systems Design with FPGAs*. Ed. by Peter Athanas, Dionisios Pnevmatikatos, and Nicolas Sklavos. New York, NY: Springer New York, 2012, pp. 1–29.

[Vil+10] Jason Villarreal et al. “Designing modular hardware accelerators in C with ROCCC 2.0”. In: *Proceedings - IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2010* (2010), pp. 127–134.

[Xil00] Xilinx Inc. *CORE Generator Guide*. Xilinx, 2000.

[Xil14] Xilinx Inc. *Vivado Design Suite User Guide: High-Level Synthesis, UG902*. 2014.

[Y E] Y Explorations Inc. *eXCite C to RTL Behavioral Synthesis*.

[Yuk+08] Yuko Hara et al. “CHStone: A benchmark program suite for practical C-based high-level synthesis”. In: *2008 IEEE International Symposium on Circuits and Systems*. IEEE, 2008, pp. 1192–1195.

List of Figures

List of Tables