UNIVERSITY OF CALIFORNIA

Los Angeles

# An LLVM-IR Datagraph-Based Simulator for Flexible Design Space Exploration over Accelerator Architectures

A thesis submitted in partial satisfaction of the requirements for the degree Master of Science in Computer Science

by

Zhengrong Wang

2018

ABSTRACT OF THE THESIS

An LLVM-IR Datagraph-Based Simulator
for Flexible Design Space Exploration
over Accelerator Architectures

by

Zhengrong Wang
Master of Science in Computer Science
University of California, Los Angeles, 2018
Professor Anthony John Nowatzki, Chair

In this thesis, we present a novel simulation framework designed for flexible design space exploration over accelerator architectures. A conventional cycle-level simulator can bring accurate simulation results, but it requires enormous efforts to simulate the performance of architectures. Trace-based and datagraph-based simulation frameworks, which modify or argument the ISA, bring some flexibility for architecture exploration, but work at the ISA level and lose high-level information from the compiler. Our framework tries to achieve flexibility of datagraph-based simulator while maintaining high-level compiler information. The main contribution of this work can be divided into two parts. First, an LLVM-IR tracer and parser are developed to generate a datagraph at LLVM-IR level, which contains high-level compiler information and is flexible to be manipulated to reflect the desired architecture change. Second, we build an LLVM-IR datagraph simulator inside gem5 to leverage gem5's existing memory framework and provide accurate simulation results. We believe this framework would be useful for rapid design and verification of new architecture proposals.

The thesis of Zhengrong Wang is approved.

Glenn D. Reinman

Milos D. Ercegovac

Anthony John Nowatzki, Committee Chair

University of California, Los Angeles

2018

*To my parents . . .*

*who—among so many other things—*

*taught me how to be a man*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

# ACKNOWLEDGMENTS

Many years later, when I am old and grey and full of sleep, I may have forgotten many things, but not this thesis. Not because it “partial satisfies the requirements for the degree”, but because it is a summary of exciting research experience and a start for a new adventure. This thesis would not have been possible without the support and advice from many people.

Among these people, I would like to especially thank my advisor and the chair of the committee, Prof. Tony Nowatzki, for his unreserved help in my research. His rich experience in computer architecture brings many insights and points out the right direction when I lost in the details. Thank you, Tony.

Prof. Ercegovac and Prof. Reinman, thank you for serving on the committee and providing critical advice. It is a great pleasure to discuss with you.

I owe a special thank to my friends. Zhanhao, you cannot imagine how your passion and persistence have silently inspired me for years; Bisheng, Zhuolin, Geng, thank you for bringing so many laughs and unforgettable memories to my master student life; Shuaihang, thanks for putting up with my badminton skill.

Dad and Mom, I love you.

Hsien-Yu, you have no idea what your encouragement and support mean to me. I know there will be many challenges ahead waiting, but I have faith in us. Thank you, for everything.

# CHAPTER 1

# Introduction

As we are hitting the end of Dennard scaling, accelerators have been developed and integrated into conventional general purpose processors to improve the performance and energy consumption. Generally, an accelerator can be categorized in three dimensions.

- *Exposed vs. Transparent.* For an exposed accelerator, the programmer is aware of its computation model and can directly control it at the source code level. The transparent approach, however, tries to automatically identify and accelerate some common patterns in various workloads without direct intervention from the programmer.

- *Fixed Function vs. Programmable.* A fixed function accelerator is highly specialized for a particular workload or algorithm and is typically more domain specific, e.g., ASIC, while a programmable accelerator can be configured via special instructions, e.g., CGRA.

- *Core-Integrated vs. Standalone.* Some accelerators are integrated with the processor and its cache hierarchy. They usually exhibit tight coupling with the core, e.g., BERET [GFA11], DySER [GHN12], etc. Others can be more independent, using their own interface to communicate with other subsystems, e.g., GPU.

This thesis will focus on modeling a transparent programmable accelerator, which is integrated with the processor and shares its cache hierarchy. Figure 1.1 shows a diagram of our target accelerator.

The conventional workflow of developing a new transparent programmable accelerator usually starts by extending the ISA with special instructions for the accelerator. If statically

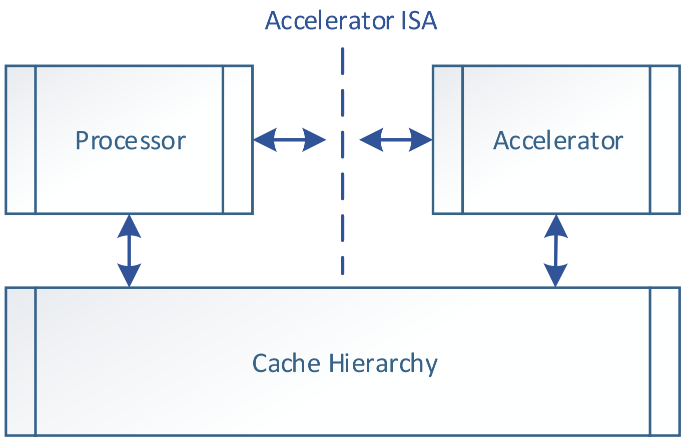


Figure 1.1: A diagram of our target accelerator.

pattern recognization is used, the compiler has to be modified to identify and transform these patterns to the extended ISA. Otherwise, pattern recognization and transformation will be integrated into general purpose processors. Developing the ISA involves many per-accelerator tasks, e.g., developing assembler, ISA encoder, instruction selector, etc., and is not flexible for design space exploration.

Compared with a conventional cycle-level simulator, datagraph-based simulation is more flexible. The datagraph is transformed to reflect the desired architecture change. An existing example is transformable dependence graph (TDG) [NGS15]. These simulators are based on $\mu$DG [FBH03], which contains low-level hardware events for accurate simulation. However, high-level compiler information is missing from the datagraph, which may limit the transformer's ability to identify more acceleration opportunities. Some other simulators are based on the LLVM-IR datagraph, e.g., Aladdin [SRW14a] and gem5-Aladdin [SRW14b]. However, Aladdin and gem5-Aladdin are for fixed-function accelerators and can not model transparent accelerators. Inspired by these frameworks, it is natural to propose the following goal:

*Goal.* Design an accurate, flexible datagraph-based simulator for cross-accelerator exploration, with full support from the compiler.

*Research Questions.* This work focuses on the development and verification of the infrastructure. However, the long-term goal of this work is to develop the tools that make answering the following research questions tractable.

1. *Domain Interaction.* What set of accelerators would be most useful in a specific domain?

2. *Accelerator/Memory Interaction.* Is it possible to specialize the interface between accelerators and memory and other systems?

3. *Novel Accelerator Design.* Does the combination of static compiler analysis and dynamic information from the datagraph provide some insights for new accelerator designs?

4. *Multi-processor Accelerator Design.* What would be the most efficient accelerator configuration for a system with multiple processors? What are the gain and cost if the accelerator is temporally multiplexed among processors?

*Proposal.* We tackle the problem in three main aspects.

- To leverage high-level information, the datagraph should be constructed and transformed within some compiler framework, instead of low-level ISA instructions or micro-operations. As a mature open-source compiler framework, LLVM provides some powerful analysis passes and is a reasonable choice.

- As the datagraph is based on LLVM-IR, which is a high-level abstraction compared with ISA instruction or micro-operation trace, there is no low-level hardware information in the datagraph itself. Thus to maintain accuracy, the simulation should not be based on the datagraph only, but take a hybrid approach – the datagraph would be simulated in a conventional simulator, gem5, to leverage its memory hierarchy and other subsystems.

- For flexibility, the framework should be extendable for new accelerator design. Considering that a major work of designing new accelerators is defining new instructions and their functionality, we decouple the node definition and execution from the datagraph transformer and simulator. Users can implement their own transformer to rewrite the datagraph and introduce new instructions. For each new instruction, the datagraph simulator will execute a user-defined execution function, which defined its functionality.

The rest of the thesis is organized as: Chapter 2 discusses the design of the whole framework and how to maintain flexibility; Chapter 3 shows the verification results, and compare with other works; Chapter 4 concludes and also discusses possible future work direction.

# CHAPTER 2

# Design

In this chapter, we explain the high-level design of the framework. Figure 2.1 shows the general workflow to use the framework:

1. The source code is instrumented by an LLVM pass and compiled to an instrumented binary. When run, this binary will produce a detailed LLVM-IR trace.

2. An LLVM-IR datagraph is constructed from the generated trace, and necessary information is gathered for replay simulation later. This step will also produce a replay binary, which is used later for replay simulation.

3. Users can transform this datagraph to reflect any architecture design they want to explore. The transformed datagraph, along with the replay binary, can be feed into our replay simulator with an integrated memory system.

4. Depending on the simulation result, a user can keep changing their architecture design and go back to step 3, as illustrated by the dashed line in Figure 2.1.

## 2.1 LLVM-IR Tracer

To construct an LLVM-IR datagraph, a trace of executed LLVM-IR instructions and the runtime value of their parameters and results must be generated. Since LLVM-IR is an intermediate representation, it can not be directly run on a host machine or a conventional architecture simulator, but in an LLVM-IR interpreter with JIT support. One possible solution is to modify the official LLVM-IR interpreter and trace every executed instruction. However, this approach has mainly two drawbacks:

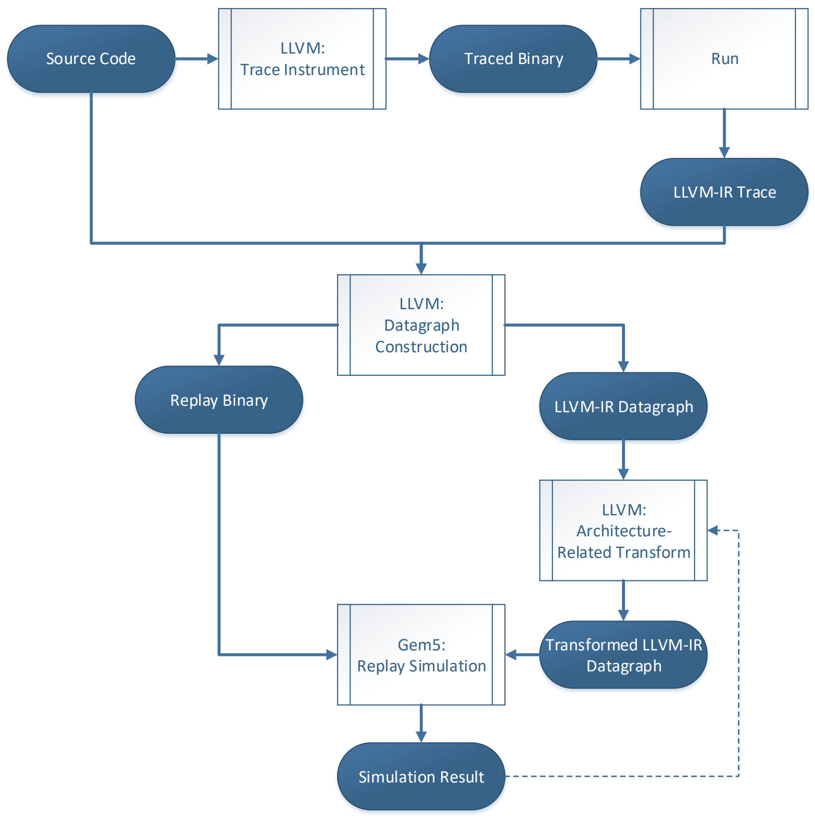


Figure 2.1: The general workflow.

- The LLVM-IR interpreter explores JIT to compile LLVM-IR to native assembly code, which will lose the fine-grained information of each LLVM-IR instruction.
- Running LLVM-IR in an interpreter directly may result in degenerated performance compared with static compiling. It may take a long time to generate traces for complicated benchmarks.

To solve these problems, we took another approach to instrument the LLVM-IR directly. The source program will first be compiled to LLVM-IR bytecode by any suitable compiler frontend, e.g., clang for C. A special LLVM optimization pass will loop through the LLVM-IR bytecode, and insert a function call to a special trace function for each LLVM-IR instruction. This special trace function will log the basic information of the traced LLVM-IR instruction, along with the runtime values of its parameters and results (if any). There are two special cases worth mentioning:

- The $\phi$ node is introduced to LLVM-IR to maintain SSA form. It will pick the parameter associated with the incoming basic block. There is no need to trace the runtime values of $\phi$ node's parameters because the incoming basic block can be determined by looking at the previous traced instruction.
- If the traced value is of a primitive type, e.g., int, float, then it can be passed to the trace function as a variadic argument. However, if it is a structure or a vector, it can not be treated as a variadic argument as it is too large to pass directly. In this case, a special slot is allocated on the stack, and the traced function will first store the value to the slot and then pass its address to the trace function. The trace function can read the value from that address. To reduce the overhead on stack allocation, the instrument pass can track the allocated slots and reuse them when there is a match.

Finally, the instrumented LLVM-IR is compiled and linked to a binary, which can be executed directly on the host machine and will produce a detailed trace. Compared with the interpreter approach, this one has better performance since the traced binary is executed directly on a host machine.

Note that even the trace is generated on one host machine, it can still be used to simulate the performance of architectures different than the host machine. This is because LLVM-IR is designed to be target-independent, and the trace of LLVM-IR on one architecture should be the same as those on other architectures.

## 2.2 LLVM-IR Datagraph Construction

It is straightforward to construct the datagraph once we have the trace. The datagraph constructor reads in the source LLVM-IR bytecode and the trace and produces an LLVM-IR datagraph for further transformation and simulation. To leverae the existing LLVM-IR framework, the constructor is implemented as an LLVM-IR optimization pass. This allows the constructor to access the full compiler information and reduces the workload of the tracer as less information is required to be traced. This also reduces the required trace size.

### 2.2.1 Handle Dependence

In the constructed datagraph, each node represents a dynamic LLVM-IR instruction. The directional edge from node $i$ to node $j$ is denoted as $i \rightarrow j$, and means that node $j$ is dependent on node $i$. There are three types of dependence needed to be tracked in the datagraph:

- Register dependence.

  LLVM-IR assumes an infinite number of registers, so here the term of "register dependence" does not mean dependence between ISA registers, e.g., $\%rax$. Instead, instruction $i$ has register dependence on instruction $j$ when it takes the output of instruction $j$ as an operand. This is trivial to handle, as it can be determined entirely during compile time with the help of LLVM framework. The only exception is the $\phi$ instruction, which should only have register dependence on the operands corresponding to the incoming basic block, and we can only collect this information from the trace.

- Memory dependence.

  For the load and store instructions, the virtual address and the size of their access are logged in the trace. During the construction of the datagraph, a load map and a store map are maintained. The load map maps each memory address to the latest dynamic load instruction which accesses that address, while the store map does the same thing for the store instruction. These two maps are used to detect read-after-write, write-after-write and write-after-read memory dependence.

- Control dependence.

  The control dependence is also trivial to handle in the LLVM framework. For each dynamic LLVM instruction $i$, we search for the latest branching dynamic LLVM instruction (`br`, `switch`, etc.) $j$ that comes before $i$. If found, we make $i$ have control dependence on $j$.

### 2.2.2 Remove $\phi$ Instruction

LLVM-IR introduces $\phi$ instructions to maintain SSA form. However, it is typically not mapped to any specific ISA instruction or does any computation, so it is reasonable to remove it from the constructed datagraph. This can be done in two steps:

1. In order to not break the dependence relationship, for a $\phi$ instruction $i$ and arbitrary instruction $j, k$, if there exists a chain of $k \to i \to j$, it is replaced by an edge $k \to j$. This will effectively remove all the edges of the $\phi$ instruction.
2. Since there is no edge between $\phi$ instructions and other instructions after step 1, now we may safely remove them from the datagraph.

### 2.2.3 Propagate Memory Base and Offset

The virtual address recorded in the trace represents the memory layout for that single run. However, the memory layout may change in the replay phase, as the trace may be replayed for architectures with different memory layout, or simply because of dynamically allocated memory. This will invalidate the virtual address in the trace, and we can not use them directly when replaying the trace.

To solve this problem, we break every virtual address in the trace into two parts: `base` and `offset`, where `base` is a string and `offset` is a integer. During replay, the simulator will maintain a map `BaseMap`, which will map `base` to the correct virtual memory address. When accessing the memory, the virtual address is computed with Equation 2.1.

$$\texttt{vaddr} = \texttt{BaseMap[base]} + \texttt{offset} \tag{2.1}$$

Here we explain how to break the virtual address into `base` and `offset` during datagraph contruction. For any virtual address in the datagraph, we initialize its `base` to an empty string and `offset` to 0. Then we fill in its `base` and `offset` depending on which instruction generates it.

- `getelementptr`

  In LLVM-IR, `getelementptr` is the most frequent instruction used to generate address. Suppose we have an instance of `getelementptr` in the trace as:

  $$\texttt{result} = \texttt{getelementptr ptr, x} \tag{2.2}$$

  Then in the datagraph, `result` will have the same `base` as `ptr`, and its `offset` is computed using the run time value of `result` and `ptr` as Equation 2.3.

  $$\texttt{result.offset} = \texttt{ptr.offset} + \texttt{result.value} - \texttt{ptr.value} \tag{2.3}$$

- `load` and `alloca`

  Special handling is required for both `load` and `alloca` instruction, as they can introduce a new `base` into the graph. For `load` instruction, if it loads an pointer value from the memory, then we set its `base` to its own name and `offset` to 0. This is crucial to support some types of indirect memory access, e.g., linked list traversal and tree traversal, since this pointer may be used to access memory later. For `alloca`, it allocates some spaces on the stack and returns the address of allocated space. We handle it the same way as `load` instruction so that the returned address can become a new base for later access.

For any other instructions, we propagate the operands' `base` and `offset` to the result if it is reasonable to do so. For example, it is reasonable to assign the operand's `base` and `offset` to the result for `bitcast` instruction, but not for `add`.

Notice that our algorithm implicitly assumes that the `offset` will remain constant for a program if the input is fixed, and only the `base` can be different due to dynamic memory allocation or different memory layout. In most cases, this is true. For example, the offset of a structure field is usually constant. However, if the programs have some time-dependent behavior for a fixed input, our framework cannot handle this and will access the wrong memory location. An example would be a program using the current time as the seed to access an array randomly. In such case, the `offset` is indeterminable from the trace. The only way to support this case is to bring in a full LLVM-IR interpreter into the simulator and compute the address during replay, which is too complicated and also loses the flexibility of datagraph-based simulation.

## 2.3 LLVM-IR Datagraph Replay

Once the datagraph is ready, it will be replayed in our datagraph simulator. To leverage the memory hierarchy of existing framework, the datagraph simulator is integrated into gem5. Figure 2.2 illustrates the whole diagram. There will be 2 CPUs working together in the system. One is a normal CPU and takes in an instrumented binary. The other one is our datagraph simulator. We need the normal CPU here because the program may link to some precompiled library. Since we do not have the source code of the library, these parts of the code cannot be traced and replayed by our framework, but can only be executed in the normal CPU. When it comes to replay a datagraph, the normal CPU will switch over to the datagraph simulator and suspend until it is done. Both CPUs will interact with the same memory hierarchy, which is essential to simulate the memory access behaviors accurately. Figure 2.3 shows the timeline of a typical transition to the accelerated region during datagraph replay.

### 2.3.1 Switch between the Normal CPU and the Datagraph Simulator

Currently, our framework supports datagraph replay at the function level. A function is defined as "replayable" if all of the functions reachable from it are traced, which means that we will have the full trace from entering this function until it returns. This ensures that the datagraph for a "replayable" function is complete and the datagraph simulator won't need help from the normal CPU in the middle of simulation.

To construct a replay binary, the body of a "replayable" function is removed, and some

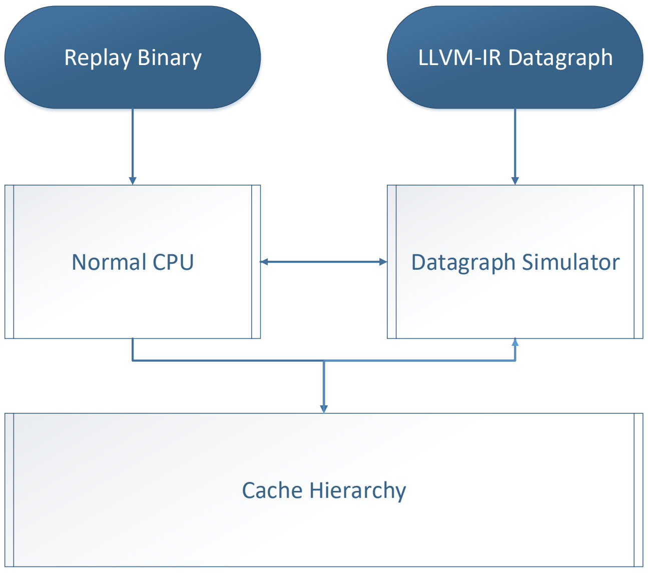


Figure 2.2: Overview of replay framework.

special instructions are inserted so that when the normal CPU enters the “replayable” function, it will switch to the datagraph simulator. One way to implement this is to insert some `ioctl` calls into the body and add a special driver in the system, which will suspend the normal CPU and activate the datagraph simulator. This is the approach taken by gem5-Aladdin [SRW14b]. However, in this approach, the overhead of switching is not negligible, as the normal CPU will have to execute the user-space `ioctl` function before it gets to the system call. Another way is to leverage gem5’s pseudo instructions, which is mapped to some unused op-code and handled specially by some user-defined function in gem5. There will still be some overhead as the normal CPU would first flush the pipeline before it executes the pseudo instruction, but it is tolerable as long as these are infrequent – which is the case in our setting. Therefore, we take the second approach in our framework.

The normal CPU will provide an execution context for the datagraph simulator, which essentially maps a `base` to some memory address. This context is used by the datagraph simulator to compute the correct memory address, as explained in section 2.2.3 and Equation 2.1.

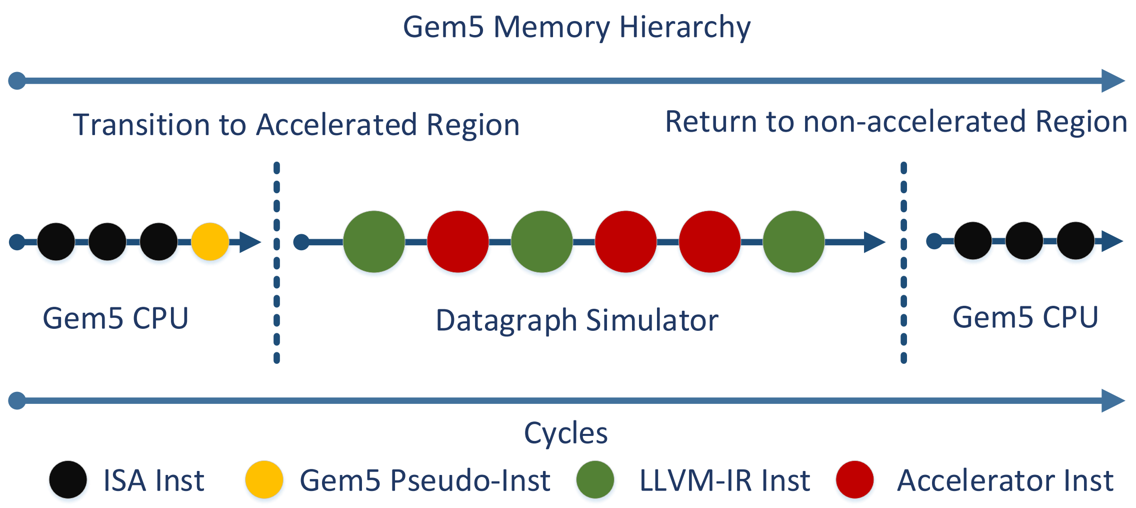


Figure 2.3: The timeline of a typical transition to accelerated region in replay.

### 2.3.2 Datagraph Simulator

Our baseline datagraph simulator emulates gem5 [BBB11] out-of-order CPU. It is also designed to be flexible for accelerator architecture exploration. Section 2.4 will discuss this in more detail.

Figure 2.4 shows the classical five-stage out-of-order pipeline of the datagraph simulator. It differs from a real out-of-order CPU in the following aspects:

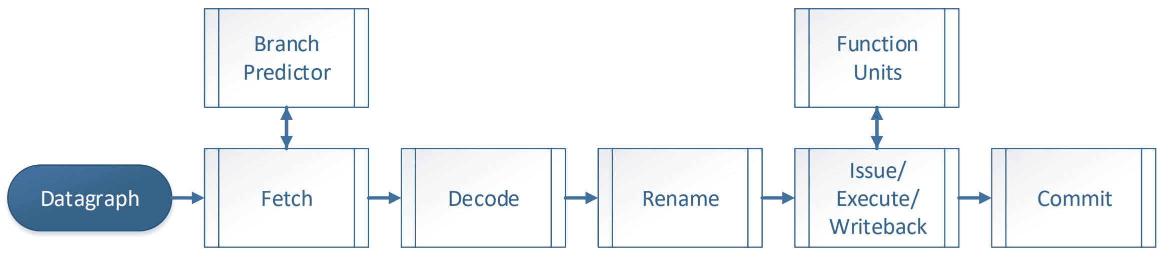


Figure 2.4: Five-stage pipeline of the datagraph simulator.

- LLVM-IR assumes a virtual machine with an infinite number of registers, and the datagraph simulator also makes this assumption. This means that the rename stage of the datagraph simulator will always succeed in finding an available physical register.

- Since the simulator executes a datagraph instead of a real code region, there is no instruction fetched from the memory and no instruction cache miss. This will cause some inaccurate results, but for a loop which can be fit completely in the instruction cache, the initial instruction cache miss may be negligible. Considering that an accelerator usually targets some frequently executed loops, we believe that the effect of not having an instruction cache can be negligible. Of course, if wanted, we can emulate with a pseudo instruction cache on static LLVM-IR index rather than the program counter. This can be integrated into the datagraph simulator for more accurate results.

- Another piece of information missing from the LLVM-IR datagraph is branch misprediction and squashed instruction. This is an issue for some programs that exhibit little branch predictability, and the cycles spending on squashing is not negligible. To improve the accuracy, a branch predictor is integrated into the simulator and will stall the fetch stage whenever a branch misprediction is detected.

## 2.4 Flexible Datagraph Transformation and Replay

To ensure rapid accelerator design exploration, it is crucial to provide flexible datagraph transformation and replay. The user should be able to define new instructions, transform the datagraph and extend the datagraph simulator without modifying existing ISA or hacking into a real compiler.

The datagraph, although constructed from an LLVM-IR trace, is designed to extendable and can contain any non-LLVM-IR nodes. New special nodes can be defined as long as they implement the interface of the datagraph node, which is effectively the same as introducing new instruction into the ISA.

The datagraph transformer is in charge of rewriting the datagraph to reflect any desired architecture features. For example, some computation subgraphs can be offloaded to an accelerator and replaced by a special node, as the case of CCA [CKP04]. A few other examples of how the datagraph can be transformed are given at the end of this section. To leverage the high-level compiler information, the datagraph transformer is implemented as an LLVM optimization pass and is seamlessly integrated with the LLVM framework. It is flexible to be inherited from, and the user can require any necessary LLVM analysis to assist the transformation.

The final step is to define the functionality of the new instructions, i.e., how should the datagraph simulator handle these new instructions. First, if wanted, the datagraph transformer can provide a context for each inserted accelerator-related instruction. This context contains information on how the instruction should be simulated, e.g., the latency, function units required, etc. The user should also provide an execution function to the datagraph simulator. During replay, when the simulator tries to execute an accelerator-related instruction, it will call the user-defined execution function with any provided context. The execution function is in charge of completing the functionality of the instruction, interacting with the simulator and reporting any statistics, e.g., register access, etc. Notice that the datagraph simulator makes no assumption on the context and the execution function, which further ensures flexibility. Also, the context/execution function extraction makes it possible to integrate multiple accelerators in a single system, and enables cross-accelerator exploration.

*Accelerator Examples.* Here are a few examples of how the datagraph can be transformed for some existing transparent accelerators. Notice that we will provide a detailed case study on CCA [CKP04] in section 3.2.

- SIMD (Loop Vectorization)

  To perform loop vectorization, the transformer will first analyze the register and memory dependence of loops, either with the `memdep` analysis pass provided by LLVM framework or using the dynamic trace. Loops with non-vectorizable memory dependence or inter-iteration register dependence will be ignored. For profitable loops, every $v$ iterations will be merged, where $v$ is the vectorization number. If there is discontinuous memory access, pack/unpack instructions will also be inserted.

- CCA (Configurable Compute Accelerator)

  CCA [CKP04] is a configurable matrix of function units integrated within the CPU, which can execute some computation instructions but no memory access. The datagraph transformation is straight-forward: detecting accelerable subgraphs and replacing them with special `cca` instructions, as discussed in section 3.2.

- BERET

  BERET [GFA11] differs from CCA in three ways. First, instead of a single CCA, there are multiple SEBs in the architecture, and they can be chained in the configuration. Second, the SEB can perform memory access. Third, if the speculation fails, the control must be transferred back to the CPU.

  To simulate BERET, the whole SEB configuration will be saved as the instruction context, and the execution function will perform memory access through the interface with the datagraph simulator. In case of missed speculation, the original subgraph will not be removed, and the datagraph simulator will restart from the original one.

- DySER

  DySER [GHN12] is another transparent CGRA accelerator for loops. Three key instructions are defined: `dyLoad` to set up the CGRA, `dySend` to send input data, and `dyRecv` to retrieve the results. Unlike BERET, the DySER does not access memory directly, which makes it simpler to implement the execution function. The main complexity comes from the complicated transformation to split the loop into sending and receiving phases. However, LLVM already provides dominance and post-dominance frontier analysis, which should simplify the implementation of datagraph transformation.

- Chainsaw

  Chainsaw [SKG16] maps a dependent chain into a temporal reused function unit and uses bypassing to internalize the communication cost. It also organizes hot regions into super-block to discover longer chains. To accurately simulate Chainsaw, the datagraph transformer will detect accelerable chains and construct the substitute instruction and context. The context should contain enough information to capture the inter-chain dependence.

- GPU-SIMT

  Generally, GPU-SIMT model is not considered as a transparent accelerator. It is possible to simulate GPU-SIMT with datagraph in our framework. The subgraphs of each loop iteration can be handled by one thread, and the execution function will simultaneously execute these subgraphs and insert necessary stalls for synchronization.

# CHAPTER 3

# Results and Analysis

## 3.1 Verification

To show that our framework works, there two levels of verification to do. The first one is functional verification. Taking the datagraph simulator as a black box reads and writes memory, we must ensure that it accesses the correct memory location and outputs correct results. The second one is performance verification. Compared with a baseline of a conventional cycle-level simulator, the datagraph simulator must deliver similar simulation results.

### 3.1.1 Functional Verification

We leverage MachSuite [RAS14] as the benchmark to perform functional verification. It provides different kernels with reference outputs. The output of the datagraph simulator is compared with the reference output to verify that its functional correctness. Experiments on all the 19 kernels of MachSuite proves that the datagraph simulator performs correct memory accesses.

### 3.1.2 Performance Verification

The O3 CPU of the gem5 simulator [BBB11] is used as the baseline for performance verification. Many metrics can be used for comparison, e.g., cache miss rate, branch misprediction rate, etc. However, for simplicity, here we will focus on three metrics: the number of memory read, the number of memory write, and the overall latency in simulated execution time. If wanted, other statistics can also be collected from the datagraph simulator and compared.

First, a micro-benchmark, Vertical [Now15], is tested. Each workload will test a specific characteristic of the simulator, e.g., memory latency, usage of function units, branch prediction, data parallelism, etc. This helps to quickly verify that whether a particular component of the datagraph simulator works appropriately or not. Table 3.1 lists a subset of the micro-benchmark with a simple description.

| Name | Description |
|---|---|
| CCa | Completely biased branch. |
| CCm | Heavily biased branches. |
| CS1 | Switch case statement of size 10 – different case each time. |
| CS3 | Switch case statement of size 10 – different case every third time. |
| DP1f | Simple data parallel loop – float arithmetic. |
| DPcvt | Simple data parallel loop – simple data parallel loop float/double conversion. |
| ED1 | Integer execution – length 1 dependency chain per iteration. |
| EF | Floating-point execution – 8 independent instructions per iteration. |
| EI | Integer execution – 8 independent computations per iteration . |
| EM1 | Integer execution – length 1 dependency chain each loop (with multiplies). |
| EM5 | Integer execution – length 5 dependency chain each loop (with multiplies). |
| MD | Cache resident linked list traversal. |
| MI | 8 streams of independent memory access, all cache resident. |
| MM | Non-cache resident linked-list traversal. |
| MM_st | Non-cache resident linked-list traversal (with stores). |
| STc | Repeatedly store in consecutive access - l1 cache. |
| STL2 | Repeatedly store, l2 cache resident. |
| STL2b | Repeatedly store, l2 resident (occasional stores). |

Table 3.1: Vertical micro-benchmark.

Table 3.2 and Figure 3.1 shows the results of verification on Vertical benchmark. The datagraph simulator can get average of 75.18% on simulated execution time. Notice that for the EM5 benchmark, a bug in gem5 causes the multiplication always dependent on the previous multiplication and thus not parallelizable. However, the datagraph simulator will be able to detect the parallelism between multiplication if there is no data dependence and produces a much smaller simulated latency.

| | Write | Read | **Execution Time** | MicroOp |
|---|---|---|---|---|
| CCa | 100.00% | 100.00% | **66.46%** | 37.50% |
| CCm | 100.00% | 100.00% | **66.67%** | 37.51% |
| CS1 | 100.00% | 100.00% | **66.93%** | 32.47% |
| CS3 | 100.00% | 100.00% | **58.89%** | 32.94% |
| DP1f | 95.87% | 96.96% | **64.55%** | 71.43% |
| DPcvt | 100.00% | 100.00% | **64.67%** | 64.28% |
| ED1 | 100.00% | 100.00% | **66.67%** | 10.53% |
| EF | 100.00% | 100.00% | **79.66%** | 73.33% |
| EI | 100.00% | 100.00% | **65.20%** | 61.11% |
| EM1 | 100.00% | 100.00% | **96.18%** | 40.00% |
| EM5 | 100.00% | 100.00% | **41.67%** | 36.36% |
| MD | 100.00% | 95.12% | **76.94%** | 99.94% |
| MI | 100.00% | 99.76% | **90.09%** | 98.44% |
| MM | 100.00% | 99.77% | **77.89%** | 99.94% |
| MM_st | 100.00% | 99.95% | **82.72%** | 85.67% |
| STL2 | 96.80% | 100.00% | **95.55%** | 68.75% |
| STL2b | 100.00% | 97.31% | **97.08%** | 74.89% |
| STc | 100.00% | 100.00% | **95.43%** | 59.97% |
| Average | 99.59% | 99.38% | **75.18%** | 60.28% |

Table 3.2: Verification of Vertical micro-benchmark.

For this micro-benchmark, since the workload is very simple, there are generally few register spills and fills, as we can see from the highly accurate results on the number of memory reads and writes. The primary source of error comes from the difference between the number of LLVM-IR instructions in the datagraph and the number of micro-operations

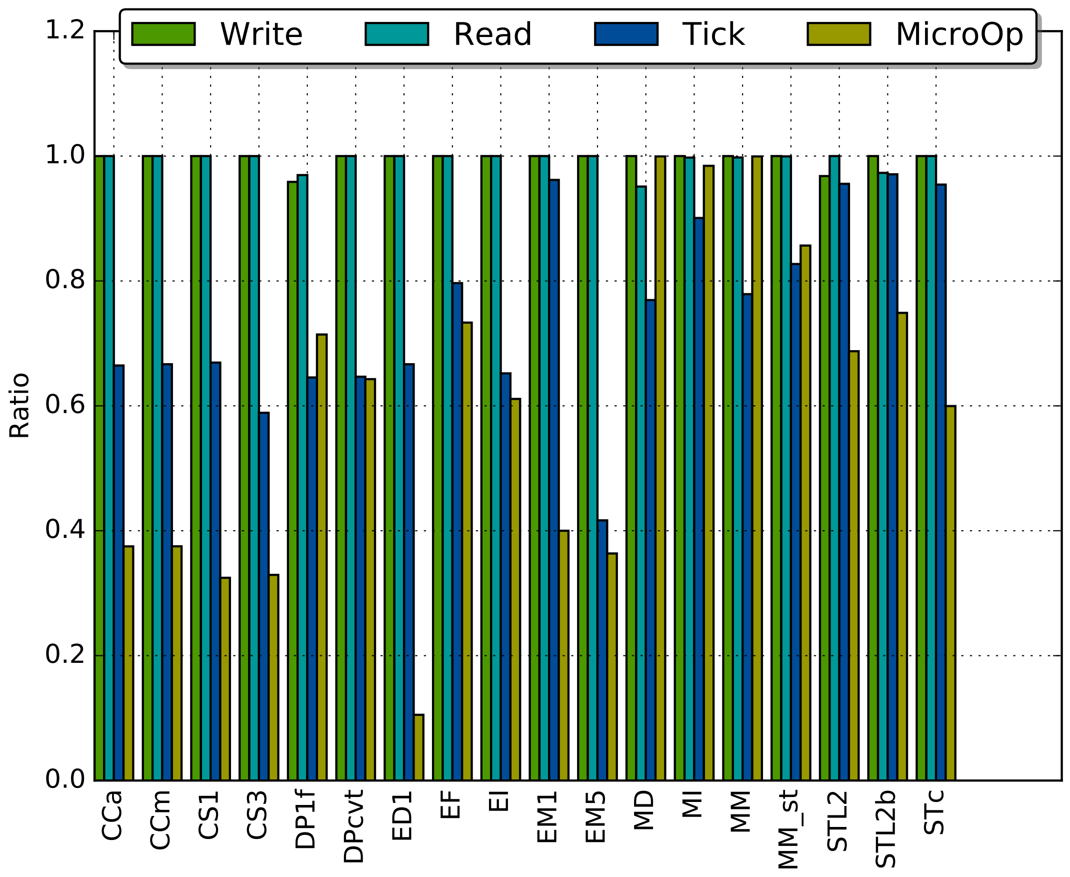


Figure 3.1: Verification of Vertical micro-benchmark.

simulated in gem5. After optimization, an LLVM-IR instruction will be compiled to some ISA instructions in code generation phase. In the decode stage of a CPU, an ISA instruction is further decomposed into one or more micro-operations. For example, in the x86 architecture, if one of the operands is an immediate number, gem5 will decode the `add` instruction into two micro-operations: one to load the immediate number into a register, and the other to add two registers together. This generally means that the number of micro-operations is higher than the number of LLVM-IR instructions, and results in less simulated execution time. Table 3.2 shows that on average the number of LLVM-IR instructions is only 60.28% of the number of micro-operations.

To mitigate this effect, one possible solution is to transform the datagraph to be more consistent with the actual micro-operations executed in the CPU. For example, for each immediate number operand of an `add` node, we can insert a "fake" node into the datagraph and make the `add` dependent on this new node. Another more straightforward way is to adjust the latency of some LLVM-IR instructions based on how they are going to be decoded. Also, we can change some parameters of the datagraph simulator, e.g., issue width, etc. to reflect the resources occupied by additional micro-operations. This can be explored in future works.

Table 3.3 and Figure 3.2 shows the results of verification on MachSuite benchmark. The datagraph simulator can get average of 53.07% on simulated execution time. The accuracy drops due to the following reasons:

- As the program gets more complicated, there are more optimization opportunities in code generate phase, which increases the gap between the number of LLVM-IR instructions and micro-operations. Table 3.3 shows that on average the number of LLVM-IR instructions is only 55.79% of the number of micro-operations, which partially explains why the simulated execution time drops to 53.07%.
- The number of memory reads and writes reported by the datagraph simulator is only one-third of those from the baseline. The extra memory access comes from register spills and fills. LLVM assumes a virtual machine with an infinite number of registers. However, the number of registers in the ISA is finite, and the compiler will insert some code to spill registers to stack or load them back. Although some of the memory access may hit the L1 cache, it still consumes the limited capacity of the cache and may affect other memory access.

Possible solutions to reduce the gap between the number of LLVM-IR instructions and micro-operations has been discussed above. For effect from register spills, the datagraph simulator can monitor the number of alive values, and if it exceeds the number of ISA registers, either a simple register allocator can be used to spill some values to the stack, or some other penalties can be introduced to achieve similar effects. However, notice that register spills is not a characteristic of the program but the architecture, and for some register-rich accelerators it may be negligible.

| | Write | Read | **Execution Time** | MicroOp |
|---|---|---|---|---|
| BFS-QUEUE | 1.88% | 22.57% | **47.15%** | 51.99% |
| FFT-STRIDE | 52.77% | 39.24% | **59.90%** | 53.47% |
| FFT-TRANSPOSE | 24.71% | 11.98% | **16.91%** | 13.48% |
| GEMM-BLOCKED | 76.03% | 43.24% | **67.82%** | 68.96% |
| GEMM-NCUBED | 21.60% | 47.07% | **57.75%** | 84.05% |
| KMP | 100.00% | 20.07% | **48.43%** | 48.19% |
| MD-GRID | 3.07% | 15.65% | **41.00%** | 39.46% |
| MD-KNN | 2.40% | 25.04% | **50.91%** | 65.98% |
| NW | 23.10% | 30.31% | **69.45%** | 68.12% |
| SPMV-CRS | 3.00% | 33.02% | **57.04%** | 56.69% |
| STENCIL-2D | 35.52% | 44.52% | **36.24%** | 35.59% |
| STENCIL-3D | 44.35% | 48.60% | **86.53%** | 86.45% |
| VITERBI | 11.82% | 52.90% | **50.84%** | 52.77% |
| Average | 30.79% | 33.40% | **53.07%** | 55.79% |

Table 3.3: Verification of MachSuite benchmark.

## 3.2 Case Study: CCA

To demonstrate the flexibility how the datagraph can be used to explore accelerator design, a case study on configurable compute accelerator (CCA) [CKP04] is conducted. A CCA contains an array of function units and can be configured to execute some subgraphs efficiently. Figure 3.3 shows the block diagram of a depth 7 CCA. The idea is to explore how integrating a CCA with a general processor architecture improves the performance.

The LLVM-IR datagraph is transformed to utilize the new CCA. An algorithm similar to the original one proposed in [CKP04] is implemented to detect subgraphs that can be offloaded to the CCA. A subgraph is accelerable if it satisfies the following constraints:

- It cannot span multiple basic blocks, i.e., there is not control dependence within the

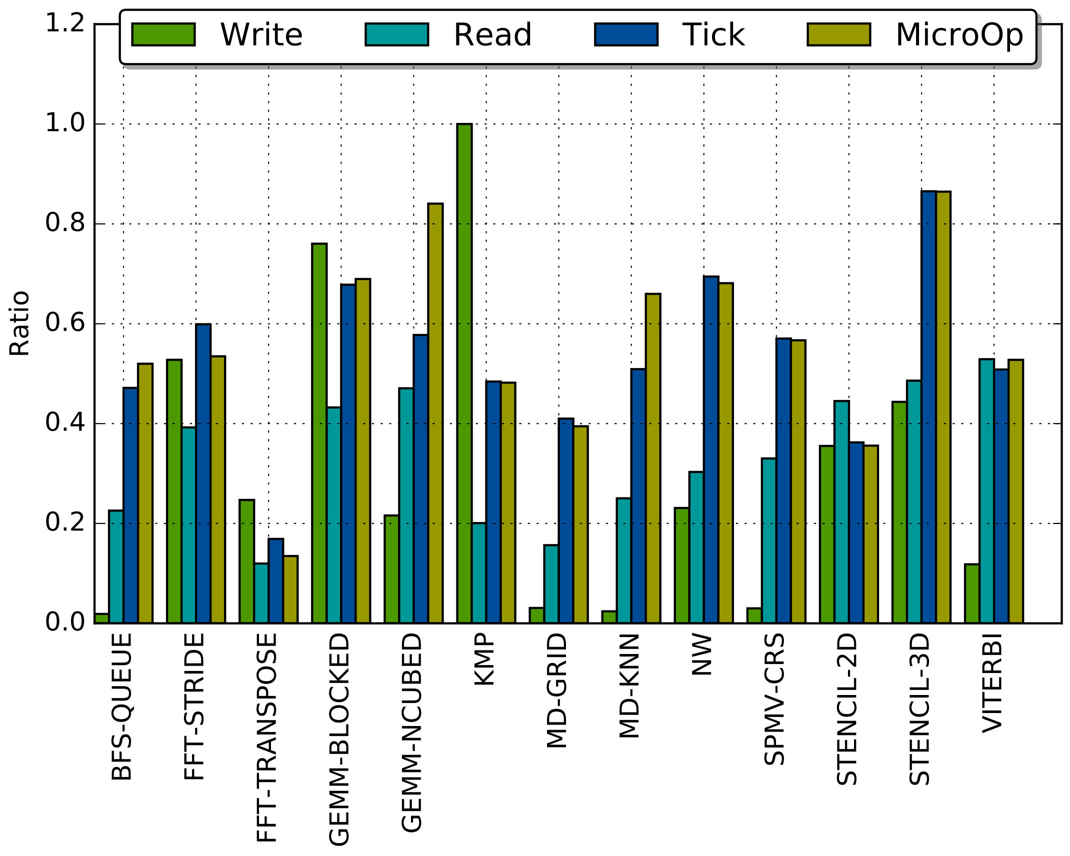


Figure 3.2: Verification of MachSuite benchmark.

subgraph.

- It is not beyond the capability of the CCA. First, it can not contain any operation which can not be handled by the CCA, e.g., memory access, etc. Also, it should be able to be handled by the limited hardware resources of CCA, e.g., number of input and output ports, number of functional units.

Figure 3.4 shows an example of how the datagraph is transformed. This comes from a basic block in NW workload of MachSuite benchmark [RAS14]. The red nodes form a subgraph that is detected and can be offloaded to CCA. After transformed, each subgraph is replaced by a special node `cca`. Each `cca` node is associated with a context, which contains information to help the datagraph simulator simulate it, e.g., the offloaded instructions, the

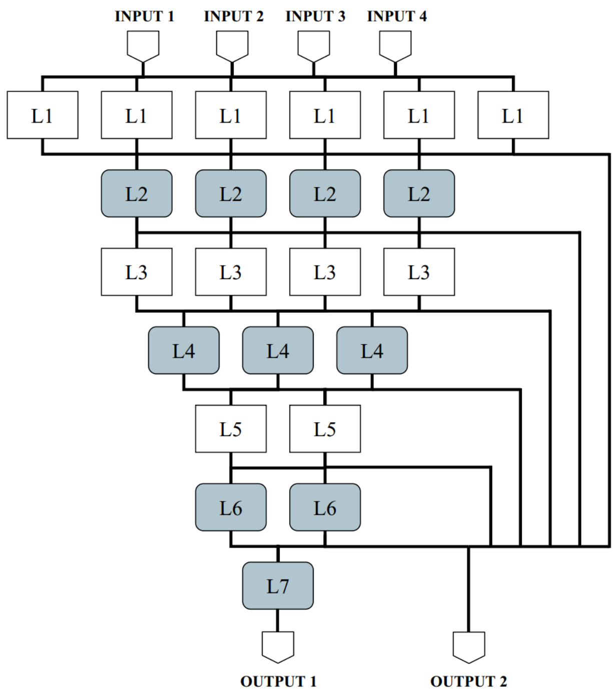
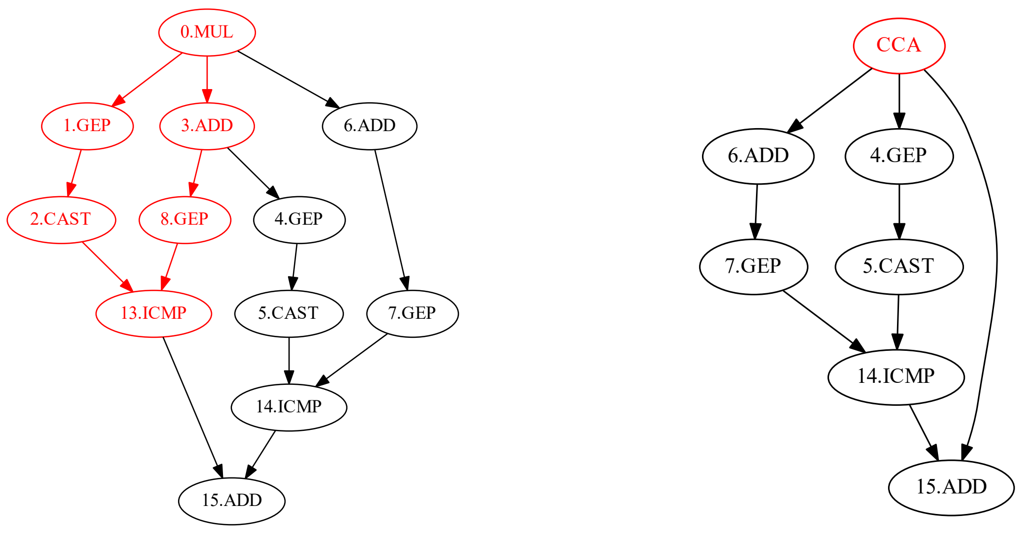


Figure 3.3: Block diagram of the depth 7 CCA. [CKP04]

computation latency, etc. The transformed datagraph is then sent into the datagraph simulator, and the simulation results can be compared with those of untransformed datagraph to determine the speedup.

Table 3.4 shows the cumulative percentage of dynamic subgraphs in MachSuite benchmark with varying depth. Here the depth of a subgraph is defined as the length of the longest dependency chain. With depth increasing from 2 to 4, the cumulative percentage increases from 78.05% to 93.57%. This suggests that a CCA with four layers of function units may be a good choice to minimize the area overhead while covering the majority of subgraphs.

Figure 3.5 shows the performance of CCAs with different maximum depth supportable. The subgraphs are identified using the dynamic discovery algorithm. The speedups are computed as the ratio between the simulated execution time of the original datagraph and the transformed one. For many workloads, increasing the depth of the CCA doesn't improve the performance, as most of the accelerable subgraphs have a depth less than 5, e.g., BFS-QUEUE and KMP. Sometimes, larger CCA may even hurt the performance, as for MD-KNN, etc. This is because the heuristic used by the dynamic subgraph discovery algorithm will



(a) A datagraph before transformed. (b) A datagraph after transformed.

Figure 3.4: An example of CCA transformed.

aggressively offload larger subgraphs to the CCA, which may reduce the size of remaining subgraphs, making them unprofitable to be offloaded and resulting in less coverage.

The case study on CCA shows that our framework can efficiently explore accelerator design space with reduced complexity.

## 3.3 Comparison with Other Work

There are some other datagraph-based simulators, and in this section, we discuss the similarity and difference between our framework and others.

Aladdin [SRW14a] and gem5-Aladdin [SRW14b] are both simulators based on LLVM-IR trace. They both construct the LLVM-IR datagraph and use it to simulate the performance and energy. The first one focuses on the performance estimation, while the second one is integrated into gem5 and takes memory bandwidth and latency into consideration. Their features can be summarized as the following.

| Depth | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| BFS-QUEUE | 97.61% | 100.00% | 100.00% | 100.00% | 100.00% | 100.00% |
| FFT-STRIDE | 92.54% | 92.54% | 92.54% | 100.00% | 100.00% | 100.00% |
| FFT-TRANSPOSE | 64.39% | 91.30% | 95.52% | 95.52% | 95.52% | 100.00% |
| GEMM-BLOCKED | 94.12% | 94.77% | 100.00% | 100.00% | 100.00% | 100.00% |
| GEMM-NCUBED | 15.24% | 97.28% | 97.28% | 97.28% | 97.28% | 100.00% |
| KMP | 80.23% | 80.23% | 100.00% | 100.00% | 100.00% | 100.00% |
| MD-GRID | 78.48% | 78.81% | 89.46% | 89.52% | 89.52% | 100.00% |
| MD-KNN | 82.42% | 82.42% | 82.42% | 82.42% | 82.42% | 100.00% |
| NW | 69.43% | 84.80% | 84.94% | 99.88% | 100.00% | 100.00% |
| SPMV-CRS | 93.54% | 93.54% | 98.22% | 100.00% | 100.00% | 100.00% |
| STENCIL-2D | 68.42% | 77.38% | 91.52% | 91.52% | 91.99% | 100.00% |
| STENCIL-3D | 87.95% | 88.72% | 89.13% | 89.50% | 89.50% | 100.00% |
| VITERBI | 90.31% | 95.30% | 95.33% | 95.33% | 95.33% | 100.00% |
| Average | 78.05% | 89.01% | 93.57% | 95.46% | 95.50% | 100.00% |

Table 3.4: Cummulative percentage of dynamic subgraph with varying depth.

- Both Aladdin and gem5-Aladdin are targeting for a fixed-function accelerator, i.e., the whole datagraph will be executed on a specialized accelerator.
- For gem5-Aladdin, the CPU communicates with the fixed-function accelerator via `ioctl` system call and DMA. It cannot model a core-integrated accelerator and capture the tight interaction between them.
- The datagraph transformation is done outside the LLVM framework, which means it can only leverage the limited compiler information from the trace.

To summarize, Aladdin and gem5-Aladdin enable design space exploration for exposed fixed function accelerators with limited compiler support.

Transformable dependence graph (TDG) [NGS15] is also a datagraph-based simulator.

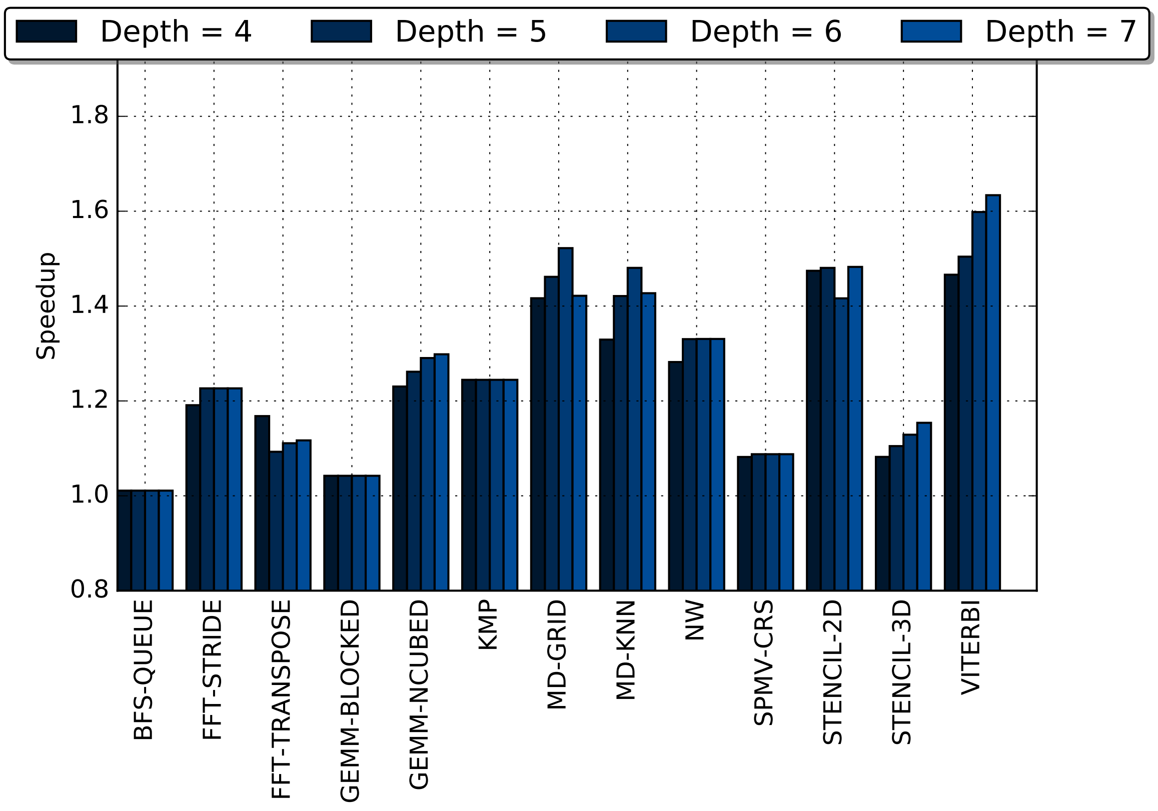


Figure 3.5: Varying the CCA configuration.

Similar to our framework, it can also transform the datagraph to reflect characteristics of different architecture and enable cross-domain comparison and design-space exploration. TDG's datagraph is composed of micro-operation and low-level hardware events, and it can achieve very accurate simulation results ($< 4\%$ avg. error) [NGS15]. However, working on micro-operation level loses some high-level compiler information, and makes it more difficult to transform the datagraph.

Chainsaw [SKG16] also comes with a simulator specialized for its architecture. It builds chains from the instruction trace and uses gem5's ruby system to simulate for memory latency.

Needle [KSS17] is another LLVM-based framework focused on the frontend. A similar LLVM-IR trace will be used to detect frequent basic blocks, which will be outlined into a separate function with software speculation. The outlined function enables optimization opportunities on super-blocks for the backend. However, it does not provide a simulation backend.

### 3.3.1 Strength

Table 3.5 lists the comparison between the existing frameworks and ours on four dimensions, full compiler support, simulation backend, cross-accelerator exploration and target accelerator paradigm. Aladdin and gem5-Aladdin are targeting for exposed accelerators, so the cross-accelerator dimension is not applicable for them. As for Needle, since it does not come with a simulation backend, this dimension is also marked as not applicable.

Compared with existing frameworks, our datagraph simulator is the first one to provide end-to-end cross-accelerator exploration with high-level compiler information. Among the previous works, the most similar one is TDG, which only lacks the compiler information. However, as discussed in section 2.4, some existing compiler analysis passes, e.g., dominance and post-dominance frontier analysis, should be useful for transforming the datagraph. More importantly, combining the static compiler analysis with dynamic information from the datagraph may provide some insights for new accelerator designs.

| Framework | Compiler Support | Sim-Backend | Cross-Accel | Target |
|---|---|---|---|---|
| Aladdin [SRW14a] | ✓ | ✓ | N/A | Exposed |
| gem5-Aladdin [SRW14b] | ✓ | ✓ | N/A | Exposed |
| TDG [NGS15] |  | ✓ | ✓ | Transparent |
| Chainsaw [SKG16] |  | ✓ |  | Transparent |
| Needle [KSS17] | ✓ |  | N/A | Transparent |
| This thesis | ✓ | ✓ | ✓ | Transparent |

Table 3.5: Comparison with existing frameworks.

### 3.3.2 Limitations

One significant limitation of our framework is that compared with ISA, LLVM-IR is still a high-level abstraction, and the verification results suffer from this gap. It requires some efforts to mitigate this effect and bring the simulation results more closely to the baseline out-of-order processor. However, in the sense of estimating accelerators' speedup, this effect may not be an obstacle, because accelerators tend to eliminate processor bottlenecks like register pressure anyway. As evidence of this, the estimated speedup of CCA is reasonable compared with the original paper. More accelerator architectures can be examined for further validation.

There are also some accelerator architectures that cannot be modeled by the existing implementation. Like any other trace-based simulators, our framework does not support accelerators for multi-thread workloads. This is because the interaction between threads is usually indeterminate, and adding accelerators to the system will change the instruction stream of some threads and invalidate the collected trace.

For some architectures, the current framework is not flexible enough. For example, the decoupled execution function can only interact with memory system through CPU's cache hierarchy. For accelerators which are more independent from the CPU, or even has its own memory system, it requires some extension to the datagraph simulation. An example is GPU-SIMT model, which requires some complicate datagraph transformation and replay control.

# CHAPTER 4

# Conclusion and Future Works

## 4.1 Conclusion

In this thesis, we present a flexible LLVM-IR datagraph-based simulator for transparent accelerators. On the Vertical micro-benchmark, the new framework achieved average 75.18% performance of the baseline out-of-order processor simulator. On the Machsuite benchmark, the performance accuracy drops to 53.07% due to the effect of register spills. We also demonstrate the flexibility and modeling capacity with a case study which models the CCA micro-architecture. With the high-level compiler information from the LLVM framework, the datagraph is transformed to utilize CCA without extending the ISA or modifying the compiler.

## 4.2 Future Works

One of the future works involves validation on more complicated existing accelerator micro-architecture, e.g., Dyser [GHN12], BERET [GFA11], etc. This will also lay the groundwork for all the research questions of multiple accelerators.

The replayable fraction of a workload is limited since the replay is performed at the function level. However, this is relatively conservative as a function may call a library function but also contains a fully traced loop which is of interest. A more fine-grained replay scheme is to automatically detect some fully traced regions of interest and outline them into separate functions.

It is also crucial to bring power estimation into the datagraph simulator to make it practical to use. McPAT [LAS09], a state-of-the-art general-purpose processor power model, is a good candidate.

Another natural extension would be support systems with multiple processors. Multiple datagraphs can be simulated on separate processors, and a particular interface can be added to allow temporal multiplexing accelerators among processors.

Finally, a long-term goal is to extend this framework for other accelerator paradigms. For example, a particular interface to bypass the cache hierarchy and communicate directly with other subsystems would enable the modeling of standalone accelerators.

## REFERENCES

[BBB11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. "The Gem5 Simulator." *SIGARCH Comput. Archit. News*, **39**(2):1–7, August 2011.

[CKP04] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization." In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pp. 30–40, Washington, DC, USA, 2004. IEEE Computer Society.

[FBH03] Brian A. Fields, Rastislav Bodík, Mark D. Hill, and Chris J. Newburn. "Using Interaction Costs for Microarchitectural Bottleneck Analysis." In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pp. 228–, Washington, DC, USA, 2003. IEEE Computer Society.

[GFA11] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August. "Bundled execution of recurring traces for energy-efficient general purpose processing." In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 12–23, Dec 2011.

[GHN12] V. Govindaraju, C. H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. "DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing." *IEEE Micro*, **32**(5):38–51, Sept 2012.

[KSS17] Snehasish Kumar, William Sumner, Vijayalakshmi Srinivasan, Steve Margerm, and Arrvindh Shriraman. "NEEDLE: Leveraging Program Analysis to extract Accelerators from Whole Programs." In *High Performance Computer Architecture (HPCA2017), 2017 IEEE 23rd International Symposium on*. IEEE, 2017.

[LAS09] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures." In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 469–480, Dec 2009.

[NGS15] T. Nowatzki, V. Govindaraju, and K. Sankaralingam. "A Graph-Based Program Representation for Analyzing Hardware Specialization Approaches." *IEEE Computer Architecture Letters*, **14**(2):94–98, July 2015.

[Now15] T. Nowatzki. "Vertical Micro-Benchmark.", 2015.

[RAS14] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. "MachSuite: Benchmarks for Accelerator Design and Customized Architectures." In *Proceedings of the IEEE International Symposium on Workload Characterization*, Raleigh, North Carolina, October 2014.

[SKG16] Amirali Sharifian, Snehasish Kumar, Apala Guha, and Arrvindh Shriraman. "Chainsaw: Von-neumann accelerators to leverage fused instruction chains." In *In Proceedings of the International Symposium on Microarchitecture*, 2016.

[SRW14a] Y. S. Shao, B. Reagen, G. Y. Wei, and D. Brooks. "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures." In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 97–108, June 2014.

[SRW14b] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. "Aladdin: A Pre-RTL, Power-performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures." *SIGARCH Comput. Archit. News*, **42**(3):97–108, June 2014.