RUNTIME AND INSTALL-TIME BINARY TRANSLATION FOR RECONFIGURABLE ACCELERATORS

Toan Xuan Mai

Computer Engineering Program
Graduate School of UNIST

Runtime and Install-time Binary Translation for Reconfigurable Accelerators

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Toan Xuan Mai

01.18.2013
Approved by

Major Advisor
Jongeun Lee

Runtime and Install-time Binary Translation for Reconfigurable Accelerators

Toan Xuan Mai

This certifies that the thesis of Toan Xuan Mai is approved.

01.18.2013

Thesis Supervisor: Jongeun Lee

Beomseok Nam: Thesis Committee Member #1

Won-ki Jeong: Thesis Committee Member #2

Abstract

Nowadays, softwares are often distributed in form of some machine-independent intermediate representation (IR), because compared to machine-dependent native binary, the IR is more portable across a wide range of architectures, has better security, and contains richer semantic information. However, the problem of making use of the accelerator in a target machine to speedup the execution of the IR on top of a just-in-time compiler (JIT) is challenging, mainly because the discovery of compute-intensive kernels and the partitioning of the application to the kernel and sequential parts must be done based on the IR alone, without the access to the program source code as well as the kernel information in the IR.

In this work, we propose a Runtime Binary Translation (RBT) technique that can dynamically identify and translate kernels IR to Coarse-Grained Reconfigurable Array (CGRA) accelerator configuration, and offload the execution of the kernels onto the accelerator. Also, we simplify the RBT approach to make the Install-time Binary Translation (IBT) approach, which does the partitioning and the translation right at the install-time instead of at the runtime. Experimental results show that our RBT and IBT techniques can improve the runtime of the application IR by 1.44 times and 1.61 times, respectively, compare to the runtime on the JIT that does not support making use of the accelerator.

Contents

List of Figures

CHAPTER I

Introduction

Nowadays, software programs can be distributed in from of some machine-independent *Intermediate Representation (IR)*, such as Java bytecode, Common Intermediate Language (CIL) or LLVM bitcode, etc., instead of in machine-specific native binary code. Distributing softwares in the IR format provides several advantages [1]. The softwares are more *portable* across different hardware architectures because the IR is machine-independent. In addition, the system running the IR will have *better security* because it can examine the IR for attempts to bypass security before executing it. Another advantage is that a program stored in the IR format generally has *smaller size*, yet contains *richer information* (e.g., type information, etc.) than the same program stored in native binary format does.

On a specific target system, an application IR will be run on top of an *execution engine* acting as a software layer between the IR and the operating system. The execution engine can be a Just-It-Time compiler (JIT), an interpreter, or some hybrid combination of the two. The JIT compiles the application IR into native binary code right before executing it, while the interpreter just reads the IR statement by statement and performs a sequence of native instructions for each IR statement. Therefore, once the JIT finishes compiling the IR, it gives better execution speed than the interpreter does. Having that said, the execution on the JIT can still be slow because of some compute-intensive loops, which we call *kernels*, in the application.

To speedup the execution of the kernels in an application, nowadays, people often make use of some accelerator (e.g., hardware accelerators, reconfigurable array, GPGPU, etc.), if any, in the target machine. Traditionally, they have the source code of the application, the information about the kernels in the applications, as well as the knowledge about the accelerator in the target machine. Then, they have to manually partition and rewrite the source code of the program into kernel code and sequential code before statically compiling those to the executable binary. However, when it comes to making use of accelerator to speedup the execution of application IR, the process is not so straightforward.

1.1 Thesis Statement

When application softwares are distributed in the IR format which is intended to be machine-independent, making use of the accelerator in the target machine is a challenging problem where the discovery of kernels and the partitioning of the application to the kernel and sequential parts must be done based on the IR alone, without the program source code as well as the kernel information in the applications.

In this thesis, we address this problem particularly for the target machines which are equipped with *Coarse-Grained Reconfigurable Array (CGRA)* accelerators (e.g., ADRES [11], FloRA [10], and SPIRA [9]), by proposing an approach called *Runtime Binary Translation (RBT)*. In the RBT approach, at the *runtime*, the *RBT Virtual Machine (RBTVM)* identifies kernels that are suitable for running on the accelerator, translates those kernels to the binary format of the accelerator (i.e., *accelerator configuration*), and offloads the execution of the kernels onto the accelerator. The RBTVM is basically a JIT compiler, but is different from the conventional JIT in that it supports more runtime features as described. Beside RBT which is the primary approach, we also come up with a simpler approach called *Install-time Binary Translation (IBT)*. In the IBT approach, all the loops that are suitable for being executed on the accelerator will be translated at the *install-time* instead of at the runtime. The accelerator configuration produced at the install-time will be coupled with the original IR, and will be used at the runtime.

We design and implement the RBT and IBT approaches based on the LLVM compilation framework [2], targeting the SPIRA architecture [9]. With the main objective of maximizing the runtime improvement, most of the challenging issues we have encountered so far are from the design and implementation of the RBT approach. The reason is because in this approach, there are lots of runtime features supported, which add significant overhead to the runtime performance.

1.2 Challenging Issues

There are two main issues in the design and implementation of the RBT approach. The first issue is that system must be able to recognize the suitable kernels that will be run on the accelerator. This is challenging because not all loops are kernels, so the performance gain of offloading the execution of those loops onto the accelerator may not be worth the cost of translation. Moreover, not all kernels are *well-suited* for running on the accelerator. Particularly, when the accelerator used is a Coarse-Grained Reconfigurable Array (CGRA), a kernel may not be well-suited if the number of iterations of it is not loop-invariant, or if the kernel body contains some function calls. To address the first issue, our RBTVM instruments and monitors the loops to detect kernels, and then examines the kernels' bodies to identify the well-suited ones.

The second issue is the high runtime overhead incurred by the RBTVM, due to a lot of runtime features supported such as loop instrumentation, loop profiling, function recompilation, etc. We tackle this issue by introducing some optimizations for the RBTVM. One example is the optimization for reducing the recompilation overhead. In the initial design of the RBTVM that we come up with, after identifying a kernel, translating it into the accelerator configuration, and inserting necessary instructions for the accelerator control in the IR of the function containing the kernel, the system has to *recompile* that function IR and update the function binary code address so that the in the subsequent calls to the function, the kernel will be run on the accelerator.

The problem with this design is that the recompilation for a specific function IR may have to be done several times if it contains multiple kernels detected within the same invocation of the function, which may lead to redundant recompilation overhead. We avoid the redundant recompilations by introducing a technique called *Lazy Recompilation.* More specifically, the recompilation is delayed until the end of the function when all the possible kernels within the invocation has already been detected. Hence, within the same invocation, the recompilation for a function will happen just once at most.

1.3 Contributions of this Thesis

The main contributions of this thesis are three-fold. The first main contribution is that we propose the design of the *RBT* approach, which enables making use of the accelerator to speed up the execution of kernels in applications stored in the *IR* format, on a machine equipped with a CGRA accelerator. Also, we simplify the design of RBT to come up with the design for IBT, which has similar features but incurs less runtime overhead.

The second main contribution of this thesis includes the optimizations for the RBTVM that help reduce the runtime overhead, and the implementation of both RBT and IBT. The third main contribution is an in-depth analysis, discussion, and comparison with quantitative experimental results for both approaches.

1.4 Document Organization

The rest of this thesis is organized as follows. We explain some background and related work in Chapter II. The design, implementation, and optimizations of the RBT approach will be explained in Chapter III. In Chapter IV, we discuss the design and implementation of the IBT approach. After that, we discuss about the evaluation of both RBT and IBT approaches in Chapter V. Finally, we conclude and expose the future work in Chapter VI.

CHAPTER II

Background and Related Work

2.1 LLVM JIT Compiler

LLVM compilation framework [2] provides an *execution engine* that is used to run the LLVM bitcode, which contains the Intermediate Representation (IR) of an application. The execution engine can operate in either interpreter mode or Just-In-Time compiler (JIT) mode. We are particularly interested in the JIT mode. We refer to the original LLVM execution engine running in the JIT mode as the Base JIT. Fig. 2.1 shows the high-level design of the Base JIT. We can see that the context can be switched between two contexts when running an IR on top of the Base JIT: the Base JIT context and the *Host Execution* context (i.e., the main processor actually executes the binary of the application). The events where the context switching happens are denoted (1) and (2) in Fig. 2.1.

Initially, after loading the LLVM bitcode into the memory, the Base JIT starts in the JIT context. The JIT compiles only the IR of the *main function* to *native binary* code. Since the main function IR body may contain the function calls to other functions, and these functions have not yet been compiled, in the native code of the main function, the function calls to these functions will be replaced by *stubs*. The stubs are sort of the place-holders for the actual function calls to the not-yet-compiled functions. It is important to note that the Base JIT that we are talking about here works in a *lazy* fashion. That means, the JIT will delay the compilation

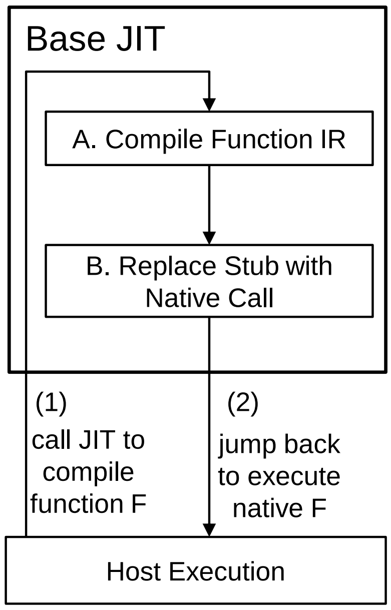

Base JIT
A. Compile Function IR
B. Replace Stub with Native Call
(1)
call JIT to compile function F
(2)
jump back to execute native F
Host Execution

Figure 2.1: Base Just-In-Time compiler design in the LLVM framework.

of a function as late as possible until right before the first execution of the function. It also means that the functions that are not executed are never compiled. After the compilation of the main function is done, the context is switched to the Host Execution context to execute the main function.

During the host execution of the main function, if a stub of a function F is reached, that means the function F definitely needs to be executed. The stub calls the JIT to compile the function F (i.e., event (1)). The context is switched from Host Execution to Base JIT, and the subcomponent *BaseJIT.A* compiles the function F. Again, because the body of the function F may contain function calls to the other functions F1, F2, etc., during the compilation of the function F, those function calls will be replaced by the corresponding stubs.

After the compilation of the function F, the subcomponent *BaseJIT.B* replaces the stub of F in the *main function's binary* with the call to the native function F which has just been produced. Then, the system will switch back to the Host Execution context (event (2) in the figure) and jump to *re-execute* the call to the native function F. The execution engine continues the execution that way until it finishes executing the application. Gradually, more and more functions will be compiled to the native binary and the context switching will occur less often.

2.2 Related Work

There have been works addressing the problems related to making use of accelerator to speedup the execution of application IR, such as [6], [7], and [8]. In [6], the IR used is Java bytecode and the target accelerator is an FPGA. At the runtime, the JVM employs a *merit-based* reconfiguration policy to decide which FPGA reconfigurations are most profitable at any given time. The translation from Java class to the FPGA configuration is actually done statically before JVM startup. On the other hand, the authors of [7] present a dynamic binary translation technique that also starts from Java bytecode, but targets a CGRA accelerator for fast reconfiguration and less control overhead. However, an additional hardware unit is required for detecting if a block of instructions is worth to be executed on the accelerator.

The authors of [8] propose a technique that can offload the execution of both dataflow and control-flow oriented code on to a CGRA accelerator. However, similar to [7], this technique requires an additional hardware that implements the binary translation algorithm and works in parallel to a MIPS processor. Our work is different from these previous work in that we provide a pure software solution, and we target the systems equipped with CGRA accelerators only. That means, our solution requires no additional hardware for the analysis and translation of the kernels to the CGRA accelerator configuration. Moreover, we propose two approaches for different time of the binary translation. The RBT does the binary translation dynamically, while IBT, which is a simpler version of RBT, does the binary translation statically at the install-time.

Coarse-Grained Reconfigurable Arrays (CGRAs) has been an active research area, and there have been lots of CGRA architectures proposed, such as ADRES [11], FloRA [10], and SPIRA [9]. For our work, we choose SPIRA as the target hardware. In the SPIRA architecture, a dedicated Sequential Processor (SP) is integrated with the Reconfigurable Array (RA) accelerator. Also, the SP and the RA share a common scratchpad data memory. These design features allow a close collaboration between the SP and the RA without excessive RA control overhead or data transfer overhead, thus expanding the range of loops that can be run on the RA.

In this work, in order to translate the IR of compute-intensive kernels to the CGRA accelerator configuration, we need to use a mapping algorithm. There have been studies on the mapping of software kernels to CGRA accelerator configuration such as [3] and [4]. These approaches are basically based on the iterative modulo scheduling algorithm [5] used for VLIW. We choose to use the *Edge-centric Modulo Scheduling* algorithm proposed in [4], because it seems to have the best mapping quality for our target RA accelerator in SPIRA at the moment.

CHAPTER III

Runtime Binary Translation

In the *Runtime Binary Translation (RBT)* approach, at the runtime, the *RBT Virtual Machine (RBTVM)* identifies kernels that are suitable for running on the accelerator, translates those kernels to accelerator configuration, and recompiles and updates the binary code of the function containing those kernels. In this chapter, we describe the design and implementation of the RBTVM, which is essentially a JIT, but with some major modifications to enable the runtime features mentioned above.

3.1 High-Level Design of RBTVM

We extend the base JIT (see Section 2.1) to make the RBTVM, which now comes with three main components: two levels of JIT and one Monitor. Fig. 3.1 shows the high-level design of the RBTVM where the new subcomponents that we added are in gray boxes. We can also see that there are three contexts when running the RBTVM: the RBTVM context, the Host Execution context (the execution on the SP), and the Accelerator Execution context. The switching between these contexts depends on the events represented by the arrows crossing the contexts in Fig. 3.1.

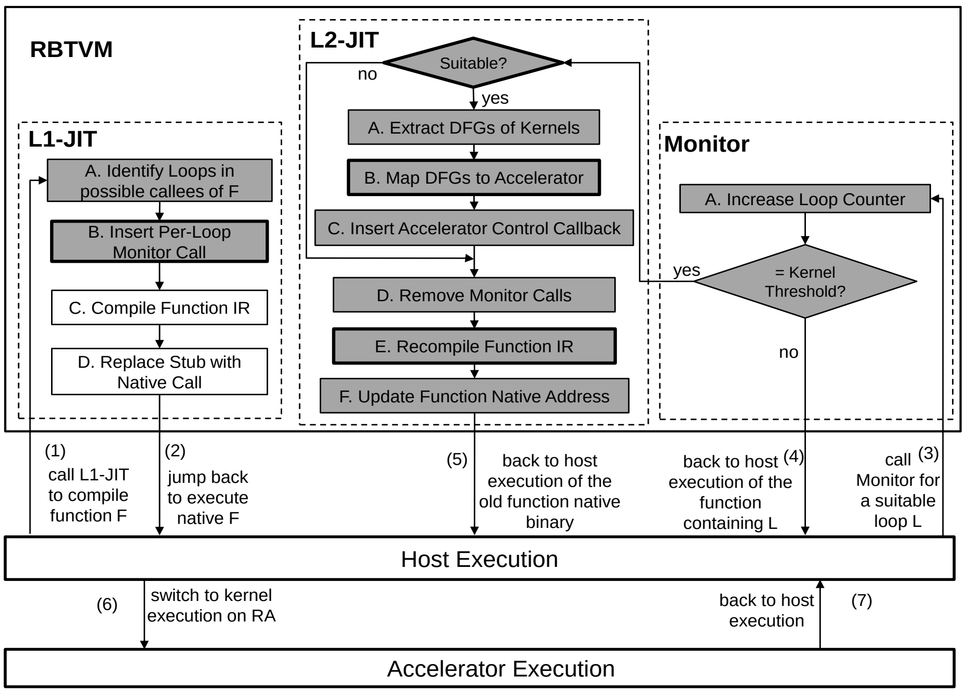

RBTVM
L1-JIT
A. Identify Loops in possible callees of F
B. Insert Per-Loop Monitor Call
C. Compile Function IR
D. Replace Stub with Native Call
L2-JIT
Suitable?
no
yes
A. Extract DFGs of Kernels
B. Map DFGs to Accelerator
C. Insert Accelerator Control Callback
D. Remove Monitor Calls
E. Recompile Function IR
F. Update Function Native Address
Monitor
A. Increase Loop Counter
= Kernel Threshold?
yes
no
(1) call L1-JIT to compile function F
(2) jump back to execute native F
(5) back to host execution of the old function native binary
back to host execution of the function containing L (4)
call Monitor for a suitable loop L (3)
Host Execution
(6) switch to kernel execution on RA
back to host execution (7)
Accelerator Execution

Figure 3.1: Runtime Binary Translation Virtual Machine design.

3.1.1 The Design of L1JIT

The main responsibility of *L1JIT* is to instrument loops in the application IR with monitor callbacks, that will be used to profile the loops to detect kernels. The *L1JIT* is actually the extension of the base JIT (see Section 2.1). The first new subcomponent in L1JIT compared to the base JIT is *L1JIT.A* which is used for getting all the loops' IR in the *possible callees* of a function F. The second new subcomponent *L1JIT.B* will then insert a monitor callback right before the IR of each of these loops.

The remaining two subcomponents of L1JIT are what the L1JIT inherits from the base JIT. They are used for compiling the IR of the function F to the main processor (i.e., the SP of SPIRA) native binary code, and replacing the stub call at the call site in the native binary code of the *caller* of F, with the address of the actual native binary code of the function F.

3.1.2 The Design of Monitor

The *Monitor*, as shown in Fig. 3.1, is designed to be a light-weight profiler of loops and a trigger of the *L2JIT*. Everytime before a loop is run, the Monitor is called by the monitor callback (inserted by the L1JIT) to increase a counter associated with that loop. If the counter reaches a certain *Kernel Threshold* value, the loop is detected as a kernel. The Monitor then passes the corresponding kernel information to L2JIT, and invokes it.

3.1.3 The Design of L2JIT

The *L2JIT* is where the most compilcated jobs of the RBTVM are done. Whenever invoked by the Monitor, the L2JIT first checks if the given kernel is suitable for running on the accelerator. Since we target the SPIRA architecture [9] which contains a CGRA accelerator, the criteria for determining well-suited kernels that we are using include: i) the number of iterations of the kernel must be loop-invariant, and ii) there must be no function call in the kernel body. When the kernel is suitable, the subcomponent *L2JIT.A* proceeds to extract the DFG from the kernel body and feed that to *L2JIT.B*. *L2JIT.B* does the actual mapping from the DFG to accelerator configuration.

After the translation, *L2JIT.C* inserts the accelerator control callback for the kernel in the IR. By this time, the corresponding monitor call of the kernel is no longer necessary. It is removed from the IR by *L2JIT.D* before *L2JIT.E* recompiles the IR of the function containing the kernel, and *L2JIT.F* updates the memory address of the native binary code of that function, so that in the subsequent invocations of the function, the new function binary code is used and the execution of the kernel will be offloaded onto the accelerator.

3.1.4 How RBTVM Works

In this subsection, we use an example in Fig. 3.2 to illustrate how the RBTVM works. Assume that the application in this example has multiple functions, among which are F1, F2, and F3. We initially run the application IR on top of the RBTVM. At some point in time during the execution (T_a), F1 is already compiled to the native binary code and executing, while F2 and F3 are still in IR and not compiled to native binary code yet. When the statement "stub(F2)" of F1 is executed, it means that F1 needs to call F2, and F2 will be compile to the native binary

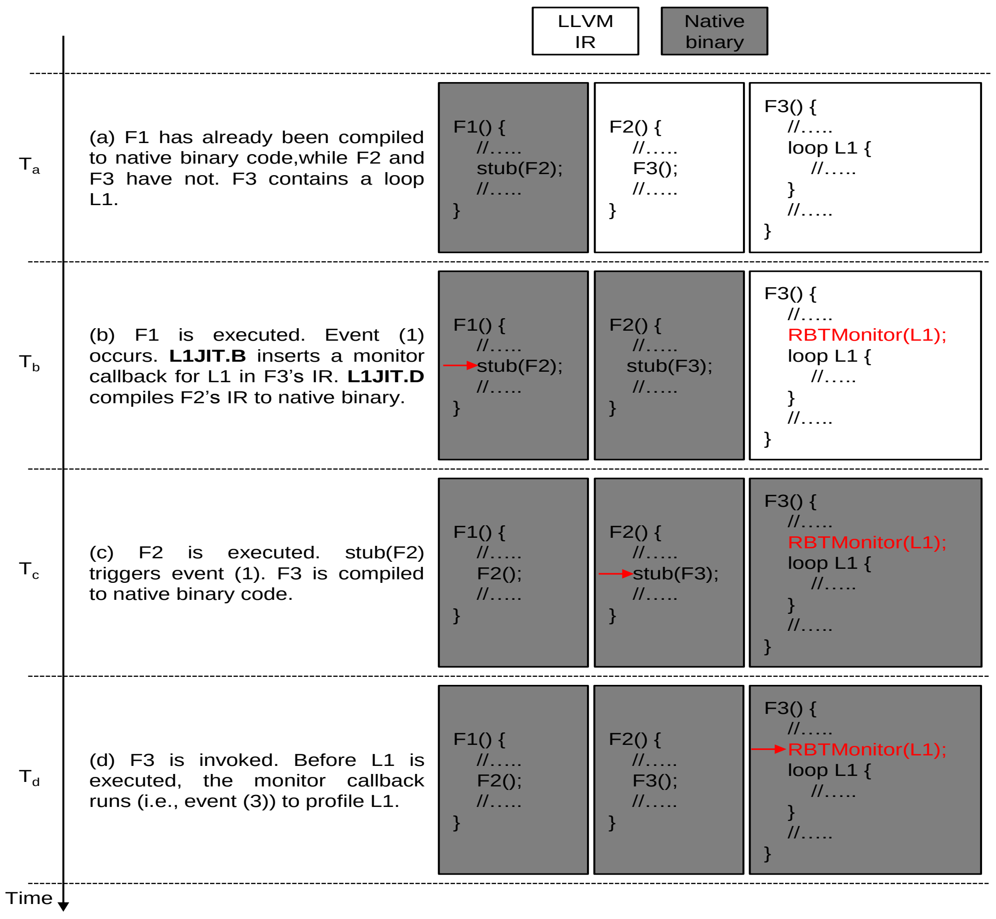

LLVM IR
Native binary
Ta
(a) F1 has already been compiled to native binary code,while F2 and F3 have not. F3 contains a loop L1.
F1() { //..... stub(F2); //..... }
F2() { //..... F3(); //..... }
F3() { //..... loop L1 { //..... } //..... }
Tb
(b) F1 is executed. Event (1) occurs. L1JIT.B inserts a monitor callback for L1 in F3's IR. L1JIT.D compiles F2's IR to native binary.
F1() { //..... stub(F2); //..... }
F2() { //..... stub(F3); //..... }
F3() { //..... RBTMonitor(L1); loop L1 { //..... } //..... }
Tc
(c) F2 is executed. stub(F2) triggers event (1). F3 is compiled to native binary code.
F1() { //..... F2(); //..... }
F2() { //..... stub(F3); //..... }
F3() { //..... RBTMonitor(L1); loop L1 { //..... } //..... }
Td
(d) F3 is invoked. Before L1 is executed, the monitor callback runs (i.e., event (3)) to profile L1.
F1() { //..... F2(); //..... }
F2() { //..... F3(); //..... }
F3() { //..... RBTMonitor(L1); loop L1 { //..... } //..... }
Time

Figure 3.2: How Runtime Translation works.

code. Right at that time, event (1) occurs (see arrow (1) in Fig. 3.1) and the *L1JIT* starts to run.

While doing the compilation for F2, *L1JIT* finds out that F3 is a possible callee of F2. *L1JIT.A* then finds all the loops in the IR of F3. After L1 is found by *L1JIT.A*, *L1JIT.B* inserts a monitor callback right before L1. Then, *L1JIT.C* does the actual compilation of F2. When the compilation is done, *L1JIT.D* replaces the instruction that calls stub(F2) in F1 native binary code with the actual call instruction to F2 native binary code. After being compiled, at T_c, F2 is executed and stub(F3) is called, causing the L1JIT to run for F3. Unable to find any possible callees of F3, *L1JIT.C* just moves on to compile F3 and replace the stub of F3 in F2's binary code. After that, at T_d, F3's native binary code is executed and RBTMonitor callback is called, causing *Monitor.A* to run, increasing the counter for loop L1.

During the execution of the application, the *Monitor* might be called many times for L1. When the counter of L1 reaches the *Kernel Threshold* value, the *Monitor* invokes the L2JIT callback. *L2JIT* first checks the suitability of L1 before running the subcomponents *L2JIT.*{*A, B, C, D, E, F*}. Those subcomponents of L2JIT will translate L1's body to accelerator configuration, recompile the F3 IR to native binary code, and update F3 binary code's address, so that the next time F3 is called, the execution of L1 is offloaded onto the accelerator.

3.2 Implementation of the RBTVM

As described above, the *RBTVM* consists of three main components: *L1JIT, Monitor*, and *L2JIT*. For the L1JIT, the implementation of *L1JIT.A* is pretty straightforward where it just needs to repeatedly picks the innermost loops in the possible callees of a function and passes those to *L1JIT.B*. For the implementation of the subcomponent L1JIT.B, a callback to the *Monitor* is inserted at the end of the preheader of each loop IR. The arguments to the monitor call are i) the information (ID) that uniquely identifies the kernel (e.g., the kernel header IR address in the memory), and ii) the address of the RBTVM in the memory. The implementation of the two subcomponents *L2JIT.C* and *L2JIT.D* is reused from the Base JIT (see Section 2.1).

The Monitor contains a map mapping from the kernel ID, which is set by L1JIT.B, to a record containing some metadata of the corresponding kernel, which also include the profiling counter of the loop. When called, the Monitor just gets the record corresponding to the kernel ID and increase the counter. When the counter reaches the *Kernel Threshold* value, the Monitor invokes the L2JIT.

For the L2JIT, the implementation of the kernel suitability checking is as follows. To check if a loop has loop-invariant number of iterations, the *ScalarEvolution* analysis pass in the LLVM framework is used. The loop has loop-invariant number of iterations if the analysis pass can compute a scalar expression of the number of iterations based on the related variables in the application program. For the second criterion, examining each instruction inside the loop body is required to make sure the loop contains no function call.

For each suitable kernel, the *L2JIT.A* goes through the define-use instruction chain in the loop body, makes a data-flow graph (DFG) structure, and feeds that into *L2JIT.B*. *L2JIT.B* implements the Edge-centric Modulo Scheduling (EMS) algorithm [4], which uses modulo scheduling to allocate accelerator resources in a periodic fashion, with a primary focus on the routing between the operations.

The implementation of the subcomponent *L2JIT.C* actually inserts a function call in the preheader of the loop IR, which later will be compiled into a callback to a special function in the RBTVM, that performs the accelerator control before the kernel execution. The implementation of *L2JIT.F* is interesting. Since updating the native binary code address of a function at all of its callsites is not an easy thing to do, instead, we reuse a method existing in the LLVM framework to put a forward instruction (i.e., a branch instruction) to the new function binary code, overwriting the first binary instructions of the old function binary code. That means, all the calls to the old function binary code together with the arguments will be forwarded to the new function binary code.

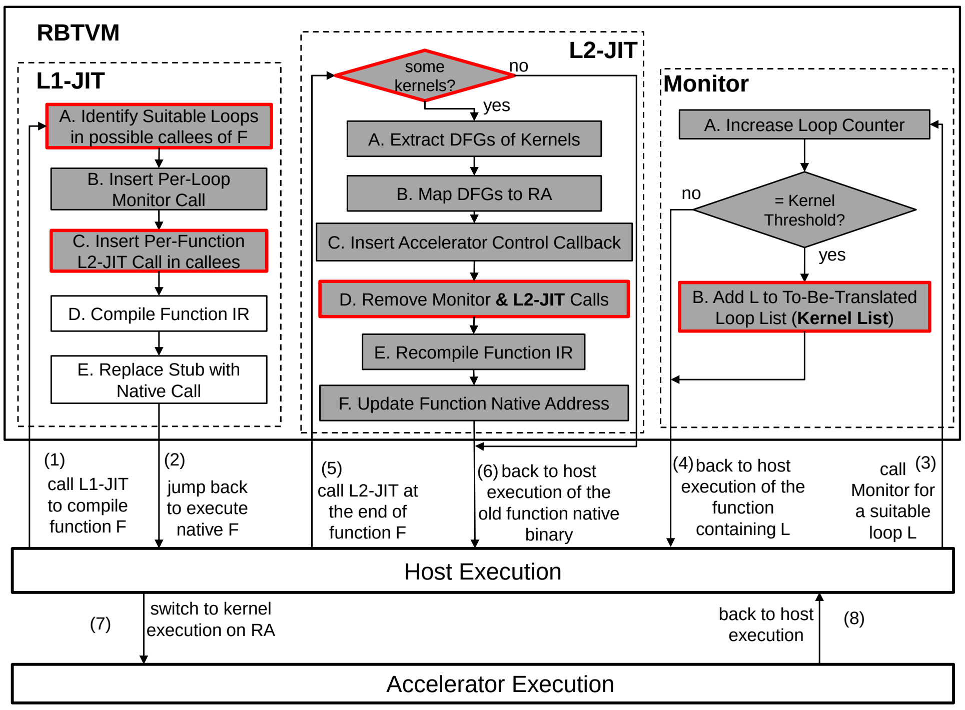
3.3 Optimizations for the RBTVM

Noticing some shortcomings in the RBTVM design in Fig. 3.1, we come up with some optimizations for i) reducing the monitoring overhead, and ii) avoiding the redundant recompilations of functions. The optimized design is shown in Fig. 3.3, in which the components in the red line boxes are the parts that we modify or newly add compared to the previous design in Fig. 3.1. In this section, we are going to explain how these optimizations can reduce the runtime overheads of the RBT system.

3.3.1 Reducing Monitoring Overhead

As described in Section 3.1, after a loop is detected as a kernel by the *Monitor*, the *L2JIT* will check if that loop is suitable for running on the accelerator or not before doing the translation. We observe that this flow has some issue. A loop which is not guaranteed to be suitable is still instrumented by *L1JIT*, profiled by Monitor and then finally checked by the L2JIT. If that loop turns out to be unsuitable, the cycles spent on monitoring become a waste.

Recall that in the previous design in Fig. 3.1, the *L1JIT.A* identifies the loops in the possible callees of a function F and inserts a monitor call in front of each loop. In this optimization, as shown in Fig. 3.3, we force the L1JIT.A to check for the suitability of loops before allowing the L1JIT.B to instrument it with monitor calls. Consequently, loops that are unsuitable are never instrumented and never profiled. That also means that the monitoring overhead is reduced compared to the previous design. We call the flow for checking the suitability in the previous



RBTVM
L1-JIT
A. Identify Suitable Loops in possible callees of F
B. Insert Per-Loop Monitor Call
C. Insert Per-Function L2-JIT Call in callees
D. Compile Function IR
E. Replace Stub with Native Call
L2-JIT
some kernels?
no
yes
A. Extract DFGs of Kernels
B. Map DFGs to RA
C. Insert Accelerator Control Callback
D. Remove Monitor & L2-JIT Calls
E. Recompile Function IR
F. Update Function Native Address
Monitor
A. Increase Loop Counter
= Kernel Threshold?
no
yes
B. Add L to To-Be-Translated Loop List (Kernel List)
(1) call L1-JIT to compile function F
(2) jump back to execute native F
(5) call L2-JIT at the end of function F
(6) back to host execution of the old function native binary
(4) back to host execution of the function containing L
call (3) Monitor for a suitable loop L
Host Execution
(7) switch to kernel execution on RA
back to host execution (8)
Accelerator Execution

Figure 3.3: Runtime Translation Virtual Machine optimized design.

design *L2-suitable*, and the optimized flow introduced in this section *L1-suitable*. We discuss the overhead reduction introduced by L1-suitable with experimental results in Section 5.2.1.2.

3.3.2 Avoid Redundant Recompilation of Functions

As shown in Fig. 3.1, the recompilation of a function IR is done by *L2JIT.E immediately* after the translation of a kernel contained by that function takes place. We refer to this recompilation technique in the initial design as *Immediate Recompilation*. We can see that if the function contains multiple kernels detected within a same invocation of that function, the function will still be recompiled immediately everytime each kernel is translated, although the newly produced binary code of that function will not be used until the next invocation. Therefore, the Immediate Recompilation technique can lead to redundant recompilations overhead.

To avoid the redundant recompilations of a function which has multiple kernels detected within the same invocation of a function, we introduce a technique called *Lazy Recompilation*.

The basic idea is that the RBTVM will keep tracks of the kernels detected during the execution of a function, and will call the L2JIT to do the translation and recompilation only at the end of that function. As shown in Fig. 3.3, we modify the *Monitor* so that whenever it detects a kernel, it will just add that kernel to a *To-Be-Translated Loop List* without immediately calling the L2JIT like in the initial RBTVM design. And the callback to the L2JIT will be made just before the function returns.

Since the function needs to make a call to L2JIT, we support this by adding the new component *L1JIT.C* to the design in Fig. 3.3, so that a call instruction to the L2JIT is inserted at the end of the IR of every function that contains suitable loops. We also modify the L2JIT flow as follows. L2JIT will now check if there is some kernels in the To-Be-Translated Loop List or not before proceeding. The component *L2JIT.D* is modified so that, in addition to removing the monitor callback of the detected kernels, L2JIT.D will also remove the L2JIT callback at the end of the function IR if all the kernels in the function have already been detected and translated.

The *Lazy Recompilation* technique can potentially reduce the redundant recompilation overhead. However, we should be aware of some technicality related to the nested function calls that could make the To-Be-Translated Loop List keep tracks of the kernels from different functions. In the implementation, we need to somehow separate kernels from different functions to make sure only kernels of the currently executed function are translated when calling the L2JIT.

CHAPTER IV

Install-time Binary Translation

In the previous chapter, we explain the RBT approach which dynamically instruments and profiles loops, and translates kernels at the runtime. In this chapter, we describe the *Installation-time Binary Translation (IBT)* approach which is a simpler version of the RBT approach. The IBT approach is different from the RBT approach in that it does the translation for all suitable loops right at the install-time instead of at the runtime, using an *Install-time Translator.* At the runtime, the *IBT Virtual Machine (IBTVM)* will be used to execute the original IR, together with the accelerator configuration produced at the install-time. Compared to the *RBTVM*, the IBTVM has a much simpler design and incurs less runtime overhead.

4.1 High-Level Design of IBT

4.1.1 Install-time Translator

Fig. 4.1 shows the design of the IBT tool flow. At the install-time, the Install-time Translator takes the IR of an application as the input. The translator iterates over the functions in the IR and analyze the loops inside. Using the same set of criteria as used in the RBT approach, the translator determines whether a loop is well-suited for running on the accelerator or not. The set of criteria includes i) the number of iterations of the loop must be loop-invariant, and ii) there must be no function call in the loop body.

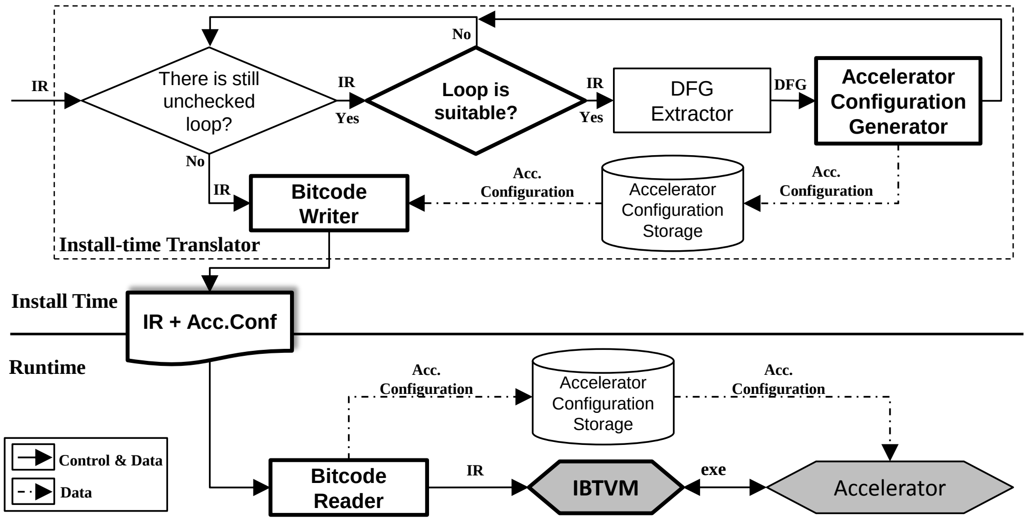


Figure 4.1: Install-time Binary Translation tool flow.

If the loop is suitable, the translator proceeds to extract the Data-Flow Graph (DFG) from the loop body and feed that to the configuration generator to generate the accelerator configuration (i.e., accelerator binary code) for that loop. This accelerator configuration is stored in a storage in the memory for later use. Following the same steps, the translator carries out the translation for all other loops. After the translation is done for all the suitable loops in the IR, the Bitcode Writer component in the translator loads the IR and the accelerator configuration from the storage and mixes them down to a single bitcode file.

4.1.2 IBTVM

As can be seen from Fig. 4.1, at the runtime, the Bitcode Reader parses the bitcode file produced at the install-time by the Install-time Translator into two parts, the IR and the accelerator configuration. The accelerator configuration of all the suitable loops are stored in a storage in the memory. The design of the IBTVM, as shown in Fig. 4.2, is much simpler than the RBTVM (see Section 3.1). This is because the IBTVM does not have to support the runtime features such as profiling the loops execution, translating loops to accelerator configuration, or recompiling the functions.

For each function F, the IBTVM just needs to instruments the suitable loops in the possible callees of F, which have already been translated at the install-time, with corresponding accelerator control callbacks. Then the IBTVM proceeds to compile F and replace the stub calls at

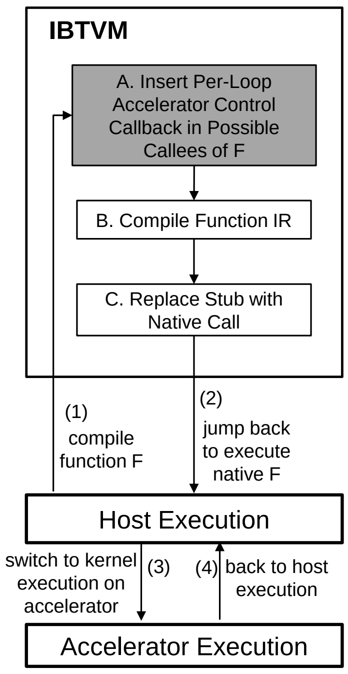

IBTVM
A. Insert Per-Loop Accelerator Control Callback in Possible Callees of F
B. Compile Function IR
C. Replace Stub with Native Call
(1) compile function F
(2) jump back to execute native F
Host Execution
switch to kernel execution on accelerator
(3)
(4)
back to host execution
Accelerator Execution

Figure 4.2: Install-time Binary Translation Virtual Machine design.

all of F's call sites the with actual function calls to the F's binary code. This flow guarantees that right before a suitable loop is executed, the accelerator control callback will be triggered and the execution of the loop will be offloaded onto the accelerator.

4.2 Implementation of IBT

The implementation of the *Install-time Translator* is separate from that of the *IBTVM*. The Install-time Translator is implemented as an LLVM Pass that will be loaded by the LLVM Optimizer tool at the install-time and will operate on the input IR. The checking of loop suitability is implemented similarly to the way the checking of kernel suitability is implemented in the RBTVM (see Section 3.2).

For each suitable loop, the DFG Extractor goes through the define-use instruction chain in the loop body and make a graph structure and feeds that into the Accelerator Configuration Generator. Similar to the component *L2JIT.B* in the RBTVM (see Fig. 3.1), the configuration generator implements the Edge-centric Modulo Scheduling (EMS) algorithm [4], which uses modulo scheduling to allocate accelerator resources in a periodic fashion, with a primary focus

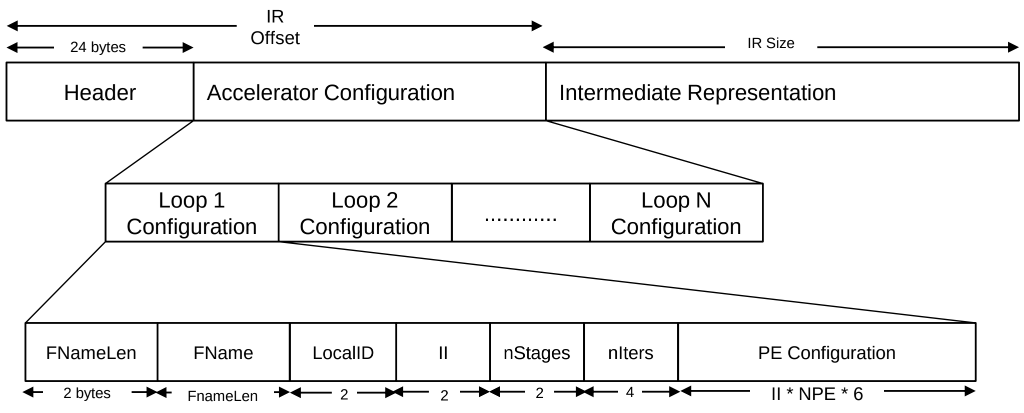


Figure 4.3: Accelerator configuration and IR wrapper format. The header is 24 bytes long consisting of *Magic number*, *Version*, *IR Offset*, *IR Size* and *CPUType*, each of which is 4 bytes long. The *Magic number* is 0x0B17C0DE, the *Version* and *CPUType* are not important. *FNameLen*: function name length; *FName*: function name; *LocalID*: local ID of the loop within FName; *II*: Initiation Interval; *nStages*: number of stages; *nIters*: number of iterations.

on the routing between the operations.

To store the accelerator configuration alongside the IR, the LLVM bitcode file format needs to be changed. Based on the wrapper structure of the LLVM bitcode, we suggest an accelerator configuration structure as shown in Fig. 4.3. The bitcode now contains a header that indicates the offset to and size of the original IR. Consequently, the serialization and deserialization processes in the Bitcode Writer and Bitcode Reader are also changed to support the bitcode format modification. The accelerator configuration structure will be put between the header and the embedded IR. The Accelerator Configuration Storage is implemented as a map mapping from *FName* and *LocalID*, which are the information for uniquely identifying a specific loop, to the corresponding configuration of that loop in the memory.

The accelerator configuration, as shown in Fig. 4.3, consists of the configuration of all the loops translated at the install-time. Each *Loop Configuration* in turns, encompasses several loop-specific information such as the number of iterations (*nIters*), the initiation interval (*II*), *PE Configuration*, etc. It is important to note that for the *nIters* field, we assume that we can determine the fixed number of iterations of each suitable loop right at the install-time. For each cycle in the *II* cycles, each PE is configured with an instruction consisting of an *opcode* (4 bytes) and *stage* (2 bytes). Therefore, the *PE Configuration* field of the *Loop Configuration* is $II * NPE * 6$ bytes long. Our experimental results in Section 5.2 show that the size of the additional accelerator configuration is small compared to the size of the original bitcode file.

The implementation of the IBTVM is almost the same with that of the base JIT (see Section 2.1). The only thing to note is that the insertion of the per-loop accelerator control callbacks must be done at the runtime. This is because what we insert are actually the callbacks to a special function that runs in the IBTVM context, which is responsible for running the loop-specific accelerator control instructions, and the address of this special function is only available at the runtime.

CHAPTER V

Experiments

In this chapter, we attempt to quantify various aspects of the RBT and IBT approaches, in order to provide an idea of the efficiency of the two approaches in the role of an infrastructure that enables making use of a CGRA accelerator to speed up kernels when executing application IRs. We first describe the setup that we use for the experiments. Then we evaluate the RBT and IBT approaches against several criteria, such as *runtime improvement*, *runtime overheads*, etc. We also evaluate the effectiveness of the optimizations we apply for the RBT approach, and finally, we compare the two approaches against each other.

5.1 Experimental Setup

We base our implementation of the RBT (see Section 3.2) and the IBT (see Section 4.2) on LLVM [2] version 3.1 (svn revision: 155980). Since LLVM version 3.1 provides full JIT support for only X86 and PowerPC targets, we choose PowerPC because it has a simpler instruction set. For the RBTVM implementation, to determine if a loop is a kernel, we set the *Kernel Threshold* at 50 times. Also, the RBTVM will avoid spending too much cycles on the kernel translation by dropping every kernel, the DFG of which contains more than 78 nodes.

For the evaluation, we use a chain of simulators including QEMU [12] PowerPC full system, PSIM [13], and Dinero IV Cache simulator [14]. We set the clock rates of the SP and the RA accelerator at 400MHz and 600MHz, respectively. Due to some limitations in the current implementation as well as in the simulators, we make the following assumptions. First, the actual control and execution of the RA accelerator is not modeled, so we estimate the kernel execution cycle count on the RA based on the loop mapping results obtained from the EMS [4] algorithm. Second, since we are targeting the SPIRA architecture [9] which uses hardware-double-buffered SPM, the DMA and computation can be overlapped. Hence we assume that the DMA overhead is hidden.

We use benchmarks from Mibench [15] (*cjpeg, djpeg, blowfish encoder/decoder*, and *gsm*) and Mediabench [16] (*mpeg2dec*). The benchmarks are compiled to LLVM IR *bitcode* using *Clang*, the front-end of the LLVM framework. To make the analyses at the runtime more effective, we turn on some compiler optimizations such as *-loop-simplify* for canonicalizing the loops IR, *-indvars* for canonicalizing induction variables, etc.

5.2 Experimental Results

5.2.1 RBT Evaluation

For the RBT approach, we evaluate the *runtime improvement* and the *effectiveness* of the optimizations. We compare four cases: *BaseJIT* as the baseline, *RBT-l2imm*, *RBT-l1imm*, and *RBT-l1lazy*. The BaseJIT case uses the base JIT (see Section 2.1), which does not have any accelerator support. The latter three cases runs the *RBTVM* with different levels of optimization, and dynamically makes use of the accelerator at the runtime.

In the case of *RBT-l2imm*, our initial RBTVM design (see Section 3.1) without any optimization. The RBTVM in the case of *RBT-l1imm* is with the first optimization turned on (i.e., checks for the loop suitability right in L1JIT instead of in L2JIT, see Section 3.3.1). The RBTVM used in the *RBT-l1lazy* is the most optimized one, with the second optimization (i.e., uses *Lazy Recompilation* instead of *Immediate Recompilation*, see Section 3.3.2) also turned on, in addition to the first optimization.

5.2.1.1 Runtime Improvement

In this evaluation, we vary the number of application runs (n_{runs}) to each application in {100, 500, 1000}. Fig. 5.1 shows the total run time speedup of the RBT cases over the BaseJIT case. We can see a general trend that we get better runtime improvement as n_{runs} increases. Also, we get better speedup as we apply more optimizations. On the average, across all benchmarks and different values of n_{runs}, the runtime improvements of the three RBT cases *RBT-l2imm*, *RBT-l1imm*, and *RBT-l1lazy* are 1.32, 1.41, and 1.44 times speedup, respectively. The *RBT-l1lazy* case, which runs the most optimized RBTVM, achieves the best runtime improvement.

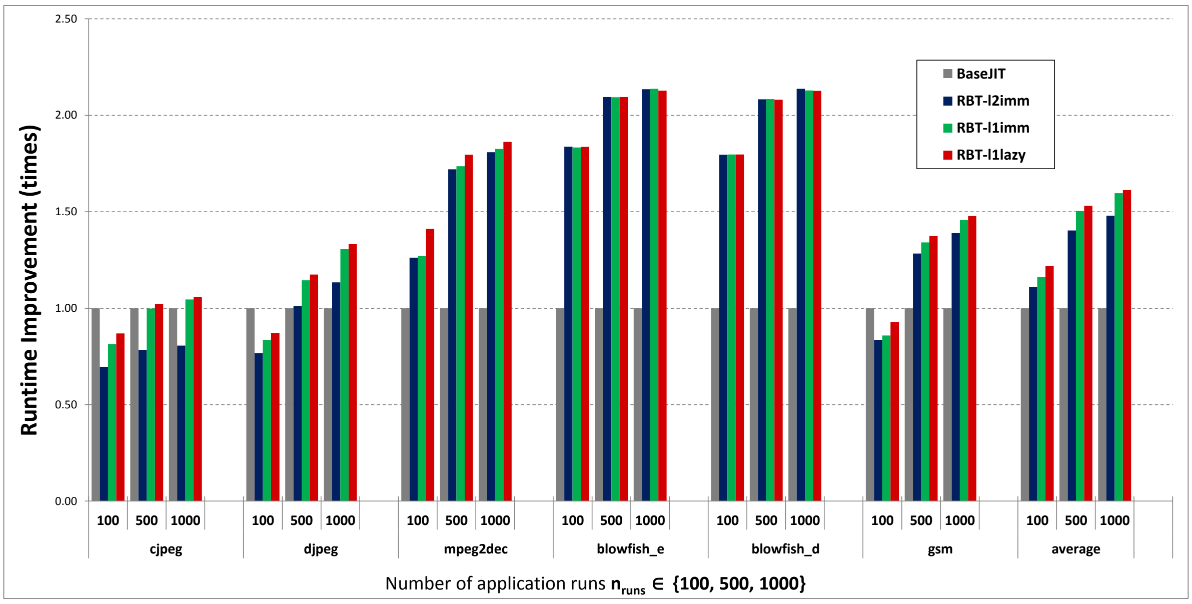


Figure 5.1: Runtime improvement of applications on the RBT system.

To see what leads to the runtime improvement, we go further and break the runtime down into three parts: *sequential execution*, *kernel execution* and *runtime overheads*, which are shown shown in Fig. 5.2. For the BaseJIT case, the *kernel execution* is on the SP, while in the three RBT cases, it is offloaded onto the accelerator. We see that the main factor leading to the runtime improvement is the kernel speedup, which is about 5.88 times on the average, across all benchmarks and different values of n_{runs}.

From Fig. 5.2, we also see that when increasing the number of application runs, the portion of the execution parts (including *kernel* and *sequential*) in the total runtime tend to go up. This together the with the kernel speedup explains why the runtime improvement gets better as n_{runs}

increases, as discussed previously. The kernel speedup is similar across the RBT cases because they use the same algorithm for mapping from the kernel DFGs to accelerator configuration. However, since the runtime overheads are different for each case, the runtime improvement are also different.

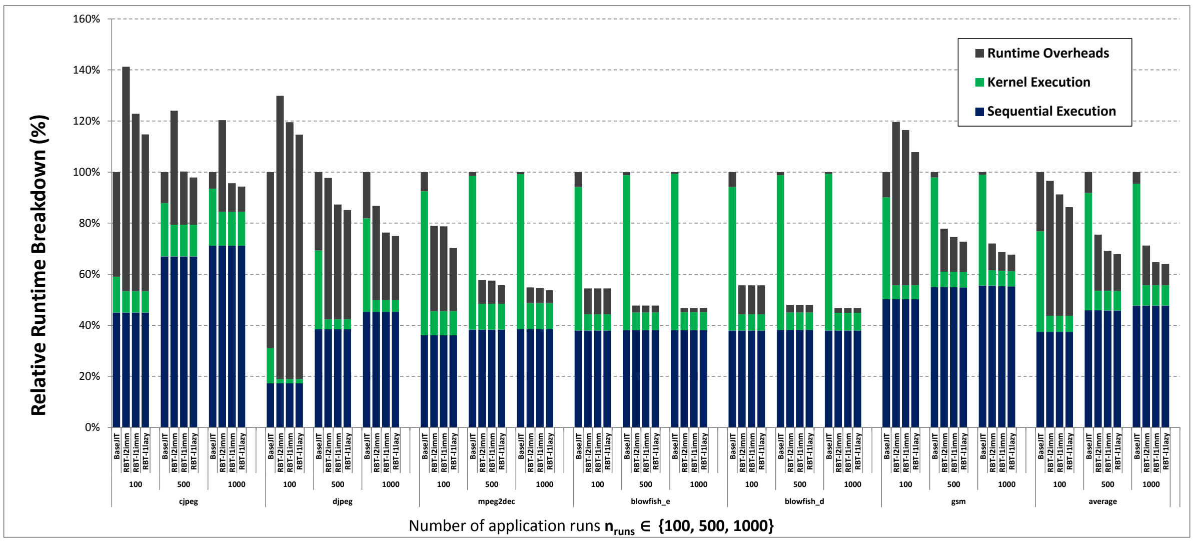


Figure 5.2: Relative runtime breakdown of the BaseJIT system and the RBT system. For each number of inputs, 100% represents the runtime of the BaseJIT.

5.2.1.2 Effectiveness of the Optimizations for RBTVM

To evaluate the effectiveness of the optimizations for RBTVM, we break the runtime overheads down into three parts: *L2JIT-Translation*, *L2JIT-Recompilation*, and *Other Overheads*, as shown in Fig. 5.3. *L2JIT-Translation* is the time spent on the mapping of DFGs to accelerator configuration, while *L2JIT-Recompilation* is the time spent on the recompilation of functions (see Section 3.1.3). The *Other Overheads* include *Monitoring* (i.e., profiling) and *Context Switching* overheads. One thing to note is that in this graph, we exclude the *L1JIT-Compilation* overhead which is the time spent on the first compilation for each function, and which is almost the same for all cases.

As described, the first optimization is turned on in the third case (i.e., *RBT-l1imm*). From Fig. 5.3, it can be seen that compared to the second case (i.e., *RBT-l2imm*) which uses the initial RBTVM design, the *RBT-l1imm* case incurs much less *Other Overheads*. The reason is because as the first optimization is applied, the RBTVM drops all the unsuitable loops right in L1JIT, thus no cycles will be spent on the *monitoring* and *context switching* for those loops. On the average, across all benchmarks and different values of n_{runs}, *RBT-l1imm* reduces the *Other Overheads* by 75.00%.

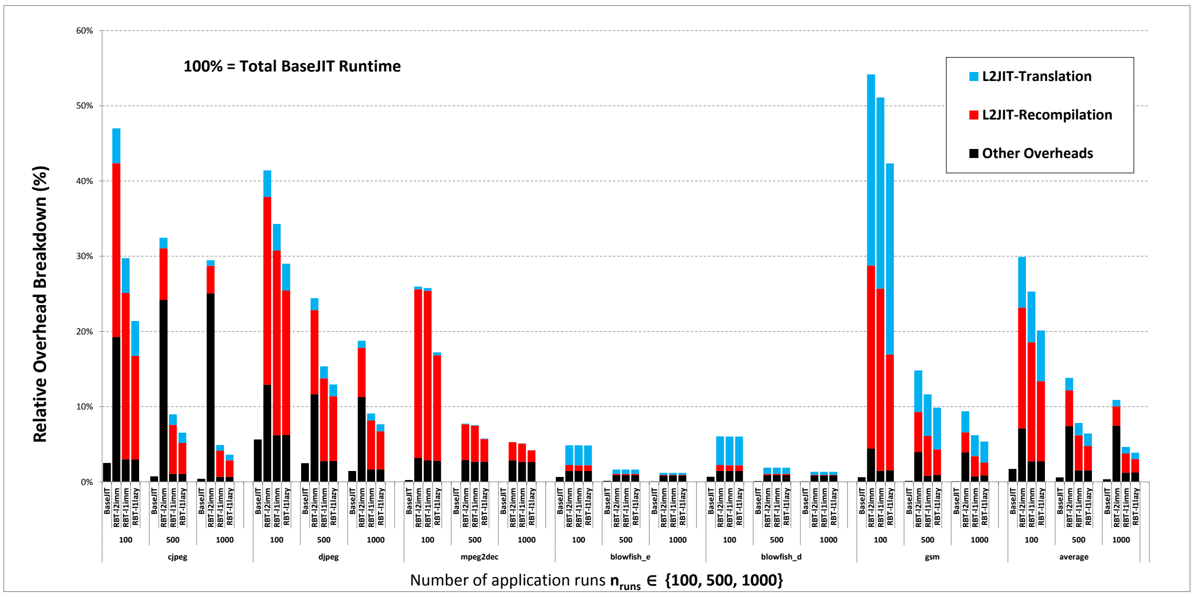


Figure 5.3: Relative runtime overheads breakdown of the BaseJIT and RBT systems. For each number of inputs, 100% represents the total runtime of the BaseJIT.

When the second optimization is also turned on in addition to the first optimization (i.e., the *RBT-l1lazy* case), the redundant recompilations of functions containing multiple kernels detected within the same invocation will be eliminated. This is why compared to the *RBT-l1imm* case, the *RBT-l1lazy* case reduces the *Recompilation Overhead* by almost 31.94% on the average, across all benchmarks and different values of n_{runs}. There are, however, exceptions in the two benchmarks *blowfish encoder* and *blowfish decoder*, where the *Recompilation Overhead* is almost the same for all three RBT case. The reason is because each of these two benchmarks contains only one kernel, so the second optimization we apply does not have any impact.

5.2.2 IBT Evaluation

For the IBT approach, we evaluate the following criteria: *runtime improvement*, *accelerator configuration size*, and *install-time increase*. Regarding the runtime improvement, we compare two cases: *BaseJIT* (i.e., the baseline, without any accelerator support) and *IBT*. The IR used in the case of BaseJIT is the original IR produced by the front-end (*Clang*), while the IR used in the IBT case is coupled with the accelerator configuration generated at the install-time.

5.2.2.1 Runtime Improvement

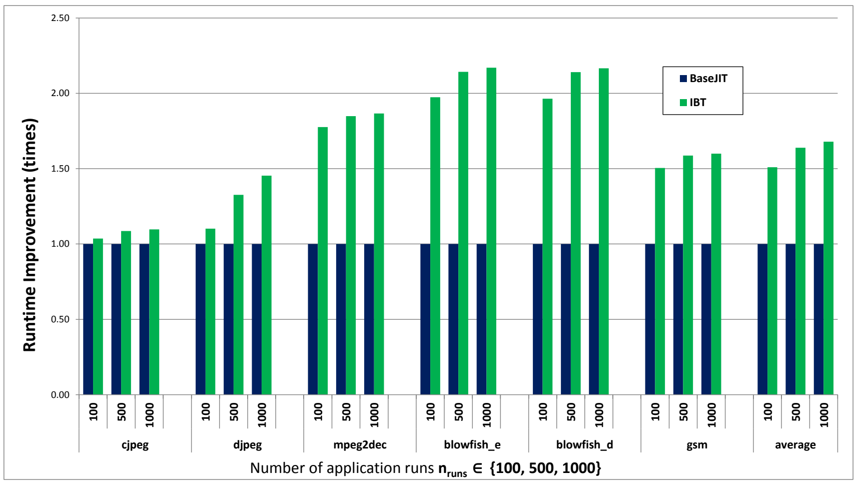


Figure 5.4: Runtime improvement of the IBT system.

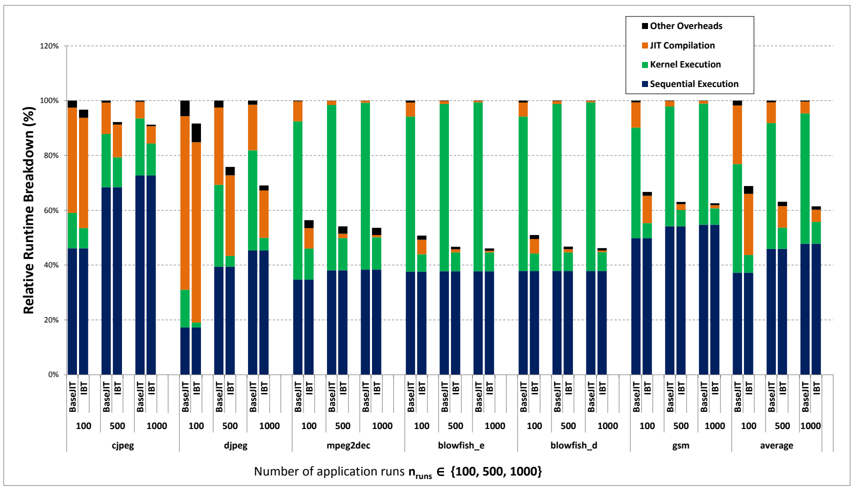


Figure 5.5: Relative runtime breakdown of the BaseJIT system and the IBT system.

For this evaluation, we also vary the number of application runs (i.e., n_{runs}) for each benchmark in $\{100, 500, 1000\}$. From Fig. 5.4, we can see that the IBT case improves the runtime by almost 1.61 times on the average across all benchmarks and values of n_{runs}, compared to the BaseJIT case. As shown in Fig. 5.5, we also break down the runtime of the two cases into 4 parts: *Sequential Execution*, *Kernel Execution*, *JIT Compilation Oerhead*, and the *Other Overheads* (which include the *context switching* and *instrumentation* overheads).

In the BaseJIT case, both the sequential and kernel parts are executed on the SP, while in the case of IBT, the kernel execution is offloaded onto the accelerator. Similar to what we see in the RBT evaluation, the main factor that leads to the runtime improvement of the IBT case is also the *kernel speedup*, which is about 5.88 times on the average, across all benchmarks and different values of n_{runs}.

5.2.2.2 Accelerator Configuration Size and Install-time Increase

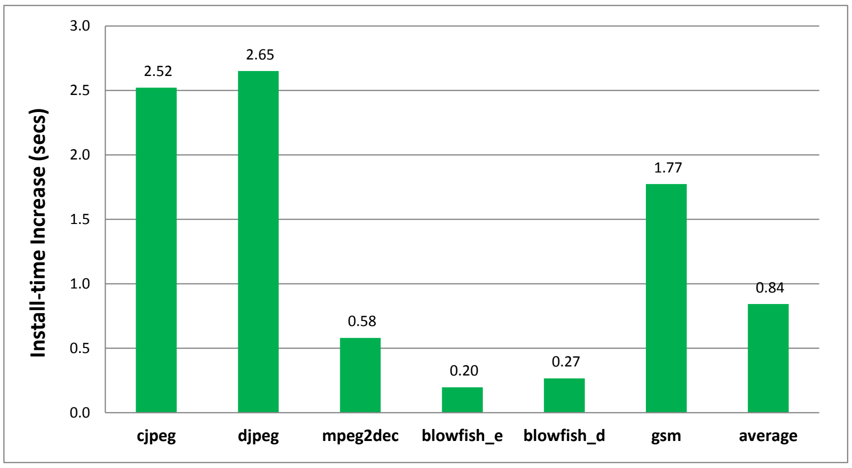


Figure 5.6: Install-time on the IBT system.

As explained in Chapter IV, the IBT approach translates all well-suited loops to accelerator configuration right at the install-time, and couples the configuration produced with the IR, so that the configuration can be used at the runtime. The install-time increase due to the translation is shown in Fig. 5.6, and the relative accelerator configuration size is shown in Fig. 5.7.

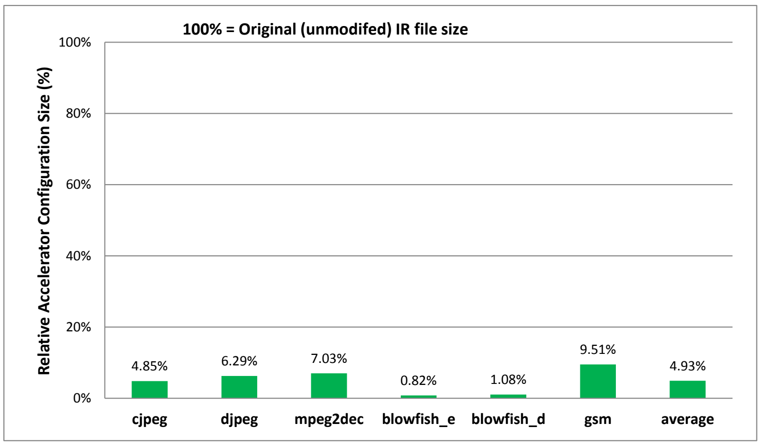


Figure 5.7: Accelerator configuration size relative to the whole IR file size.

From these graphs, we can see that the average install-time increase is only about 0.84 second. Also, the relative accelerator configuration size is only about 4.93% the size of the original IR on the average, across all benchmarks. Consequently, the *store* and *load* time of the configuration is very short. The store time is always less than 2% of the install-time increase, and the load time is always less than 1% of the total runtime.

5.2.3 Comparison between RBT and IBT

In this evaluation, we attempt to compare the RBT and IBT approaches in terms of runtime improvement and runtime overheads. There are three cases in the evaluation: *BaseJIT* (baseline, without any accelerator support), *RBT* (the best RBT case, i.e., *RBT-l1lazy*), and *IBT*. Fig. 5.8 shows the runtime improvement of RBT and IBT over the BaseJIT. We can see that the IBT can always improve the runtime, while the RBT sometimes slows down, especially when the number of application runs is small such as in the cases of *cjpeg*, *djpeg* and *gsm* with $n_{runs} = 100$. In these cases, the kernel speedup is not enough to compensate for the runtime costs the RBT spend. On the average, the runtime improvement over the BaseJIT of the IBT case is 1.61 times, which is 11.33% better than that of RBT.

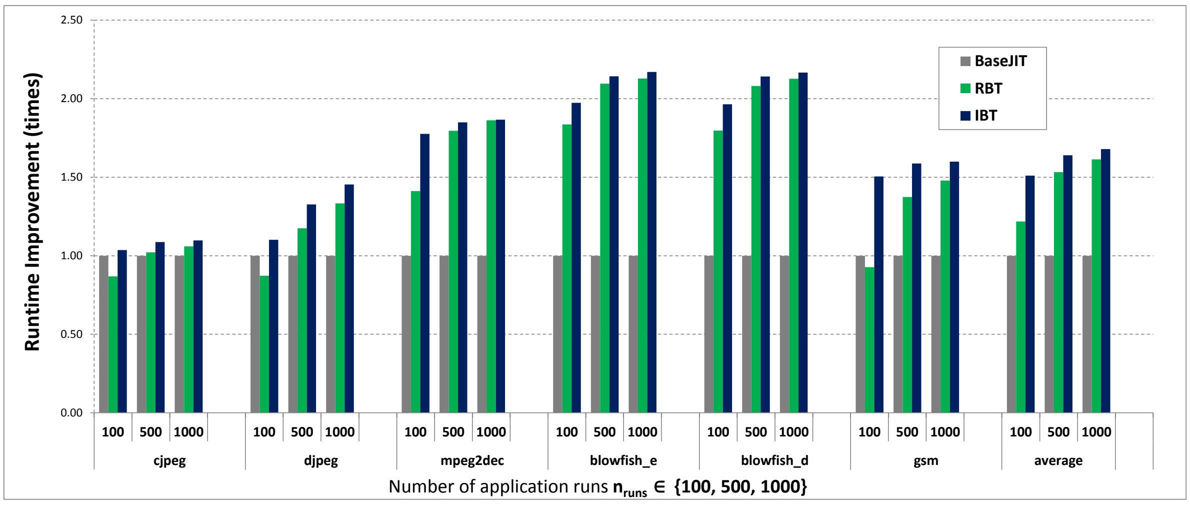


Figure 5.8: Runtime improvement on the RBT system and IBT system.

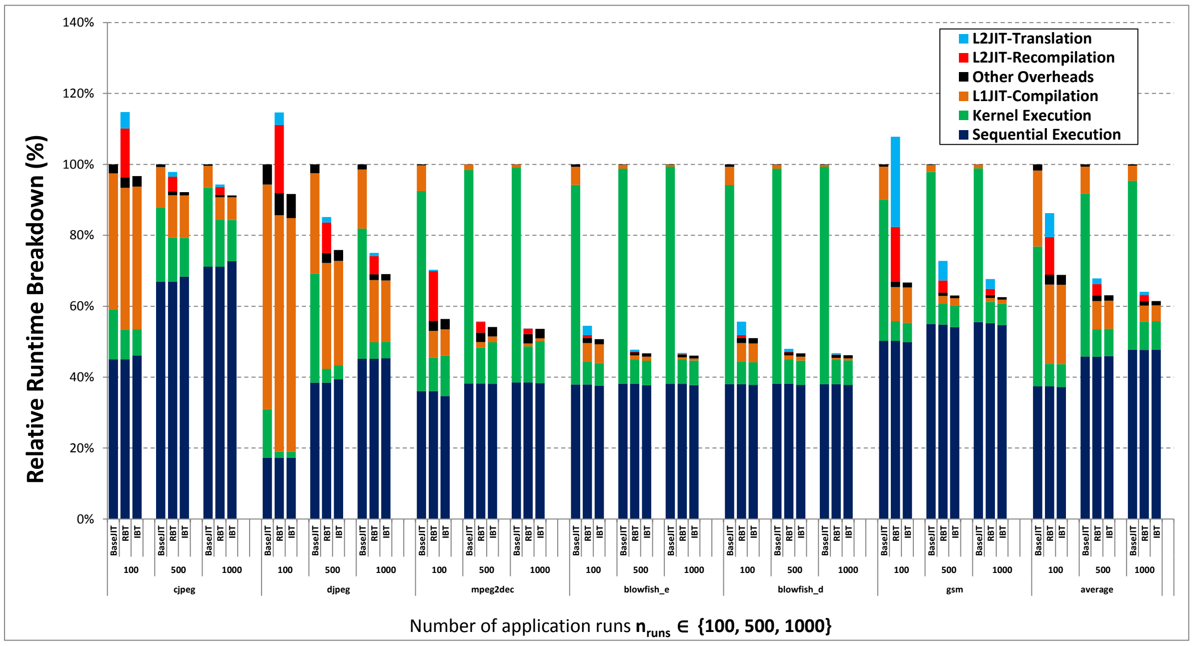


Figure 5.9: Relative runtime breakdown of the BaseJIT system, the IBT system, and the RBT system.

Finally, we compare the runtime overheads of RBT and IBT. Fig. 5.9 shows the relative runtime breakdown of the three cases. The runtime now consists of 6 parts: *Sequential Execution*, *Kernel Execution*, *L1JIT Compilation* (i.e., overhead of the first compilation of functions), *L2JIT Recompilation* (i.e., overhead of the recompilation of functions), *L2JIT Translation* (i.e., overhead of the kernel translation at the runtime), and *Other Overheads* (includes *Monitoring* and *Context Switching* overheads).

It can easily be seen from Fig. 5.9 that both IBT and RBT share the *L1JIT Compilation* overhead, and the *Context Switching* part of the *Other Overheads*. However, only RBT has to incur the *L2JIT Translation*, *L2JIT Recompilation*, and *Monitoring* overheads at the runtime. Therefore, on the average across all benchmarks and values of n_{runs}, the runtime overheads of the RBT case are 61.91% higher than that of the IBT case. In summary, the IBT case has higher runtime improvement and less runtime overheads than the RBT case.

CHAPTER VI

Conclusions

In this work, we propose the Runtime Binary Translation (RBT) approach to support making use of the CGRA accelerator to speedup the execution of application IR. The IR is run on the *RBT Virtual Machine (RBTVM)*, which dynamically identifies suitable kernels inside the application, translates them to the accelerator configuration, and offloads the execution of those kernels on to the accelerator. Since the RBTVM incurs a lot of runtime overheads, we also introduce two optimizations for reducing the unnecessary monitoring overhead and redundant recompilation overhead at the runtime.

We also simplify the RBT approach to come up with the *Install-time Binary Translation (IBT)* approach, which does the translation for all the suitable loops at the install-time instead of at the runtime. Experimental results show that, on the average, our RBT and IBT techniques can improve the runtime of the application IR by 1.44 times and 1.61 times, respectively, compared to the runtime on the JIT that does not support making use of the accelerator. In the comparison between RBT and IBT, we also show that IBT has 11.33% better runtime improvement over RBT, which has 61.91% higher runtime overheads than IBT. The IBT approach does increase the install-time and the IR file size, but the increase amounts are almost negligible, given the 1.61 times runtime improvement IBT brings with it.

For the future work, we intend to implement more mapping algorithms on our current framework to investigate the tradeoff between the translation cost and the mapping quality of those algorithms. Also, we intend to overcome the limitations in the experimental setup by extending the simulators to model the actual CGRA accelerator execution and control.

References

[1] Gosling, J., 1995. 'Java intermediate bytecodes: ACM SIGPLAN workshop on intermediate representations' (*IR*95). SIGPLAN Not. 30, 111-118. 1

[2] Lattner, C., Adve, V., 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, in: Proceedings of the *International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO* 04. IEEE Computer Society, Washington, DC, USA, p. 75. 2, 5, 21

[3] Mei, B., Vernalde, S., Verkest, D., De Man, H., Lauwereins, R., 2003. 'Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling', in: Proceedings of the *Conference on Design, Automation and Test in Europe - Volume 1, DATE* 03. IEEE Computer Society, Washington, DC, USA, p. 10296. 7

[4] Park, H., Fan, K., Mahlke, S., 2008. 'Edge-centric modulo scheduling for coarse-grained reconfigurable architectures', in: In Proc. of the *17th International Conference on Parallel Architectures and Compilation Techniques*. pp. 166-176. 7, 12, 18, 22

[5] Rau, B.R., 1994. 'Iterative modulo scheduling: An algorithm for software pipelining loops', in: In Proceedings of the *27th Annual International Symposium on Microarchitecture*. pp. 63-74. 7

[6] Greskamp, B., Sass, R., 2005. ‘A virtual machine for merit-based runtime reconfiguration’, in: *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* 2005. pp. 287-288. 7

[7] Beck, A.C.S., Carro, L., 2005. ‘Application of binary translation to Java reconfigurable architectures’, in: *Parallel and Distributed Processing Symposium* 2005. Proceedings. 19th IEEE International p. 8. 7

[8] Beck, A.C.S., Rutzig, M.B., Gaydadjiev, G., Carro, L., 2008. ‘Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications’, in: *Design, Automation and Test in Europe* 2008. pp. 1208-1213. 7

[9] Lee, J., Jeong, Y., Seo, S., 2013. ‘Fast Shared On-Chip Memory Architecture for Efficient Hybrid Computing with CGRAs’. *Proc. Int’l Conf. Design, Automation and Test in Europe (DATE)* 2013. 2, 7, 10, 22

[10] Lee, D., Jo, M., Han, K., Choi, K., 2009. ‘FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability’, in: *International Conference on Field-Programmable Technology*, 2009. pp. 376-379. 2, 7

[11] Mei, B., Vernalde, S., Verkest, D., Man, H.D., Lauwereins, R., 2003. ‘ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix’, in: *Field Programmable Logic and Application, Lecture Notes in Computer Science. Springer Berlin Heidelberg*, pp. 61-70. 2, 7

[12] http://qemu.org 21

[13] http://sourceware.org/psim/ 22

[14] http://pages.cs.wisc.edu/∼markhill/DineroIV/ 22

[15] Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B., 2001. ‘MiBench: A free, commercially representative embedded benchmark suite’. *2001 IEEE International Workshop* 2001. IEEE Computer Society, Washington, DC, USA, pp. 3-14. 22

[16] Lee, C., Potkonjak, M., Mangione-Smith, W.H., 1997. ‘MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems’, in: Proceedings of the 30th Annual ACM/IEEE *International Symposium on Microarchitecture, MICRO*. IEEE Computer Society, Washington, DC, USA, pp. 330-335. 22

Acknowledgements

In particular, I would like to express my sincere gratitude to my advisor, Professor Jongeun Lee for accepting me as his student. I would have never completed the Master's program and this thesis without his extensive and constant guidance and support. I own him my deepest thanks for his patience and his trust.

I would also like to express my appreciation to Professor Beomseok Nam and Professor Won-ki Jeong for their support, advice, and suggestions during the preparation of this thesis.

In addition, I would like to give thanks to my labmates and friends for their generous helps, both in study and in life.

Finally, and most importantly, I would like to thank my family, especially my parents for their encouragement, love, and care for me on every step of my life.