UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
The Arsenal Tool Chain for the GreenDroid Mobile Application Processor /

Permalink
https://escholarship.org/uc/item/1d22306v

Author
Jia, Fei

Publication Date
2013

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

The Arsenal Tool Chain for the GreenDroid Mobile Application Processor

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

in

Computer Science

by

Fei Jia

Committee in charge:

Professor Michael Bedford Taylor, Chair
Professor Jason Mars
Professor Steven James Swanson

2013

Copyright
Fei Jia, 2013
All rights reserved.

The thesis of Fei Jia is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2013

DEDICATION

To my beloved family, who support me all the time.

TABLE OF CONTENTS

Signature Page . . . iii

Dedication . . . iv

Table of Contents . . . v

List of Figures . . . vii

List of Tables . . . viii

Acknowledgements . . . ix

Abstract of the Thesis . . . x

Chapter 1 Introduction . . . 1
- 1.1 Motivation . . . 1
- 1.2 LLVM and the Arsenal Tool Chain . . . 3
- 1.3 Organization of the Thesis . . . 4

Chapter 2 Overview of GreenDroid . . . 5
- 2.1 The GreenDroid Architecture . . . 5
- 2.2 The Software-Hardware Interface . . . 7
- 2.3 Hardware Verification . . . 8

Chapter 3 Description of the Arsenal Tool Chain . . . 10
- 3.1 Introduction . . . 10
- 3.2 CC-IR and A-IR . . . 12
- 3.3 CC-IR Generation . . . 15
- 3.4 CC-IR Scheduler . . . 18
- 3.5 CCIR to AIR Backend . . . 20
  - 3.5.1 The Data Path Module . . . 20
  - 3.5.2 Basic Block Modules . . . 23
  - 3.5.3 The Control Unit . . . 26
  - 3.5.4 The Tree-Structured Pipelined Multiplexer . . . 27
- 3.6 AIR to C Backend . . . 27
- 3.7 AIR to V Backend . . . 29

Chapter 4 Calling Convention of C-Cores . . . 30
- 4.1 MIPS O32 ABI . . . 30
- 4.2 Argument Passing with a C-Core as a Callee . . . 31
- 4.3 Stack Frame of C-Cores as Leaf Functions . . . 33
- 4.4 Sub-function Calls in C-Cores . . . 34

4.5 Global Variables 36

Chapter 5 A C-Core for the Android Dalvik Garbage Collector 37
5.1 Introduction to Dalvik and Dalvik GC 37
5.2 Generation and Test Mechanism of the C-Core 38
5.3 The Generated C-Core 39

Chapter 6 Related Work 43
6.1 Dark Silicon 43
6.2 High-level Language to Silicon 44
6.3 Specialized Hardware for Garbage Collection 46

Chapter 7 Conclusion 47

Appendix A Format of A-IR Modules 49

Appendix B Standard Ports and Registers of an SPE 51

Bibliography 54

LIST OF FIGURES

Figure 2.1: The GreenDroid Architecture [GHSV+11] 6

Figure 3.1: The Arsenal Tool Chain 12
Figure 3.2: An Example of PHI Node and PHI Mov 19
Figure 3.3: An Example of PHI Mov and PHI Register 22
Figure 3.4: An Example of a GEP Instruction 25

Figure 4.1: MIPS O32 Stack Layout [Swe07] 33
Figure 4.2: Stack Frame of C-Cores as Leaf Functions 34
Figure 4.3: Stack Frame for C-Cores as Non-Leaf Functions 35

LIST OF TABLES

Table 3.1: The Transformation Passes Used in CCIRGen 15
Table 3.2: The Analysis Passes Used in CCIRGen 16

Table 4.1: Integer Register Usage Related with MIPS O32 ABI [Swe07] . . 32

Table 5.1: Statistics of `scanObject` CC-IR 39
Table 5.2: The Instruction Count of `scanObject` CC-IR 40
Table 5.3: `scanObject` C-Core Area 41
Table 5.4: `scanObject` C-Core Critical Paths 42

Table A.1: Fields of A Data-Path Module 49
Table A.2: Fields of A Control-Path Module 49
Table A.3: Fields of A Instance 50
Table A.4: Fields of An Assignment 50

Table B.1: The Default Inputs for SPE 51
Table B.2: The Default Outputs for SPE 52
Table B.3: The Default Registers for SPE 53

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my parents for their endless support for both my study and my life. The thesis is also dedicated partly to them.

I must give my high, respectful gratitude to my adviser Prof. Michael Taylor, who is also my committee chair, for his guidance, advice, and support throughout the project. His vision, passion, and hacking have always been inspiring me to do better job. Without his persistent help and support, the thesis would not have been possible. I would also like to thank my other two committee members, Prof. Steven Swanson and Prof. Jason Mars.

I would like especially to express my appreciation to Dr. Jack Sampson for his invaluable suggestions and patience. From the early stage of learning, to the final stage of thesis writing, he has been giving me precious and patient advice and assistance. I have learned quite much from the experience of working with him.

Nathan Goulding helped me hugely when I started hacking on the project, and his encouraging emails always inspired me. A thank you to Qiaoshi Zheng, one of the first several friends I made after I moved to San Diego, who has been helpful in both my work and life. I would also like to acknowledge all the other members in Prof. Michael Taylor's groups for their support and suggestions.

Discussion with Lu Zhang, Yanqin Jin, and other friends helped me greatly. Their passion, carefulness and knowledge are impressive. Lu Zhang provided great feedbacks on the process of thesis writing.

ABSTRACT OF THE THESIS

The Arsenal Tool Chain for the GreenDroid Mobile Application Processor

by

Fei Jia

Master of Science in Computer Science

University of California, San Diego, 2013

Professor Michael Bedford Taylor, Chair

In recent years, the utilization wall has become a serious problem that prevents processor performance from increasing. GreenDroid, a heterogeneous architecture, has been proposed to attack the utilization wall in the mobile domain. Conservation cores (C-Cores) are exploited in GreenDroid, and a C-Core is produced from Android application source code by an automated compiler tool chain.

This thesis examines the design of a new LLVM-based compiler tool chain for the GreenDroid architecture. It examines the choice of LLVM as the base compiler, provides an overview of the GreenDroid system, and discusses the design and implementation of the compiler tool chain. A C-Core generated from the Android Dalvik garbage collector is employed as a detailed case study.

Chapter 1

Introduction

1.1 Motivation

Currently, transistor densities and speeds continue to increase with Moore's Law, but limits on threshold voltage scaling have prevented the per-transistor switching power from scaling downwards. Meanwhile, the cooling capacity and power budget provided for processors keep relatively constant. As a result, the percentage of transistors that can be used simultaneously diminishes exponentially with each process generation, which is called the utilization wall [VSG+10]. The transistors that must stay inactive to satisfy the power constraint are called dark silicon [Tay12] [GSV+10].

To effectively use the dark silicon in the mobile domain, GreenDroid, an heterogeneous architecture to optimize energy efficiency, has been proposed [GHSV+11] [ST11]. The GreenDroid architecture is composed of arrays of tiles, and each tile contains one host processor and several application-specific cores, which are called conservation cores, or C-Cores. The C-Cores are produced automatically from the source code of android applications by a compiler tool chain. Simply speaking, a function of an application is picked first, and then transformed into a C-Core with the identical semantics. When the application runs, and the function is called, instead of executing the function code directly, the host processor transfers the control to the C-Core and reads the result directly from the C-Core when the C-Core finishes execution. Since the C-Core is a fixed-function hardware and runs

the function more energy-efficiently, the total energy consumed by the application drops.

A compiler tool chain is used by the GreenDroid architecture to generate C-Cores automatically from application source code. The tool chain contains several stages, a profiler to select proper functions to target for transformation, a compiler frontend to compile source code into intermediate representation (IR), multiple transformation and analysis passes to reconstruct and optimize the IR, and a backend to transform the IR into hardware modules.

In the old-version tool chain (termed the *C-Core tool chain*) [VSG+10], C is used as the IR for transformation and analysis, and several compilers are applied. GCC is used for preprocessing. LLVM [LA04] is applied for profiling and reconstitution of the source code, such as function inlining, deglobalization, etc. The LLVM C Backend is used to transform the LLVM IR back to C. CodeSurfer [BGRT05] [Tei00] helps with code analysis, and a set of in-house OpenImpact [Kid07] passes is employed to transform C to hardware modules. Raw-GCC (a ported GCC for the Raw architecture [TKM+02] [Tay99]) is used to compile application source code into the Raw assembly, which can invoke the C-Cores.

There are several drawbacks of the C-Core tool chain. First, multiple compilers are used, which impairs the maintainability and reliability of the tool chain. To make things worse, OpenImpact is not well-documented and no longer actively supported, and CodeSurfer is a commercial analysis tool for C programming language, which is neither open-source nor free. Second, C is not a good option as an IR for transformation and analysis, because backends like C++ to C, or LLVM IR to C are not well developed, and there are always bugs and unsupported features. Meanwhile, analysis and transformation passes for C code are included in various tools. Third, for validation purpose, hardware description code in C++ for BTL (the Raw simulator [TPS+04] [TJ03]) is generated along with Verilog code. Implementing two hardware backends, C to Verilog and C to BTL C++ (the hardware description C++ for the BTL simulator), separately causes code duplication and makes it hard to maintain. These drawbacks in the C-Core tool chain make us come up with the idea of using LLVM as the base infrastructure to design and

implement the Arsenal compiler tool chain.

1.2 LLVM and the Arsenal Tool Chain

Low Level Virtual Machine (LLVM) [LA04] is a compiler infrastructure with increasing popularity in both academia and industry, and is supported by Apple and University of Illinois at Urbana-Champaign. It is well modulized into three components: frontend, optimization, and backend. LLVM IR is the core of the infrastructure. Frontends, such as LLVM-GCC [Lat06] and Clang [Lat08], are used to transform various frontend languages, such as C, C++, fortran, Java, etc. to LLVM IR. Optimization includes transformation and analysis passes. Transformation passes are conversions from LLVM IR to LLVM IR, while analysis passes analyze and extra program information from LLVM IR. Backends transform LLVM IR to assembly code, such as MIPS, or X86 assembly code.

The use of LLVM as the base of the Arsenal tool chain allows us to use a single compiler to help with the analysis and transformation. There are a number of well developed transformation and analysis passes in LLVM which can be used directly. The multiple-language-support frontends allow the Arsenal tool chain to support a wide range of frontend languages, which increases the code coverage of the Arsenal tool chain significantly.

To reduce the code duplication in hardware backends, a hardware description IR (Arsenal-IR, or A-IR) is designed. LLVM IR will be transformed into A-IR first, and then several hardware transformation and analysis passes can be done on A-IR. At last A-IR modules are translated into Verilog or BTL C++ modules without modification of semantics. In this way, generation of different target hardware description languages from the same source file shared the same A-IR optimization flow, and therefore semantics equivalence is ensured and code duplication is minimized.

1.3 Organization of the Thesis

Chapter 2 provides a description of the GreenDroid architecture. In Chapter 3, the details of the Arsenal tool chain are provided. In Chapter 4, the importance of calling-convention compatibility of C-Cores to MIPS is explained, and design and implementation details for mips-compatibility are presented. Chapter 5 describes a C-Core generated from Dalvik Garbage Collector by the compiler tool chain as a case study. Chapter 6 discusses the related work, and Chapter 7 is a summary of the whole thesis.

Chapter 2

Overview of GreenDroid

To exploit dark silicon in mobile platforms, the GreenDroid architecture is proposed as a hardware solution. In the GreenDroid architecture, each host processor is tightly coupled with several conservation cores (or C-Cores). A C-Core is an application-specific fixed-function co-processor, which runs a function with at least the same speed as the host processor, but more energy-efficiently. C-Cores allow architects to trade area for energy-efficiency, and the trade-off is preferable in the regime of dark silicon [VSG+10].

In this chapter, first I provide an overview of the GreenDroid architecture from the hardware perspective. Then I describe the software/hardware interface in the GreenDroid architecture and how C-Cores are used by applications. Finally, I explain the mechanisms used in hardware verification for the GreenDroid architecture.

2.1 The GreenDroid Architecture

As is shown in Figure 2.1, the GreenDroid system is comprised of an array of one or more tiles, each of which is produced from the same template, dipicted in Figure 2.1a [GHSV+11] [GHSZ+12]. Figure 2.1b shows the architecture for one tile. In each tile, there is an in-order host processor (a MIPS core), a 32-Kb L1 data cache, a point-to-point mesh interconnect (on-chip network), and an array of energy-efficient co-processors, called Conservation Cores, or C-Cores. Each tile

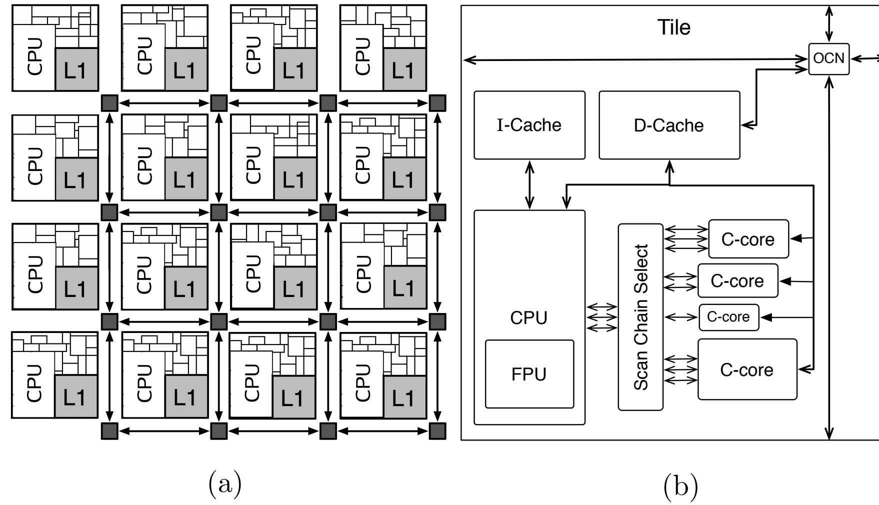


Figure 2.1: The GreenDroid Architecture [GHSV+11]

typically contains 8 to 15 C-Cores, which may or may not intersect with the set of C-Cores on other tiles in the sytem. The C-Cores are coupled directly to the host processor within the same tile through the L1 data cache and a specialized interface. With the interface, the host processor can address, read from and write to the registers inside all the C-Cores within the same tile [SAGH+11]. The on-chip network is used for memory traffic, and both host processor and C-Cores can send memory accesses through the on-chip network.

C-Cores are application-specific co-processors, and a C-Core corresponds to a function in a targeted application. C-Cores are generated automatically by the Arsenal tool chain from application functions based on a standard template. A standard template includes a control logic module, multiple basic block modules, registers, and a specialized interface to communicate with the host processor. A control logic module directs the execution flow among basic block modules and registers. It consists of a set of state machines, and the state machines resemble the control-flow graph of the targeted function [SAGH+11]. A basic block module is a data-flow circuit for one basic block [SVGH+11b], which contains only combinatorial logic that preserves semantics of the software block, and may have several memory requests. There are two different types of registers, state registers and data registers. State registers keep the states of C-Cores, and data registers

save the temporary value of cross-block variables when one basic block module finishes execution, and could be also viewed as the boundary of basic block modules. The specialized interface is used by the host processor to access registers inside a C-Core, and is implemented as a tree-structured, pipelined multiplexer.

2.2 The Software-Hardware Interface

As mentioned above, C-Cores are generated from application source code automatically by the Arsenal tool chain. First, a profiler is used to identify the hot spots in the source code of an application, which may be regions, such as functions, loops, or long traces of basic blocks. The hot spots are outlined to be hot-spot functions, and all the selected hot-spot functions are transformed into C-Cores by the compiler tool chain automatically. The generated C-Cores are integrated into the tiles to make the GreenDroid architecture.

In order for a targeted application to invoke the C-Cores, a C-Core aware GCC compiler. rgcc [TJ03], compiles the application source code into the GreenDroid-extended Raw assembly code [TKM+02]. In the assembly code, a C-Core related function call is compiled to be a trampoline. The trampoline first checks the state of the C-Core. If the C-Core is busy, the original function code is executed. Otherwise, the trampoline sets the value of registers in the C-Core, transfers the control to the C-Core and waits for the C-Core to finish. When the C-Core finishes execution, it notifies the host processor, and the trampoline copies the returned value from the C-Core to the host processor registers [SAGH+11].

Meanwhile, during the execution of C-Cores, exceptions or function calls may be raised for the host processor to handle. Exceptions include inner hardware faults, such division-by-zero fault, page fault and faults for patching. Patching is a mechanism used in the GreenDroid architecture to exploit similar code patterns within and across applications to ensure that a small set of specialized cores could support a large range of computations [VSG+11] [VSG+10]. Three facilities are used to support patching: configurable constants, generalized single-cycle datapath operators, and control flow changes [VSG+10], and the patching fault is for control

flow changes. When a patching fault is raised, the C-Core stops execution and transfers control to the host processor. The host processor extracts values from the C-Core, performs patching execution, writes values back to the C-Core, and resumes execution. Since the C-Core can be resumed at any point in the control flow graph (CFG), the control flow can be arbitrarily changed or replaced. Function calls are for C-Cores to call subroutines. When an inner hardware fault happens, the host processor reads the exception code from C-Cores and call an exception handler to deal with the exception. When a C-Core calls a subroutine, it prepares the content in the stack properly for the subroutines, raises the attention of the host processor, and then waits in an exceptional state. The host processor receives the attention signal, reads the states of the C-Core, and dispatches the subroutine call. When the subroutine call finishes, the host processor writes back the returned value of the subroutine call to the C-Core, and resumes the execution of the C-Core. In case of direct or indirect recursion, the C-Core may need to stash the value of registers to the stack, and later resume the values of the registers.

Both the Raw ISA [WTS+97] [TKM+02] [TPS+04] [BTLAA03] [KTMW03] [TKM+03] [AAB+97] [Tay04] [Tay99] and GreenDroid-extended Raw ISA are MIPS-compatible and use the MIPS O32 calling convention. Compared with the Raw ISA, the GreenDroid-extended version adds four new instructions: tree load/store, wait, and attention mask. Tree load/store is used by the host processor to access the registers inside C-Cores via the tree-structured multiplexer. Attention mask sets the mask for the attention bits from C-Cores, which is reserved for future concurrent support. Wait lets the host processor sleep and wait for the attention signal from C-Cores.

2.3 Hardware Verification

The Arsenal tool chain transforms Android application functions in C or C++ to Verilog modules, which are verified and later synthesized by IC design tools to produce chips. In order to verify the correctness of C-Cores described in Verilog, the BTL simulator and the Synopsys Verilog Compiler Simulator (VCS)

[Syn13] are employed. The BTL simulator is used to simulate the process of running applications on the GreenDroid architecture and generates cycle-by-cyle signal transition information of the C-Core ports. The Synopsys VCS takes the Verilog modules in, uses the input port information from the BTL simulator to drive the simulation, and compares the output port value with the output port information from the BTL simulator for verification. Energy efficiency is evaluated by placing and routing with Synopsys IC compiler, sample-based simulation and analysis with Synopsys PrimeTime.

In order for the BTL simulator to run, C-Cores described by BTL C++ are also produced besides the Verilog description. In BTL, each C-Core is described as a subclass inheriting from a common base class, and the factory method pattern is used to execute all the C-Cores. In each C-Core, two functions need to be implemented, calc and edge. The calc and the edge functions are called per cycle, while the calc function is called first to evaluate combinatorial logic, and the edge function is invoked later to update the values for all the registers. Concretely, all the modules except registers in a C-Core are evaluated in the calc function, and registers are updated in the edge function.

In the BTL simulator, the behaviors of all the host processors, C-Cores, memory systems, network, etc. are cycle-accurately simulated. Meanwhile, the host processors (MIPS cores) in GreenDroid run with a frequency of 1.5 GHz [VSG+10]. This means that 1.5×10^9 snapshots of port values are generated to simulate one second of actual running of the GreenDroid architecture, which takes too much disk space, and logging itself becomes a heavy overhead to the BTL simulator. To speed up the simulation while still maintaining the accuracy of verification, we only log a random sample of cycles of the tested C-Cores, and use the log for verification.

Chapter 3

Description of the Arsenal Tool Chain

The Arsenal tool chain is used to automatically generate C-Cores for the GreenDroid architecture. The tool chain is based on the Low Level Virtual Machine (LLVM) compiler infrastructure and is centered on two newly designed IRs: CC-IR and A-IR. The tool chain is composed of a frontend to generate CC-IR, a LLVM backend to transform CC-IR to A-IR, and two hardware backends, A-IR to Verilog, and A-IR to BTL C++. Two important features, patching and generalization [VSG+10] [VSG+11], which are also supported in the C-Core tool chain (the old-version tool chain) , are not discussed in the thesis since the design and implementation is similar to that in the C-Core tool chain and relatively independent from the other parts of the Arsenal tool chain.

In this chapter, I introduce the Arsenal tool chain first, and then I describe the two IRs used in the Arsenal tool chain in detail. In the remaining parts I explain the design detail of each component of the tool chain.

3.1 Introduction

The Arsenal tool chain is based on the LLVM compiler infrastructure. A LLVM frontend, Clang [Lat08], compiles C/C++ to LLVM IR with the optimization level of O3. Multiple LLVM analysis and transformation passes are written to

transform LLVM IR to CC-IR. A newly created LLVM backend transforms CC-IR to A-IR.

LLVM helps increase the code coverage and improve the reliability of the tool chain. The well structured framework and well defined IR of LLVM allow us to build our tool chain efficiently. With LLVM IR, the tool chain could support most frontend languages supported by the LLVM frontends, besides C or C++.

In the center of the compiler tool chain are two IRs, C-Core IR (or CC-IR) and Arsenal IR (or A-IR). CC-IR is a subset of LLVM IR with constraints and analysis results to help with hardware module generation. A-IR is a hardware description IR designed particularly for C-Cores, and is in XML form for easy generating and parsing. The idea of CC-IR and A-IR is from the optimization perspective. The optimizations for C-Core can generally be divided into two categories, software optimizations and hardware optimizations. Software optimizations map to compiler optimizations, such as loop unrolling, and common subexpression elimination. Hardware optimizations include more circuit-related approaches, such as hardware module replacement, where a fast module in a non-critical path may be replaced with slow but more energy-efficient modules. With CC-IR and A-IR, software optimizations can be performed on CC-IR with LLVM optimization passes, while hardware optimizations are applied to A-IR. Another benefit of using A-IR is that A-IR allows us to separate the hardware optimizer from hardware backends. The hardware optimizer does A-IR to A-IR transformations, and two backends, AIR to Verilog, and AIR to C++, directly interpret A-IR to hardware description languages without the modification of semantics. The isolation between hardware backends and the hardware optimizer also ensures the semantic equivalence between modules produced by different backends.

As shown in Figure 3.1, the compiler tool chain is divided into several components: CCIRGen, CCIRScheduler, CCIRToAIR, AIRToV, and AIRToC. CCIRGen contains a series of LLVM transformations and analyses that transform LLVM IR to CC-IR. CCIRScheduler schedules the instructions within basic blocks and generates states for instructions. CCIRToAIR is a LLVM backend, transforming CC-IR to A-IR with the scheduling information from CCIRScheduler. AIRToV

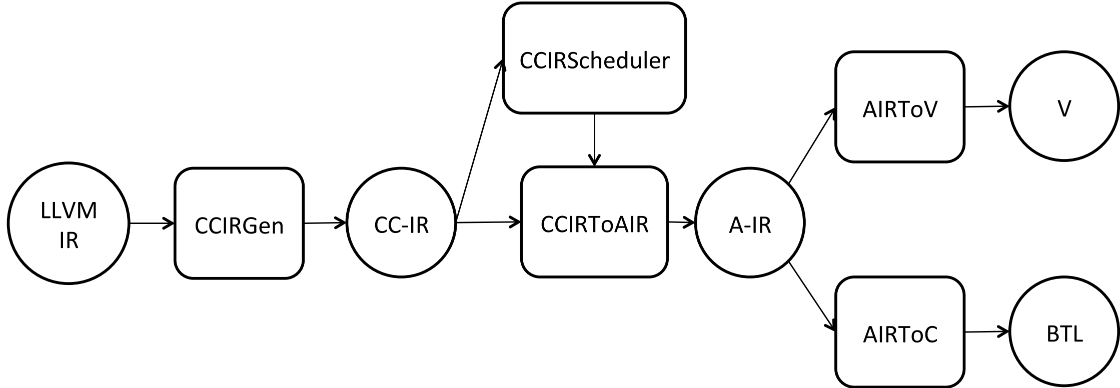


Figure 3.1: The Arsenal Tool Chain

and AIRToC are hardware backends, which transform A-IR to Verilog and BTL C++ respectively. Because A-IR is in XML form, AIRToV and AIRToC do not depend on the LLVM infrastructure for parsing.

3.2 CC-IR and A-IR

CC-IR is a subset of LLVM IR with constraints enforced by transformation passes and annotations added by analysis passes. CC-IR is the interface between LLVM IR and A-IR. On one hand, a CC-IR function has exactly the same semantics as the LLVM IR function. On the other hand, a CC-IR function could be directly mapped to an A-IR module.

LLVM IR is defined in static single assignment (SSA) [LA04]. In SSA, each variable is defined only once, and ϕ operators help select values for variables in the merging points [CFR+91]. In C-Cores, basic block modules follow the style of spatial computation [BVCG04], and the SSA form allows a CC-IR basic block to map to an A-IR basic block module easily. Meanwhile, the control-flow graph is explicitly presented in LLVM IR, which maps to part of the control logic unit. Moreover, the strict type system of LLVM IR help us identify the width of ports and wires in AIR. All of the above nice features make LLVM IR a good option as hardware behavior description IR.

There are also limitations of LLVM IR as a proper hardware behavior de-

scription IR, because LLVM IR is designed to be an RISC-like instruction set with higher level information for analysis [LA04]. First, LLVM IR is designed to be a target-independent representation, and some hardware features are not presented explicitly, such as how arguments are passed by argument registers or stacks, how global variables are accessed, and how arrays are allocated. Second, to provide a compact form, several instructions may be nested together in LLVM IR, which increases the complexity of backends. For example, constant memory addressing instructions may be nested into store or load instructions. Finally, some instructions like long-long-type memory accesses have not been supported directly by the architecture yet, but can be implemented by two int-type memory access instructions instead. It will be nicer to explicitly present in the IR how there instructions are indirectly supported. These limitations of LLVM IR make us come up with the idea of CC-IR.

CC-IR follows most parts of LLVM IR, but has some added rules and annotations to better describe the behavior and organization of hardware. One rule of CC-IR is that the CC-IR should express the hardware system organization if possible. According to this rule, the calling conventions should be expressed explicitly in CC-IR. Another important design principle of CC-IR is that each CC-IR instruction performs a single task that can be mapped to a primitive hardware module if possible. This principle simplifies the logic and reduces the code duplication in the CCIRToAIR backend, and also makes it easier for resource binding, where a template may be used. For example, we may say an *add* instruction in LLVM is mapped to an adder in AIR, and a *select* instruction is mapped to a 2-way multiplexer. With these rules, CC-IR is designed to be a more expressive and helpful hardware description IR.

A-IR is in the format of XML and is a hardware description IR at register-transfer-level (RTL) [TLW+89], which allows us to isolate the hardware optimizer from hardware backends. The hardware optimizer includes A-IR transformations and analyses, and there are two hardware backends used in the Arsenal tool chain. One is for A-IR to Verilog, and the other is for A-IR to BTL C++. All the hardware optimizations are done by the hardware optimizer, and the two hardware backends

do not change the semantics or structure of hardware modules described by A-IR.

An A-IR C-Core module is described by one A-IR XML file. There are several components contained in an A-IR module: a top-level data path module, a sequence of basic block modules, a control unit, several dynamically defined one-hot multiplexers, and a tree-structured pipelined multiplexer for register addressing and accessing by the host processor. Appended to an AIR C-Core module are two information nodes. One describes the multiple-cycle constraints for the C-Core, and the other defines the logic of how to deal with exceptions raised by the C-Core in the host processor.

A-IR modules are generally classified into two types: data-path modules and control-path modules. Data-path modules only contain combinatorial logic, which are purely defined by instances and interconnections between then, and there is no branching statement, or sequential logic. Control-path modules contain state machines, and branching statements and sequential assignments, and the order of assignments matters.

All the A-IR modules has a name, a list of port description, and a list of wire description. A data-path module also contains the description of instances and wires between instances, while a control-path module contains register/wire assignment lists, nested state machines, and conditional basic blocks.

The top-level data path module and all the basic block modules are described as data-path modules. The control unit, one-hot multiplexers, and the tree-structured multiplexers are defined as control-path modules. The reason to separate data-path modules from control-path modules is from the semantics and hardware optimization perspective. Control-path modules are mainly composed mainly of state machines and conditional blocks, and fewer optimizations can be done to improve their energy efficiency. Data-path modules are combinatorial circuits, and many hardware optimizations are applicable. For example, if a path is not the critical path in data flow model, part of it may be replaced with slower, but more energy-efficient, modules to improve the energy efficiency without reducing the performance.

3.3 CC-IR Generation

CCIRGen generates CC-IR from LLVM-IR, and this section describes the LLVM transformation passes and analysis passes developed to transform LLVM IR to CCIR. A list of transformations is shown in Table 3.1, and analyses are presented in Table 3.2.

Table 3.1: The Transformation Passes Used in CCIRGen

Name	Description
AddStoreForCallInstArg	Add store instructions before each call instruction to store arguments on the stack to support subroutine calls from C-Cores.
BreakConstantGEPs	Break nested constant GetElementPtr (GEP) instructions. GEP instructions are used for memory addressing.
LongLongMemAccessLowering	Lower long-long-type memory access to add support for 64-bit memory operations.
SetMipsCallConv	Enforce MIPS O32 calling convention.
SetNameForUnnamedVar	Give names to anonymous variables in LLVM IR.
SplitBlockOnCallInst	Split basic block on call instructions for subroutine call handling.

BreakConstantGEPs is an LLVM-specific transformation pass to eliminate the use of constant GetElementPtr (GEP) values in LLVM IR. GEP is a LLVM instruction for memory addressing [LA04]. Constant GEPs are nested into store or load instructions by the LLVM transformation pass ConstantFolding [LA06]. This makes a single instruction map to several primitive hardware modules, which is less straightforward, and increases the complexity of the CCIRToAIR backend. Thus, in CC-IR, the constant GetElementPtr values are unfolded to be separate

Table 3.2: The Analysis Passes Used in CCIRGen

Name	Description
BlockIDInfo	Give each basic block a unique ID.
PhiMovInfo	Push a PHI node up to be phi mov instructions in the predecessors.
StackOffsetInfo	Calculate the offset of arguments, local variables, and arguments for subroutines.

GetElementPtr instructions.

LongLongMemAccessLowering is used to support 64-bit store and load instructions in LLVM IR. Since the memory system in the GreenDroid architecture only natively supports 8-bit, 16-bit, and 32-bit memory operations, memory operations of 64-bit need to be lowered. In detail, a 64-bit load instruction is split into two 32-bit load instructions, and the two loaded 32-bit values are concatenated into the 64-bit value. A 64-bit store instruction is transformed into two 32-bit store instructions, the inputs of which are the higher and lower 32 bits of the initial stored value respectively. The support for the 64-bit memory operations is primarily for the long-long type, since double-precision floating-point number calculation is not supported by GreenDroid at the moment, and is rewritten to be single-precision in source code reconstitution.

Another constraint is that CC-IR function should comply the MIPS O32 / Raw calling convention explicitly [Swe07]. The constraint is enforced by the transformation pass SetMipsCallConv. Through this pass, each LLVM function is transformed into a CC-IR function with a signature based on the same template. The first four arguments represent the registers arg0 to arg3 in MIPS calling convention with the type of unsigned int. The fifth argument is argSP, which stores the stack pointer, and the sixth argument is argGP, which corresponds to the global pointer. Several casting and load instructions are added to the beginning of the CC-IR function to resume the value and type of original arguments. In this way, the MIPS O32 calling convention is enforced explicitly in a CC-IR function, and

we could generate hardware pieces straightforwardly based on CC-IR functions.

SetNameForUnnamedVar is used to give each anonymous LLVM variable a unique name. This allows us to name hardware wires based on LLVM variables. The SSA property [CFR+91] of LLVM IR [LA04] and well-designed naming rules ensure that there is no naming conflict among hardware wires or instances.

Finally, SplitBlockOnCallInst isolates function calls from other instructions to make each function call become an independent basic block ending with an unconditional branch instruction. In C-Cores, a function call is implemented as an exception raised by C-Cores to the host processor, and once the execution is transferred back to the host processor, an exception handler is called to decide how to process the exception. With function call isolation, there is no more than one function call in the same basic block, and the handler can decide which function to be called based on the ID of the currently executed basic block. Furthermore, since each function call is followed by an unconditional branch instruction, the execution of the C-Cores could be always resumed from the next basic block, which simplifies the C-Core execution resume logic in the host processor.

Some transformations may invalidate others. For example, SetMipsCallConv may add 64-bit load instructions, which need to be handled by LongLongMemAccessLowering. AddStoreForCallInstArg may add store instructions before call instructions in the same basic block, which need to be further separated by SplitBlockOnCallInst. With the help of the dependency graph of the transformation passes, we figure out a feasible order: SetMipsCallConv, LongLongMemAccessLowering, AddStoreForCallInstArg, BreakConstantGEPs, SplitBlockOnCallInst, SetNameForUnnamedVar.

There are three LLVM analysis passes. In LLVM, analysis passes analyze the IR code, generate and store the analysis result, which can used directly by LLVM transformation passes and LLVM backends. In CCIRGen, BlockIDInfo gives a unique ID to each basic block. StackOffsetInfo, which calculates and provides the stack offset information for arguments and allocation instructions, is used by both AddStoreForCallInstArg and SetMipsCallConv.

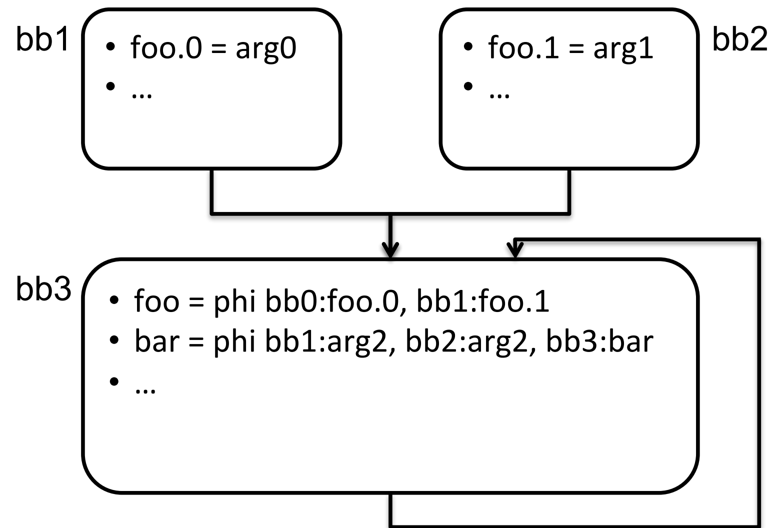
PhiMovInfo is used to transform PHI nodes into PHI mov instructions.

LLVM IR is defined in the form of SSA, and PHI nodes are used to select values for variables in the merging points. As shown in Figure 3.2a, *foo* and *bar* are both assigned by PHI nodes, and the value assigning to *foo* and *bar* depends on which predecessor the flow comes from. PHI mov are instructions generated to replace PHI nodes. Figure 3.2b shows the CC-IR with PHI mov that has the same semantics as the LLVM IR in Figure 3.2a. Each PHI node in LLVM IR is "pushed up" to the predecessors to become PHI mov instructions. PHI mov instructions simply move the value from source to the destination. One thing to notice is that the existance of PHI mov breaks the SSA form, so PHI mov are stored in an analysis pass to keep CC-IR still a legal LLVM IR. The idea of replacing PHI nodes with PHI mov instructions is from optimization perspective, and will be discussed later in Section 3.5.1.

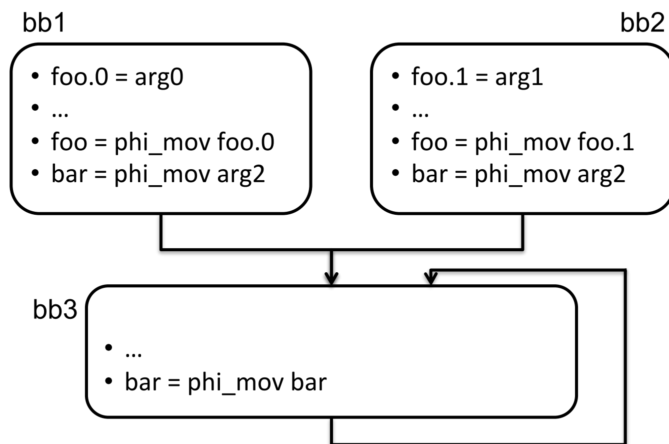
Analysis passes should come after transformation passes if they are not used by transformation passes. Here, StackOffsetInfo is used by two transformation passes, but the two transformation passes do not change the result of StackOffsetInfo, so the analysis result calculated by StackOffsetInfo can still be used in later stages.

3.4 CC-IR Scheduler

This section describes the scheduling algorithm in the Arsenal tool chain. The technique of selective depipelining [SAGH+11] is used to improve energy efficiency. There are two logical clocks used in C-Cores with different frequencies. The "fast" clock is used to drive memory request, and the "slow" one is used to control registers. Another way to look at it is that fast clock drives the execution logic inside a basic block, and the slow clock drives the execution between basic blocks, since memory requests are only raised in the middle of a basic block, and registers are the boundaries of basic block modules. The state of slow clock is called PC, while the state of execution inside basic blocks is called substate. So a pair of PC and substate specifies a unique state of C-Cores. PC represents the ID of the basic block module being executed, and substate represents the inner state



(a)



(b)

Figure 3.2: An Example of PHI Node and PHI Mov

of the basic block module.

CCIRSchedule is an analysis pass which calculates and stores substates of instructions inside basic blocks with a greedy algorithm. The detailed scheduling algorithm is described in [Sam].

3.5 CCIR to AIR Backend

CCIRToAIR is an LLVM backend transforming CC-IR to A-IR. It is composed of several parts: DataPathModuleGen, ControlPathModuleGen, BasicBlockModuleGen, TreeRegMuxGen, ExceptionHandlerGen and MultiCycleConstraintGen. DataPathModuleGen generates the top-level data-path description of a C-Core. ControlPathModuleGen produces the control unit, which drives the execution of the C-Core. BasicBlockModuleGen generates basic block modules. TreeRegMuxGen generates the tree-structured pipelined multiplexer used as the specialized interface between the host processor and the C-Core. ExceptionHandlerGen and MultiCycleConstraintGen generates ExceptionHandler node and MultiCycleConstraint node respectively. These two nodes do not directly describe the hardware implementation of C-Cores. ExceptionHandler describes a dispatcher called by the host processor to handle exceptions from C-Cores, and MultiCycleConstraint provides timing constraints for hardware pieces that are used.

3.5.1 The Data Path Module

DataPathModuleGen generates the top-level data-path module for a C-Core. In the GreenDroid architecture, all C-Cores are generated from the same template, and use the same standard ports for communication with the host processor and the memory system. A list of standard ports is provided in Table B.1 and Table B.2. A C-Core is composed of a control unit, several basic block modules, many registers, a tree-structured pipelined multiplexer, and a memory interface.

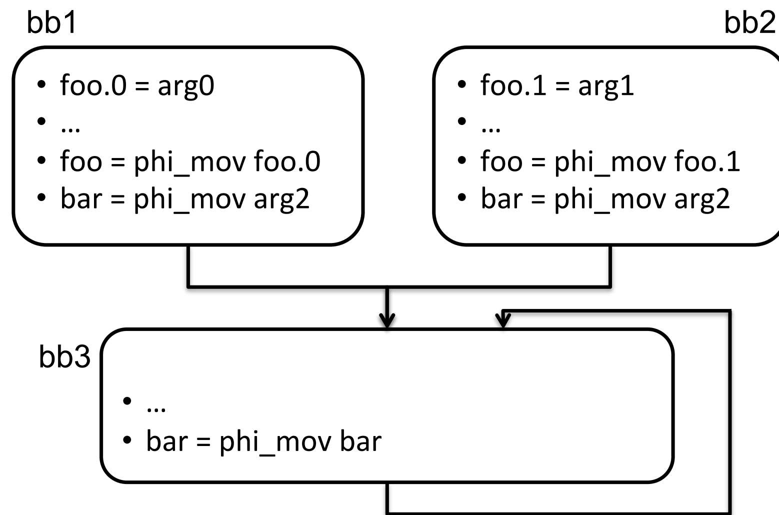
In C-Cores, registers are generally classified into two types, state registers and data registers. State registers include spe_status, PC, substate, and edge_id registers. Spe_status represents the execution status of a C-Core, namely done,

programming, running and waiting. PC is the ID of the basic block running, and substate is the inner state of a basic block. Edge_id stores the ID of the preceding basic block and the ID of the current block, and is used for exception dispatching.

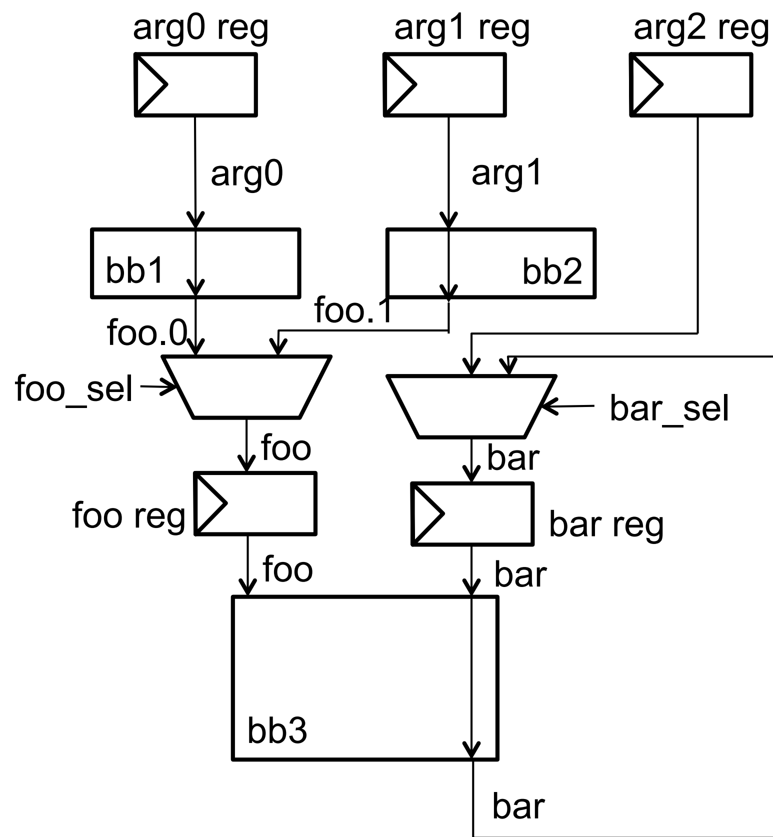
Data registers are argument registers, global variable offset registers, live-out registers, and PHI registers. Argument registers and global variable offset registers are written by the host processor to pass arguments to C-Cores. Both live-out registers and PHI registers serve to store the temporary values of cross-block variables when C-Cores are running. The difference is that PHI registers store the output of PHI mov instructions, and live-out registers store the value of other live-out variables. The PHI mov destinations with the same name are stored into the same PHI register through a multiplexer. The select signal of the multiplexer is asserted by the control unit based on PC and substate. Figure 3.3 gives an example of PHI registers. There are two PHI registers in Figure 3.3b, foo_reg and bar_reg. The input of foo_reg is from $bb1$ and $bb2$ through a two-way multiplexer, and the input of bar_reg is from $bb2$ and $bb3$ through another multiplexer. When the last substate of $bb1$ is reached, $foo.0$ is selected to be stored in foo_reg, and when the last substate of $bb2$ is reached, $foo.1$ is selected. Silimar routing scheme applies to bar_reg.

An alternative mechanism to deal with PHI nodes, which may be more straightforward, is to transform each PHI node into a multiplexer in the same basic block module where the PHI node is defined. The select signal is determined by the control path based on the preceding block ID. However, in this way we need to reason about the preceding block which may be very odd in exception or patching cases. Meanwhile, live-out registers need to be generated for each PHI node source instead of one PHI register for a PHI node. In general cases, intuition tells us that the number of registers will be much larger compared with the PHI mov mechanism, which will consume more power.

In the remaining parts of the section, I will explain basic block modules, the control unit, and the tree-structured pipelined multiplexer in detail.



(a)



(b)

Figure 3.3: An Example of PHI Mov and PHI Register

3.5.2 Basic Block Modules

A basic block module is produced for each basic block in CC-IR. The instructions of a basic block are implemented as a data-flow model, similar to that in spatial computation [BVCG04] [MCC+06].

The ports of a basic block module include live-in variables as input, live-out variables as output, several memory access related ports, and enable signals and state signals for the control unit. The data flow of a basic block starts from the live-in variables, namely outputs of live-out registers or phi registers, passes through inner instructions of basic blocks, and reaches the live-out variables, namely inputs of live-out registers or phi registers. By this means, the instructions of a basic block satisfy a strict data-flow model. In the end, the state information, usually the predicate value for conditional branch instructions or switch instructions, is sent to the control path module to decide which basic block module to be enabled next.

Instructions within basic blocks can be classified as several types: memory access instructions, binary operations, casting instructions, memory addressing instructions, and termination instructions.

Memory access instructions are stores and loads. Each load or store instruction is divided into two stages, request and confirm. In the request stage, for a load instruction, the memory address is sent to the memory system, and for a store instruction, besides memory address, value to be stored is also sent to the memory system. Then control unit waits in the substate of the confirm stage. When the memory system finishes the memory request, a valid signal is sent back to the C-Core, and then the control unit moves to the next substate. The valid signal resembles the memory ordering token in systems like Tartan [MCC+06] and WaveScalar [SMSO03].

Binary operations include arithmetic operations and logical operations. A schema is used for resource binding, which maps a binary operator to a hardware module. For most integer binary operators, a mapped module is instantiated separately, but for integer multiplication and division and floating-point operations, modules are shared among instructions within and across basic blocks, because

these operations consume much more power and area and generally appear much less frequently. To execute such a shared module, a request is sent to the module from basic blocks, and the state of basic blocks is handled in a similar way as that for memory access.

Casting instructions include extensions and truncations . Each of these instructions is mapped to a module with a single input and a single output, and source and destination width as parameters.

GetElementPtr (GEP) is a memory addressing arithmetic in LLVM IR. It is designed in a way that preserves the type information and allows machine-independent memory addressing [LA04]. In A-IR, a GEP instruction is translated into a sequence of adders and constant shifters depending on the machine data layout information, such as alignment and endian. For the example in Figure 3.4, the GEP instruction is used to calculate the offset of $arg0[arg2].r.c$. In 32-bit, byte-addressed, little-endian machine, the circuit of the GEP instruction is shown in Figure 3.4b. Two points to be noticed are that the size of struct *block* is 16 bytes, and the size of struct *row* is 8 bytes due to alignments, and the offset of c in a struct *row* is 0 because of the little endianness.

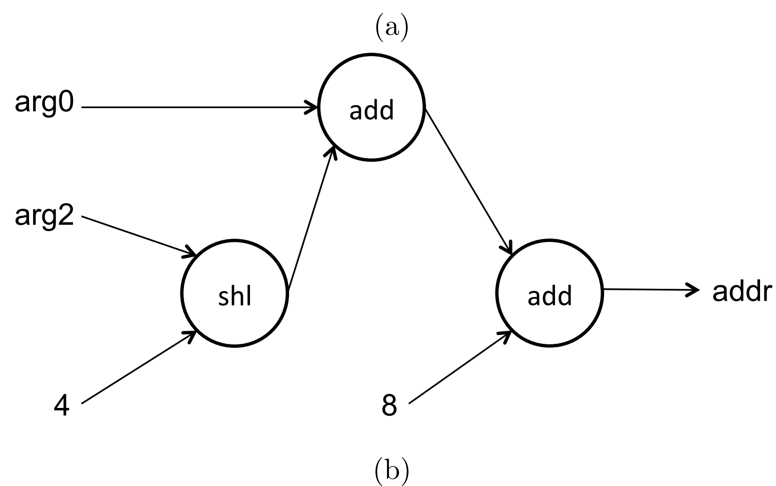
Allocation instructions allocate space on the stack for local static arrays. In C-Cores, stack offsets are allocated statically and in the run-time an allocation instruction only return the stack address of the allocated space. Thus, allocation instructions could also be viewed as memory addressing instructions, and the address is decided statically.

Basic Block termination instructions are conditional and unconditional branch instructions, switch instructions, and return instructions. For branch instructions or switch instructions, the values of the condition variables are passed to the control unit as state signals. For a return instruction, a signal is sent to the control unit to tell that the execution is finished. If the type of the returned value is not void, the returned value is stored in C-Core register V0, V1. On returning to the host processor, the values of C-Core register V0, V1 are copied to the V0, V1 of the host processor to comply with the O32 function calling conventions.

To deal with the memory ordering problem in the data-flow model of basic

```
; struct block { char c; int i; struct row r; } 16 bytes
%struct.block = type { i8, i32, %struct.row }
; struct row {char c; int i; }
%struct.row = type { i8, i32 } 8 bytes
; arg0[arg2].r.i
%addr = getelementptr %struct.block* %arg0
       , i32 %arg2, i32 2, i32 0
```

(a)



(b)

Figure 3.4: An Example of a GEP Instruction

blocks, the inner state of a basic block is kept by the control unit, and the memory requests are issued in order based on the inner state [VSG+10]. For example, assume there are three load instructions in a basic block, load1, load2, and load3. Each load instruction takes two substates, one for request and the other for confirm. The confirm substate for the preceding loads could be the same as the request substate of the current load. So in substate 0, the request from load1 is sent through a memory access multiplexer whose select signal is based on the substate. Then the BasicBlockModule moves to substate 1, sends the request for load2, and waits for the confirmation of load1. When load1 is confirmed successfully, the module moves to substate 2, sends the request for load3, and waits for the confirm of load2. When load2 is confirmed, it moves to the next substate and waits for the confirm of load3 and then finishes. The store instructions are dealt with in the same way as load instructions, and in this way the memory ordering is ensured.

3.5.3 The Control Unit

ControlPathModuleGen is used to generate the control unit. The control unit is a state machine, and there are two levels of states, PC and substate. PC decides which basic block module to be enabled, and substate decides the inner execution state of a basic block. PC changes based on the terminator instruction. There are several kinds of terminator instructions, such as the conditional branch instruction, unconditional branch instruction, switch instruction, return instruction, etc. For unconditional branch instructions, the next PC is the id of the following basic block. For conditional branch instructions and switch instructions, the condition value is passed from basic blocks to the control unit to decide which basic block to be enabled next. For the return instructions, the "done" state is reached, which means the C-Core finishes execution successfully, and valid values can be read from registers V0 and V1.

3.5.4 The Tree-Structured Pipelined Multiplexer

TreeRegMuxGen is used to generate the tree-structured pipelined multiplexer for each C-Core [SAGH+11]. Each register in the GreenDroid has a unique ID, which is composed of three parts: C-Core ID, group ID and in-group register ID.

TreeRegMuxGen is different from other module generators, since it is actually a transformation pass based on A-IR, which could be also viewed as an hardware optimization to A-IR modules. TreeRegMuxGen first scans the module SPE and finds out all the registers accessible to the host processor, and then groups them. A schema is used to tell the group ID and register ID for standard registers, which are always put into group 0, and the register IDs are fixed for easy access from the host processor. Then TreeRegMuxGen generates the multiplexers, registers, wires and connections for AIR modules.

3.6 AIR to C Backend

AIRToC transforms the optimized A-IR to BTL C++. BTL C++ is a hardware simulator library used for Raw and GreenDroid.

In BTL C++, factory design pattern is used, and a C-Core is translated into a “subclass” of *spebase*. Since ports for all C-Cores are identical, the stardard ports are declared as public member variables in spebase. In a C-Core subclass, the wires are defined as private variables, and two base-class functions are overridden: calc and edge. For each simulator cycle, calc is called first to execute the data-path logic, and then the edge function is called to update register values.

Each of the other AIR modules is transformed into a function. Inputs of modules are transformed into input parameters, while outputs of modules are transformed into output parameters, which are passed by reference in C++. Then the instances of modules are transformed into function calls.

In top-level C-Core description module, all the non-register instances, wire connections are transformed into function calls in calc, and registers are in edge. The execution order of instances and wires in calc are decided by topological or-

der. Each instance or wire connection could be regarded as a node in the graph, and the edge in the graph represents the dependency between instances and wire connections. A node is dependent on another node if and only if the first node uses a wire defined by the second node. For the topological sort to be applied, there should be no loop in the graph. Basic blocks are separated by value registers, so the loop between basic blocks do not become the loop relationship between basic block instances in calc. But as to control path module, it is first used to generate the signal to enable basic blocks, and then the state signals from basic blocks are used by control path module to decide the next state, which makes control path module used twice in one calc call leading to loop dependency in the graph. To resolve the problem, we use a schema to divide a control path module into two parts, the control-enable logic and the next-state logic. The control logic calculates the control signals for C-Core, and the next-state logic reads the state of C-Core, and decides the next state. The content of the two modules are exactly the same, but they have different ports. The two new modules replaced the initial module in the graph, and in this way, the loop dependency is removed. We can further prove that, for circuits without non-deterministic states, we can always use such a way to resolve the loop dependency.

The BTL C++ function for basic block modules can be generated in a similar way as that for the top-level data path module. In basic block modules, most instances are of modules from a common hardware module library, such as adders, shifters, etc. In BTL C++, to take care of the common library modules, a library of inlined functions for all the modules is created. The functions can be called by the AIR modules. Instead of copying and pasting the module code directly, inlined functions simplify the abstraction, increase the code readability and reduce the impact on the performance.

For the control-path modules, like the control unit and multiplexers, a state machine is translated into a C++ switch statement, and a conditional block is converted into a C++ if-else statement. The ordered RTL assignments are translated into C++ assignments with the same semantics and the same order. The "unused" ports in the split control units become local variables of the function.

In this way, an A-IR C-Core module is translated into a BTL C++ module which can be used by the BTL simulator for simulation. In the simulation, BTL generates cycle-by-cycle port value information, which can be used to verify the correctness of Verilog C-Core modules.

3.7 AIR to V Backend

AIR is a hardware description IR which is quite similar to Verilog [TM02]. Each AIR module can be transformed into a Verilog module in a straightforward way. Data-path modules, including the top-level data path module and basic block modules, are interpreted as Verilog modules with instances and wire non-blocking assignments. In control path modules, state machines and conditional blocks are translated into case statements and if-else statements in Verilog RTL representation respectively and are surrounded by an always block to be executed by both “slow” and “fast” clock edges.

The generated Verilog C-Core modules are later synthesized, verified, and used to produce C-Cores in the GreenDroid architecture.

Chapter 4

Calling Convention of C-Cores

The semantics of the execution of a C-Core is equivalent to running its source function's code on the host processor. The host processor needs to pass parameters, stack pointer, the global pointer to C-Cores, and read the returned value from C-Cores. When a C-Core to be used is not available, the host processor may execute the function code locally without transferring control to the C-Core. A trampoline is used to wrap the call of a C-core. Meanwhile, a C-Core may call other functions, the code of which needs to be executed by the host processor or other C-Cores. To simplify the work done by the trampoline and make sub-function calls in C-Cores easier, C-Cores are made *compatible* to the calling conventions used by GreenDroid architecture, namely the Raw ABI, which closely resembles the MIPS O32 ABI differently only in which registers are callee vs. caller saved. The compatible here means that C-Core reads the arguments in the same way as a function called in the host processor, and functions called by a C-Core could also read the arguments in the same way, which means that both caller and callee functions of a C-Core may not be aware of the existence of the C-Core.

4.1 MIPS O32 ABI

MIPS O32 ABI (Application Binary Interface) is composed of two parts: arguments passing and stack conventions [Swe07].

Table 4.1 shows the usage of integer registers in MIPS O32 ABI, and Fig-

ure 4.1 shows the stack frame in MIPS O32 calling convention.

Both argument registers and stack can be used for argument passing. When a function is called, sp points to the bottom of the current stack frame, and should be 8-byte aligned. The slots right above sp is the space for argument passing. The first argument is in the lowest position, and each argument takes at least four bytes. Long long type takes 8 bytes, and should be 8-byte aligned. The argument space should be at least 16 bytes. The content of arguments is the same as the lowest 16 bytes in stack argument space, and sometimes the lower 16 bytes in stack argument space could be empty.

The way the returned value is stored in v0 and v1 depends on the type of the returned value. If the returned value is of basic type with width no more than 32 bits (including pointer type in 32-bit machine), for most compilers, only v0 is used. If the returned value is of type double or long long, the lower 32 bits are stored in v0, while the higher 32 bits are stored in v1. If the returned value is of type struct, the function type is transformed with a pointer to the returned value being the first argument, and the other arguments are pushed backward by one position.

Accessing global variables is not part of MIPS O32 ABI, but is also discussed here. Global variables are stored on the heap, and are addressed with global pointer gp. A read access to a global variable is a load instruction, and a write access is a store. The offsets of global variables with reference to gp are calculated at the linking time.

4.2 Argument Passing with a C-Core as a Callee

In GreenDroid, the execution of a C-Core is equivalent to a function call, and is compatible with MIPS O32 ABI. ABI compatibility maximizes the interoperability with C-cores and existing tools such as debuggers and libraries that rely upon the ABI. In C-Cores, there is a local version of register a0 to a3, sp and gp used for argument passing. When the trampoline calls a C-Core, it copies the value of these registers in the host processor to the registers in the C-Core, and the

Table 4.1: Integer Register Usage Related with MIPS O32 ABI [Swe07]

Register Number	Name	Purpose
$0	zero	Always 0
$2, $3	v0, v1	Returned value registers
$4 - $7	a0-a3	Argument registers
$28	gp	Global pointer
$29	sp	Stack pointer
$31	ra	Return address for subroutines

stack layouts are the same, and the value of sp is unchanged. When the C-Core starts, it needs to load the value of arguments from local argument registers and stacks.

CC-IR is the LLVM-formatted hardware representation, and we express the MIPS O32 ABI in CC-IR explicitly. This ensures CC-IR is closer to the hardware implementation of a C-Core. A transformation pass is added to CCIRGen to enforce the MIPS O32 ABI. In detail, a LLVM function to be made into a C-Core is transformed into an equivalent function with arg0 to arg3, sp, and gp as the arguments. The type of arg0 to arg3 is the same as the type of the first four arguments in the original function respectively if the first four arguments are not of 64-bit type. Otherwise, two arguments maybe combined to represent one long-long-type argument, or padding may be needed. The content of arg0 to arg3 is the same as the argument registers a0 to a3. Sp and gp are byte-pointers, which stores the stack pointer and global pointer respectively. To ensure the same semantics, a new basic block is added at the beginning of the function to load from the stack the value of arguments in the original function.

In this way, a CC-IR function conforms to the MIPS O32 ABI by construction, which will be directly mapped to an A-IR hardware module in a direct way. Optimizations related to memory accesses could also be applied to the CC-IR function.

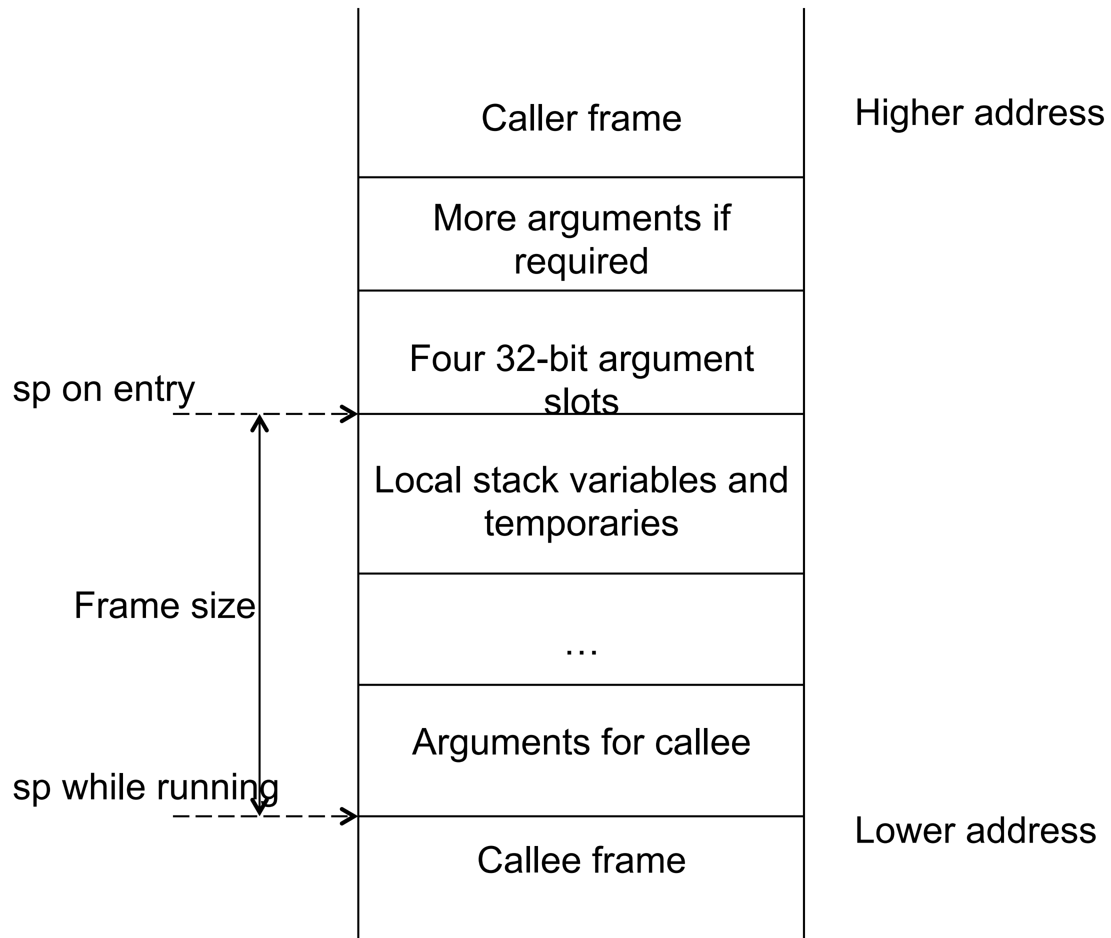


Figure 4.1: MIPS O32 Stack Layout [Swe07]

4.3 Stack Frame of C-Cores as Leaf Functions

In order to be compatible with MIPS O32 calling convention, the stack layout in GreenDroid is similar to that in MIPS O32, shown in Figure 4.2. One difference is that there is no stack space for temporaries or local variables except those assigned by an `alloc` instruction, because temporaries and local variables typically turn into physical registers and wires in C-Cores. An `alloc` instruction allocates a static space on the stack and assigns the base address of the space to the destination variable.

A special case is that the address of a local variable is passed to the sub-function as an argument. In this case, since all the local variables are stored in the

local registers, and there is no space on the stack for them, the subroutines cannot access the variable with the address. To solve the problem, an alloc instruction is used to create a space for the variable in the stack, and the pointer to the space is passed to the subroutine.

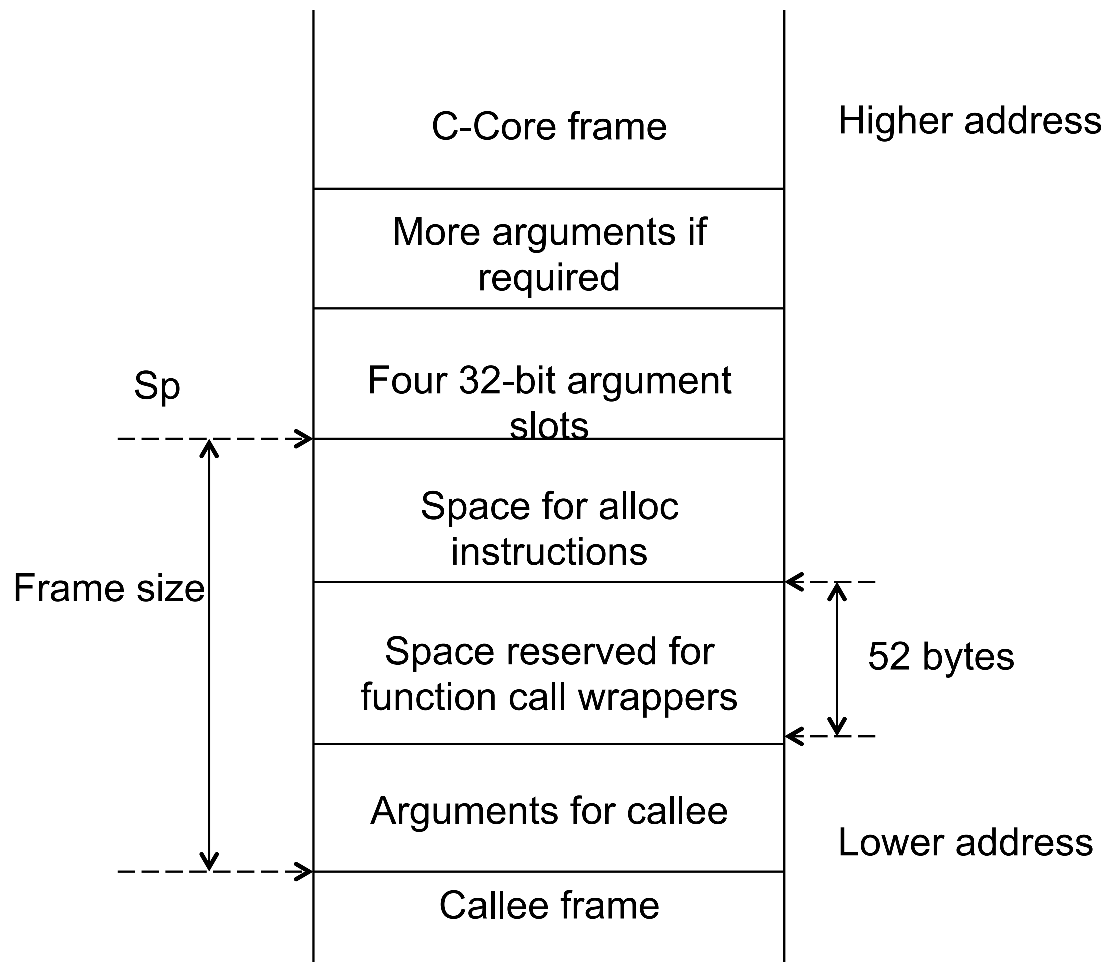


Figure 4.2: Stack Frame of C-Cores as Leaf Functions

4.4 Sub-function Calls in C-Cores

Functions to be converted into C-Cores may also call other functions, in which case the C-Cores are not leaf functions, and parameters for the sub-functions need to be prepared by the C-Cores.

When a C-Core needs to call a sub-function, it prepares the arguments in the stack, sends an attention signal to the host processor, and waits in an exceptional state. The host processor receives the attention signal, and executes a wrapper function that reads the state of the C-Core and decides which exception-handling function to be actually executed. Then it copies arguments from stack to argument registers (a0 to a3 in the host processor) conforming to MIPS O32 ABI, executes the function, copies the returned value from v0 and v1 back to the registers of C-Core, and in the end resumes execution of the C-Core.

Two levels of wrapper functions are used to dispatch and execute the function, extra space in the stack is reserved for the argument space of wrapper function call, as is depicted in Figure 4.3. Since arguments in the stack should be adjacent to the callee frame, the reserved space for the wrappers is above the argument space of the callee.

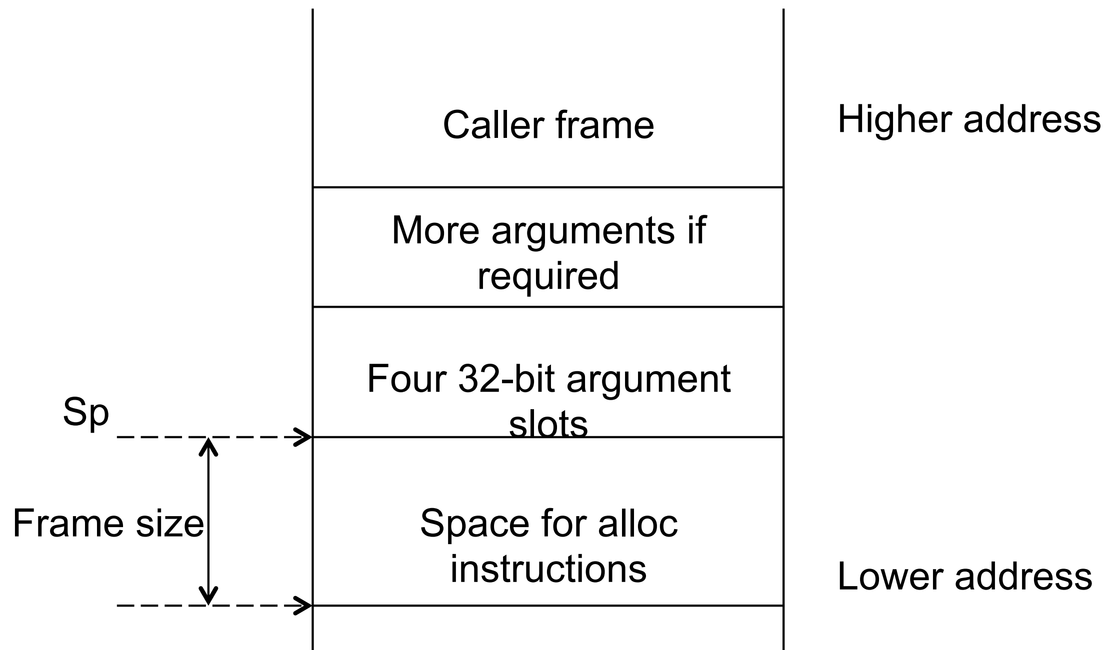


Figure 4.3: Stack Frame for C-Cores as Non-Leaf Functions

4.5 Global Variables

In LLVM IR, global variables are accessed by the pointers. When the value of a global variable is read, a load instruction is used to get the value referenced by the pointer, and when the value is changed, a store instruction is issued.

When a function is called, the global pointer gp is passed in, and the offset to each global variable is calculated in advance at the linking time. In this way, the function can access each global variable.

In order for the C-Core to access global variables, a similar method is used. In a C-Core, for each global variable the C-Core accesses, a register called global variable offset register is created. Before the C-Core is executed, the offset of each global variable is written to the global variable offset register in the C-Core. During the execution, the pointer to each global variable is calculated by adding GP and the offset of the global variable, and the pointer is then used for memory accessing.

The offsets for global variables are set during the linking stage. So an easy way to initialize the global variable offset registers is to read the offsets at the start of a program, and then write the correct offset values to registers, which is done at the start of the main function.

One problem os this approach is that static global variables are only accessible within the file scope. Therefore if static global variables not in the same file as the main function, the offset cannot be obtained. The solution is to append an offset reading function for each global variable to the file where the global variable is defined. In the main function, the offset reading functions are called to get the offsets of static global variables. For static variables within a function scope or a class scope, similar methods can be used, but have not been implemented yet.

Another reason to use global variable offset registers is for patching. When the order of global variables changes, or new global variables are added, the offset of each global variable may change. With global variable offset registers, we can support the change of global variable offsets without generating new C-Cores.

Chapter 5

A C-Core for the Android Dalvik Garbage Collector

The GreenDroid architecture is designed to be used in mobile phones to increase energy-efficiency and exploit dark silicon. The Android operating system [Dev11] is one of the current mainstream mobile operating systems, and an energy-consumption hot-spot in the Android system is the Dalvik [Bor08] garbage collector [GSV^{+}10].

In this chapter, I describe a C-Core generated from the Android Dalvik Garbage Collector as a case study. First, I make a brief introduction of the Dalvik Java virtual machine and the garbage collector in it. Then I explain the mechanisms used in generating and testing a C-Core from the garbage collector. Finally, I provide the software and hardware features of the generated C-Core.

5.1 Introduction to Dalvik and Dalvik GC

Dalvik is the Java virtual machine (VM) used in the Android operating system [Bor08] [Ehr10]. Due to the resource constraints of mobile phones, Dalvik is designed to run on a slow CPU, use less memory and consume lower power than JVM. In Dalvik, each application runs on a virtual machine in a separate process, and has an independent heap. Garbage Collection (GC) for each application is done separately.

There are four kinds of memory: *shared clean*, *private clean*, *shared dirty*, and *private dirty*. Shared means the memory is used by many processes, and private means the memory is used by one process. A clean memory space is `mmap()`ed from dex files and unwritten, while a dirty memory is created in the heap through `malloc()`. GCs are done independently for dirty memory spaces and should respect shared memory spaces.

The mark-sweep algorithm [BW88] is used in Dalvik for GC [Cha09]. In the *mark* step, objects in heaps are scanned recursively, and reachable objects are marked. Then in the *sweep* step, the unreachable objects are swept and deallocated. Before the mark step, all the other threads in the current process are paused, and resumed after mark is finished. Unmarked memory is not used any more, so sweep could be done in parallel while the application resumes execution.

From the profiling result in [SVGH+11a], the coverage of `scanObject` is 3.6% of the Android execution time, ranking second among all the functions. `scanObject` is used in the mark step of the GC. Thus a C-Core will be made from `scanObject` to make the GreenDroid architecture run Android more energy-efficiently.

5.2 Generation and Test Mechanism of the C-Core

To generate a C-Core from `scanObject` function, only the function source code is required. To test the C-Core, the C-Core needs to be run by the Android operating system, or Dalvik Java virtual machine. Since BTL is a cycle-by-cycle simulator, it is too slow to run either android operating system or Dalvik Java virtual machine. At the same time, the Dalvik Java virtual machine is deeply coupled with the Android operating system, and could not be readily isolated and ported.

Thus, to test the C-Core, we need to isolate the garbage collector from Dalvik first. Then an object and object dependency initialization function is called before `scanObject`, and a result check function is invoked after the execution of

scanObject. To test performance, QEMU [Bel07] will be modifed to dump the snapshot of object and object dependency when GC is called each time, and the snapshot is used by the initialization function. Because the mark step is executed in single-thread mode, writing a test to test the mark step separately is reasonable.

5.3 The Generated C-Core

Table 5.1 shows the statistics for the CC-IR function for scanObject. The optimization level of the LLVM frontend Clang is O3. From Table 5.1, it is noticed that there are totally 87 basic blocks, and memory instructions account for 50.11% of the total instructions.

Table 5.2 is a list of static instruction counts in the CC-IR function for scanObject by instruction type. The top five instruction types take up 78.17% of the total static instructions. GetElementPtr instructions are used for memory access addressing, which are implemented as combination of constant-offset shifters and adders in hardware. Br are branch instructions, including conditional and unconditional ones. The larger the number of branch instructions, the more complex the control path module will be. BitCast instructions are no-op casting from one type to another, which is basically a wire connection in hardware. Load and Store are memory access instructions. Memory instructions include Store instructions, Load instructions, GetElementPtr instructions, Call instructions, Invoke instructions and Alloca instructions. Table 5.1 and Table 5.2 both show that scanObject is a memory-intensive application.

Table 5.1: Statistics of scanObject CC-IR

Name	Count
Basic blocks	87
Instructions (of all types)	449
Memory instructions	225
Non-external functions	1

Table 5.2: The Instruction Count of `scanObject` CC-IR

Instruction Name	Count	Percentage
GetElementPtr	112	24.94%
Br	83	18.49%
BitCast	60	13.36%
Load	49	10.91%
Store	47	10.47%
ICmp	33	7.35%
Call	17	3.79%
PHI	17	3.79%
And	9	2.00%
Add	6	1.34%
LShr	6	1.34%
Sub	3	0.67%
Switch	3	0.67%
PtrToInst	1	0.22%
Ret	1	0.22%
Shl	1	0.22%
Xor	1	0.22%
Total	449	1

For synthesis a TSMC 45-nm GS process is targeted using Synopsys Design Compiler (C-2009.06-SP2) and IC Compiler (C-2009.06-SP2). The generated synthesizable Verilog is processed automatically in the Synopsys CAD tool flow, starting with netlist generation and continuing through placement, clock synthesis, and routing, and finally performing post-route optimization.

As is shown in Table 5.3, `scanObject` takes $0.0556mm^2$ in total. v

Table 5.4 shows the top five critical paths in `scanObject` C-Core. The longest critical path is from the live-out register load_23_1_lo_reg, to the `mem_addr` output of the C-Core. The output of the start-point register is used in memory

Table 5.3: `scanObject` C-Core Area

Port Count	213
Net Count	25671
Cell Count	21436
Combinational Cell Count	16678
Sequential Cell Count	4557
buf/inv Count	3423
Reference Count	596
Combinational area	$0.0384mm^2$
Noncombinational area	$0.0173mm^2$
Total area	$0.0556mm^2$

address calculation, and then routed to `mem_addr` through a multiplexer. The next several longest critical paths all start from some bit of `pc_reg` to some bit of `mem_store_value`. `pc_reg` is used by the control path module to decide the value of `enable` and `select` signals. One of the `select` signals is used by the multiplexer for memory to-be-stored value, the output of which is connected to `mem_store_value`. The critical path information tells that memory addressing logic and the state machines in the control unit take the longest time, which may be further optimized to improve the performance of C-Cores.

Table 5.4: `scanObject` C-Core Critical Paths

Start Point	End Point	Time/ns	Slack/ns
load_23_1_lo_reg/ state_reg[11]	mem_addr[31]	0.6459	-0.1469
pc_reg/ state_reg[3]	mem_store_value [19]	0.6324	-0.1334
pc_reg/ state_reg[3]	mem_store_value [25]	0.6323	-0.1333
pc_reg/ state_reg[3]	mem_store_value [27]	0.6317	-0.1327
pc_reg/ state_reg[3]	mem_store_value [23]	0.6313	-0.1323

Chapter 6

Related Work

In this chapter, related works on dark silicon, synthesis tools from high-level languages to silicon, and specialized hardware for garbage collection, are provided and discussed.

6.1 Dark Silicon

With the breakdown of Dennard scaling [DGR+74], the percentage of active silicons used simultaneously in a chip drops exponentially with each process generation. Several works analyze the phenomenon of *dark silicon* [Tay12] [GHSV+11] [GSV+10] [VSG+10] [EBA+11].

In [VSG+10] [GSV+10] [GHSV+11] [Tay12], the utilization wall is proposed as the cause of dark silicon and the derivation of utilization wall is discussed in detail. The number of transistors in chips with same areas continues to scale by 2x and the switching speed of a transistor scales by 1.4x every two years as predicted by Moore's Law. The transistor capacitance reduces 1.4x continually, but the threshold voltage fails to reduce in the post-Dennardian scaling regime. Thus the energy efficiency is only improved by 1.4x every two years, which requires a shortfall of 2x for the percentage of active transistors, which leads to the utilization wall. The transistors that must stay inative to meet the power constraint are called dark silicon. [EBA+11] also analyzes dark silicon and the end of multicore scaling based on scaling models, performance models and empirical results to point out

the coming regime of dark silicon.

To attack the utilization wall and exploit dark silicon, chip shrinking, dim silicon, hardware specialization and fundamental breakthrough in semiconductor devices may be potential directions [Tay12]. The GreenDroid architecture employs the specialized hardware to use dark silicon in the mobile domain [GHSV+11] [VSG+10] [GSV+10], and [HFFA11] analyzes the potential of the mechanism of hardware specialization in the server domain.

6.2 High-level Language to Silicon

There are several other frameworks which build accelerators directly from high-level language source code, such as AutoPilot from AutoESL [ZFJ+08], Impulse C [C10], Synopsis Synphony/PICO [SAM+02], SUIF [BRM+99] etc. In this section, I discuss the difference of the tools and ours.

AutoPilot [ZFJ+08] is a commercial Electronic system-level (ESL) synthesis platform. Instead of traditional HDL, a subset of C or C++ is used to describe the hardware, which is called synthesizable C, or C++. A tool chain is used to compile the C, C++ description, apply some optimizations, and synthesize the code to RTL VHDL or Verilog. Since in AutoPilot, C/C++ is employed to replace traditional RTL as hardware description, but only a limited number of C/C++ features are supported, and the usage of dynamic pointers, dynamic allocations, and function recursions are disallowed. In comparison, our tool chain is targeted supporting arbitrary C/C++ functions, and is designed to support all the C/C++ features, which allows C/C++ functions in existing applications to be synthesized directly without modifications of the source code.

Synopsis Synphony/PICO [SAM+02] automatically synthesizes accelerators for loop nests in C. Several compiler loop optimizations, and scheduling mechanisms are used to employ the parallelism of loops, and the compiler requires that the loop nests are *perfect*. Perfect means that all the statements in the loop nests except the `for` statements should be in the innermost loop. For example, the below nested loop provided in [SAM+02] is not a perfect nest, because there is an initial-

ization statement in the outer loop. Users need to manually move the initialization code to another loop to make it a perfect nest. Such a strong constraint limits the coverage of code that could be automatically transformed to accelerators. In contrast, in our tool chain, irregular code is also supported and the parallelism can still be used through the mechanism of selective depipelining [SAGH+11].

```
for (j1 = 0; j1 < 8192; j1++) {
    y[j1] = 0;
    for (j2 = 0; j2 < 16; j2++)
        y[j1] = y[j1] + w[j2] * x[j1 + j2];
}
```

CHiMPS [PEB+09] is a Field Programmable Gate Array (FPGA) compiler for manycache, an application-specific FPGA-based memory architecture. CHiMPS reads the C-Code, and transforms it into CHiMPS Target Language (CTL), and then generate the VHDL for the memory architecture based on the CTL. Compared with the compiler tool chain for GreenDroid, CHiMPS targets at the memory architecture and FPGA, while our compiler tool chain generates the hardware code for computing accelerators, which can be either FPGA or ASIC. CTL in CHiMPS is a dataflow intermediate language used to analyze memory access pattern. The CC-IR and A-IR in our tool chain are both used to describe the hardware, while CC-IR is the behavioral specification, and A-IR is a combination of gate level and register transfer level (RTL) description.

Altera C2H[LPM06] is to generate stand-alone hardware modules from C/C++ functions. Similar to our tool chain, Altera C2H also supports the feature of specifying a function in an application code, and transforming it automatically to a hardware module. The difference is that our compiler tool chain targets at normal mobile applications, while Altera C2H is basically a platform or software IDE to allow developers to use C for both software and hardware development. Meanwhile, Altera C2H generate accelerators for speed-up, and the accelerators from our tool chain are for energy-efficiency.

Impulse C [C10] is a subset of ANSI C for FPGA programming. Impulse C tool chain optimizes C code for parallelism, generates HDL files ready for FPGA

synthesis, and also generates hardware/software interface. Similar to CHiMPS, Impulse C is another tool chain targeting for FPGA. Moreover, applications need to be written specifically complying with the subset of ANSI C supported by impulse C, and needs to use a function library provided by impulse C to improve parallelism with FPGA.

Spatial computation uses a compiler tool chain similar to ours [BVCG04] [MCC+06], but the generated co-processors use asynchronous logic, and are optimized for performance. The mechanism to deal with PHI node is also different from that in C-Cores.

6.3 Specialized Hardware for Garbage Collection

Several previous works that use hardware to help accelerate garbage collectors are discussed in this section.

In [Moo84], a small piece of special hardware (a barrier) is used to help with the garbage collection in Lips system. [SN94] introduces a special memory architecture with garbage-collected memory modules (GCMM) to help with garbage collection in C++ programs. GCMM has a local processor to run the garbage collection algorithms. The GCMM is connected with CPU and memory through buses. Both [SaLC03] and [GS05] are designed for Java in embedded systems. [SaLC03] explains a memory processor used as a hardware garbage collector with reference counting and a mark-sweep garbage collection algorithm. [GS05] describes a concurrent garbage collector unit (GCU) for the Java Optimised Processor (JOP) [Sch08], and the GCU implements a mark-compact algorithm with concurrency support. All of the hardware coprocessors are designed and implemented manually for a targeted platform with specific garbage collection algorithms to increase performance. In comparison, the Arsenal tool chain automatically produces a garbage collector C-Core based on the Dalvik GC source code, and is used to improve energy-efficiency.

Chapter 7

Conclusion

In the current regime, the utilization wall has prevented processor speed from increasing, and heterogeneous architecture is an effective way to exploit dark silicon. GreenDroid, a heterogeneous architecture optimized for energy efficiency, has been proposed as a solution to attack utilization wall in the mobile domain. The Arsenal tool chain is used to automatically generate conservation cores (C-Cores) for the GreenDroid architecture. In this thesis, I describe the design and implementation details of the Arsenal tool chain. Then I discuss in detais the calling convention of C-Cores to show how dynamic allocation, dynamic memory access, sub-function call, and global variables are supported in C-Cores. At last, I examine a C-Core generated from the Android Dalvik garbage collector as a case study.

The Arsenal tool chain is based on the LLVM compiler infrastructure. The tool chain itself provides a mechanism to synthesize hardware behavioral description in C/C++ to a combination of RTL and gate-level representation in Verilog and BTL C++. Two IRs are designed and used in the tool chain, CC-IR and A-IR. CC-IR, a subset of LLVM IR, is proposed as an hardware behavioral description IR. A-IR, a hardware RTL and gate-level description IR, is introduced for hardware optimization and to reduce the effort in hardware backend design. The CC-IR to A-IR is an LLVM backend which bridges the hardware behavioral description and lower-level description.

As to the future work, the impact of software optimization, such as loop un-

rolling and function inlining, on hardware performance, area and energy-efficiency needs to be explored. The support for C++ features, such as exceptions, class inheritance, and exception, needs to be further studied and discussed. An effective mechanism to handle direct or indirect recursion of C-Cores need to be designed and developed. Moreover, the experiments on what percent of Android application code can be supported by the Arsenal tool chain, and the energy efficiency improved by the GreenDroid architecture on the whole Android system require to be conducted. At last, the tool chain support for other frontend languages than C/C++ is to be explored.

Finally, the work presented in the thesis provides a LLVM-based tool chain for the GreenDroid architecture, which is more reliable and maintainable, and easier to apply hardware optimizations. The Arsenal tool chain has supported most of the benchmarks supported by the old-version tool chain, and a chip using the GreenDroid architecture and including two C-Cores from the Android source code produced from the Arsenal tool chain is scheduled to be taped out. It is my hope that the Arsenal tool chain could continuously contribute to the research and development in the GreenDroid project.

Appendix A

Format of A-IR Modules

Table A.1: Fields of A Data-Path Module

Field Name	Description
Name	The module name
Ports	The description of ports
Wires	The description of wires
Connections	The connection between wires
Instances	The instances of the modules

Table A.2: Fields of A Control-Path Module

Field Name	Description
Name	The module name
Ports	The description of ports
Wires	The description of wires
Assignments	The RTL-assignments
StateMachine	One state machine block
ConditionBlock	One conditional block

Table A.3: Fields of A Instance

Field Name	Description
Name	The instance name
Module	The name of the module of the instance
Ports	The mapping between ports and wires

Table A.4: Fields of An Assignment

Field Name	Description
SVar	The left value of an assignment
Expr	The expression

Appendix B

Standard Ports and Registers of an SPE

Table B.1: The Default Inputs for SPE

Name	Description
clk	clock
mem_load_value	loaded value from memory
mem_tag_in	tag from memory
mem_valid	valid signal from memroy
reset	reset signal
tree_addr	tree access address
tree_re	read enable signal for tree access
tree_store_value	to be stored value for tree access
tree_we	write enable signal for tree access

Table B.2: The Default Outputs for SPE

Name	Description
attention	attention signal the host processor
err_flag_real	error flag for the host processor
mem_access_type	memory access type, 2 for load, 1 for store, 0 for none
mem_addr	memory access address
mem_store_mode	alignment for store instruction, 0 for 8-bit, 1 for 16-bit, and 2 for 32-bit
mem_store_value	value to be stored in the memory
mem_tag_out	memory access to memory
tree_load_value	loaded value from tree reg mux

Table B.3: The Default Registers for SPE

Name	Description
spe_status	The register storing the spe current execution status. 1 : Ready; 2: Off / Done; 4: Programming; 8: Waiting for exception to be handled
pc	Storing the id of the currently executing block
prev_pc	Storing the id of the predecessor block
substate	Storing the substate in the current block
prev_pc	Storing the last substate
edge_id	Storing the edge id. The higher 16 bits equal to pc, and the lower 16 bits equal to prev_pc.
ret_code	Storing the return code. 0 for running, 1 for exception handling, and 2 for normal finish.
exceptionCause	Not used currently.
arg0	first argument
arg1	second argument
arg2	third argument
arg3	fourth argument
argSP	Stack pointer
argGP	Global variable base address
argV0	Return value register V0
argV1	Return value register V1
argRA	Return address

Bibliography

[AAB+97] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, D. Srikrishna, and M. Taylor. The raw compiler project. *Proceedings of the Second SUIF Compiler Workshop*, pages 21–23, 1997.

[Bel07] Fabrice Bellard. Qemu open source processor emulator. *URL: http://www. qemu. org*, 2007.

[BGRT05] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86a platform for analyzing x86 executables. In *Compiler Construction*, pages 139–139. Springer, 2005.

[Bor08] Dan Bornstein. Dalvik vm internals. In *Google I/O Developer Conference*, volume 23, pages 17–30, 2008.

[BRM+99] Jonathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank, Rajeev Barua, and Saman Amarasinghe. Parallelizing applications into silicon. In *Field-Programmable Custom Computing Machines, 1999. FCCM'99. Proceedings. Seventh Annual IEEE Symposium on*, pages 70–80. IEEE, 1999.

[BTLAA03] M. Bedford Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar operand networks: On-chip interconnect for ilp in partitioned architectures. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 341–353. IEEE, 2003.

[BVCG04] Mihai Budiu, Girish Venkataramani, Tiberiu Chelcea, and Seth Copen Goldstein. Spatial computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, 1988.

[C10] Impulse C. Impulse accelerated technologies. *Inc., http://www.impulsec.com*, 2010.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[Cha09] Brett Chabot. vm/alloc - platform/dalvik - git at google. https://android.googlesource.com/platform/dalvik/+/android-2.0_r1/vm/alloc/, 2009.

[Dev11] Android Developers. What is android? *http://developer.android.com/guide/basics/what-is-android.html*, 2, 2011.

[DGR+74] Robert H Dennard, Fritz H Gaensslen, VL Rideout, E Bassous, and AR LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[EBA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[Ehr10] David Ehringer. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 2010.

[GHSV+11] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The greendroid mobile application processor: An architecture for silicon's dark future. *Micro, IEEE*, pages 86–95, March 2011.

[GHSZ+12] Nathan Goulding-Hotta, Jack Sampson, Qiaoshi Zheng, Vikram Bhatt, Steven Swanson, and Michael Taylor. Greendroid: An architecture for the dark silicon age. In *Asia and South Pacific Design Automation Conference*, 2012.

[GS05] Flavius Gruian and Zoran Salcic. Designing a concurrent hardware garbage collector for small embedded systems. *Advances in Computer Systems Architecture*, pages 281–294, 2005.

[GSV+10] Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Jonathan Babb, Michael Taylor, and Steven Swanson. GreenDroid: A Mobile Application Processor for a Future of Dark Silicon. In *HOTCHIPS*, 2010.

[HFFA11] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *Micro, IEEE*, 31(4):6–15, 2011.

[Kid07] Robert E Kidd. *The OpenIMPACT Whole Program Optimization Framework*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.

[KTMW03] J.S. Kim, M.B. Taylor, J. Miller, and D. Wentzlaff. Energy characterization of a tiled architecture processor with on-chip networks. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 424–427. ACM, 2003.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[LA06] Chris Lattner and Vikram Adve. Llvm language reference manual, 2006.

[Lat06] Chris Lattner. Introduction to the llvm compiler infrastructure. In *Itanium Conference and Expo*, 2006.

[Lat08] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference, Ottawa, Canada*, 2008.

[LPM06] David Lau, Orion Pritchard, and Philippe Molson. Automated generation of hardware accelerators with direct memory access from ansi/iso standard c functions. In *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*, pages 45–56. IEEE, 2006.

[MCC+06] Mahim Mishra, Timothy J Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C Goldstein, and Mihai Budiu. Tartan: evaluating spatial computation for whole program execution. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 163–174. ACM, 2006.

[Moo84] David A Moon. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 235–246. ACM, 1984.

[PEB+09] Andrew Putnam, Susan Eggers, Dave Bennett, Eric Dellinger, Jeff Mason, Henry Styles, Prasanna Sundararajan, and Ralph Wittig. Performance and power of cache-based reconfigurable computing. *ACM SIGARCH Computer Architecture News*, 37(3):395–405, 2009.

[SAGH+11] Jack Sampson, Manish Arora, Nathan Goulding-Hotta, Ganesh Venkatesh, Jonathan Babb, Vikram Bhatt, Steven Swanson, and Michael Bedford Taylor. An evaluation of selective depipelining for fpga-based energy-reducing irregular code coprocessors. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 24–29. IEEE, 2011.

[SaLC03] Witawas Srisa-an, C-TD Lo, and J-M Chang. Active memory processor: A hardware garbage collector for real-time java embedded devices. *Mobile Computing, IEEE Transactions on*, 2(2):89–101, 2003.

[Sam] John Morgan Sampson. *Design and Architecture of Automatically-generated Energy-reducing Coprocessors*. PhD thesis.

[SAM+02] Robert Schreiber, Shail Aditya, Scott Mahlke, Vinod Kathail, B Ramakrishna Rau, Darren Cronquist, and Mukund Sivaraman. Pico-npa: High-level synthesis of nonprogrammable hardware accelerators. *The Journal of VLSI Signal Processing*, 31(2):127–142, 2002.

[Sch08] Martin Schoeberl. *Jop: A java optimized processor for embedded real-time systems*. VDM Publishing, 2008.

[SMSO03] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 291. IEEE Computer Society, 2003.

[SN94] William J Schmidt and Kelvin D Nilsen. Performance of a hardware-assisted real-time garbage collector. In *ACM SIGPLAN Notices*, volume 29, pages 76–85. ACM, 1994.

[ST11] Steven Swanson and Michael Taylor. GreenDroid: Exploring the next evolution for smartphone application processors. In *IEEE Communications Magazine*, March 2011.

[SVGH+11a] Jack Sampson, Ganesh Venkatesh, Nathan Goulding-Hotta, Saturnino Garcia, Manish Arora, Siddhartha Nath, Vikram Bhatt, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Energy-saving coprocessors for nasty real world code. In *Languages, Compilers, Tools and Theory for Embedded Systems*, April 2011.

[SVGH+11b] Jack Sampson, Ganesh Venkatesh, Nathan Goulding-Hotta, Saturnino Garcia, Steven Swanson, and Michael Bedford Taylor. Efficient Complex Operators for Irregular Codes. In *HPCA 2011: High Performance Computing Architecture*, 2011.

[Swe07] Dominic Sweetman. *See MIPS run*. Morgan Kaufmann, 2007.

[Syn13] Synopsys. Synopsys vcs. *http://www.synopsys.com/VCS*, 2013.

[Tay99] M.B. Taylor. *Design decisions in the implementation of a Raw architecture workstation*. PhD thesis, Citeseer, 1999.

[Tay04] M.B. Taylor. The raw processor specification. *Comprehensive specification for the Raw processor*, 2004.

[Tay12] Michael B. Taylor. Is dark silicon useful? harnessing the four horesemen of the coming dark silicon apocalypse. In *Design Automation Conference*, 2012.

[Tei00] Tim Teitelbaum. Codesurfer. *ACM SIGSOFT Software Engineering Notes*, 25(1):99, 2000.

[TJ03] Michael Taylor and Paul Johnson. Raw resources. *http://groups.csail.mit.edu/cag/raw/raw_intro_day_web/RawMap.html*, 2003.

[TKM+02] Michael B. Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. In *IEEE Micro*, March 2002.

[TKM+03] M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, W. Lee, A. Saraf, et al. A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network. In *Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International*, pages 170–171. IEEE, 2003.

[TLW+89] Donald E Thomas, Elizabeth D Lagnese, Robert A Walker, Jayanth V Rajan, Robert L Blackburn, and John A Nestor. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*, volume 85. Springer, 1989.

[TM02] Donald E Thomas and Philip R Moorby. *The Verilog® Hardware Description Language*, volume 2. Springer, 2002.

[TPS+04] Michael Bedford Taylor, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, Anant Agarwal, Walter Lee, Jason Miller, et al. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 2–13. IEEE, 2004.

[VSG+10] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *ASPLOS 2010: Architectural Support for Programming Languages and Operating Systems*, 2010.

[VSG+11] Ganesh Venkatesh, John Sampson, Nathan Goulding, Sravanthi Kota Venkata, Michael Bedford Taylor, and Steven Swanson. Qscores: Configurable co-processors to trade dark silicon for energy efficiency in a scalable manner. In *Proceedings of The 44th International Symposium on Microarchitecture*, 2011.

[WTS+97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, et al. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[ZFJ+08] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. Autopilot: A platform-based esl synthesis system. *High-Level Synthesis: From Algorithm to Digital Circuit*, pages 99–112, 2008.