A BINARY INSTRUMENTATION TOOL FOR THE BLACKFIN PROCESSOR

A Thesis Presented

by

Enqiang Sun

to
The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Master of Science

in
Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

August 2009
Abstract

Program analysis and simulation tools have been demonstrated to be valuable in the hardware or software analysis of general purpose microprocessors. Techniques like profiling and instrumentation enable software and hardware designers to make decisions on various design trade-offs in a timely and cost-effective manner. While a large number of program profiling and instrumentation tools have been developed to support hardware and software analysis on general purpose systems, there is a general lack of sophisticated tools available for embedded architectures. Embedded systems are typically sensitive to performance bottlenecks, memory leaks and other software inefficiencies. There is a growing need for better tools in this rapidly growing design space.

In this thesis we describe DSPInst, a binary instrumentation tool for the Analog Device’s Blackfin family of Digital Signal Processors (DSPs). DSPInst provides fine-grained control over the execution of programs. Instrumentation tool users are able to gain transparent access to the processor and memory state before or after every executed instruction, without perturbing the architected program state.

DSPInst provides a platform for building a wide range of customized analysis tools at the instruction level granularity. To demonstrate the utility of this toolset,
we present three example analysis tools: 1) a utility to count the number of 64-bit instruction executed by the DSP, 2) a profiling tool to collect information for loop execution, and 3) the use of voltage and frequency scaling on any instruction boundary.
Acknowledgements

First of all, I’d like to thank my advisors, Dr. David Kaeli and Dr. Xinping Zhu, for all their guidance and support. I’d also like to thank my parents and Liwei, who gave me encouragement and support all the way.

I’d also like to thank the people in Analog Devices, especially in Platform Tools Group, for their support and invaluable advices.
# Contents

Abstract ............................................... iv

Acknowledgements ........................................ vi

1 Introduction ........................................... 1
  1.1 Motivation ........................................... 1
  1.2 Contributions of Thesis .............................. 3
  1.3 Organization of Thesis .............................. 4

2 Background ............................................ 6
  2.1 Embedded Systems ................................. 6
  2.2 Instrumentation ..................................... 8
  2.3 Related Work ........................................ 10
    2.3.1 Hardware Simulation and Emulation .......... 10
    2.3.2 Binary Optimization ............................ 11
    2.3.3 Static Binary Instrumentation ............... 12
    2.3.4 Dynamic Binary Instrumentation ............. 14
    2.3.5 Instrumentation on Embedded System ....... 15
    2.3.6 Binary Translation ............................ 15
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The instrumentation process</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>An instrumented application</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Executable and Linking Format</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>Padding the FORCE INTERRUPT instruction</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>Instrumenting PC related instruction</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>Instrumenting the JUMP instruction</td>
<td>26</td>
</tr>
<tr>
<td>3.7</td>
<td>Instrumenting the CALL instruction</td>
<td>27</td>
</tr>
<tr>
<td>3.8</td>
<td>Instrumenting the Zero-overhead Loop Setup instruction</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Product image of ADSP-BF548 EZ-KIT Lite Evaluation System</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>An instruction-counting analysis function</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>The instrumentation overhead counting 64-bit instructions</td>
<td>39</td>
</tr>
<tr>
<td>5.3</td>
<td>Design flow of loop analysis function</td>
<td>41</td>
</tr>
<tr>
<td>5.4</td>
<td>Application with voltage and frequency scaling</td>
<td>44</td>
</tr>
<tr>
<td>5.5</td>
<td>Execution Time and Energy Estimation with or without VFS</td>
<td>45</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Parallel Issue Combinations ................................. 23

4.1 Subset of EEMBC benchmark applications ...................... 36

5.1 Counting the occurrence of 64-bit instructions in the EEMBC benchmark. 40
5.2 Profiling information for loop execution in HUFFDE ............... 42
5.3 Configurations with and without voltage/frequency scaling .......... 43
Chapter 1

Introduction

1.1 Motivation

Instrumentation tools have been shown to be extremely useful in analyzing program behavior on general purpose systems. Software developers have used them to gather program information and identify critical sections of code. Hardware designers use them to facilitate their evaluation of future designs. Instrumentation tools can be divided into two categories, based on when instrumentation is applied. Instrumentation applied at run-time is called \textit{dynamic instrumentation}; instrumentation applied at compile time or link time is called \textit{static instrumentation}.

Instrumentation techniques in the embedded domain are relatively immature, though due to the increased sophistication of recent embedded systems, and the need for more powerful tools is growing. It is critical for embedded system designers to be able to properly debug hardware and software issues. The focus on improving instrumentation frameworks for digital signal processing systems has increased \cite{43, 21}.

The goal of this thesis is to create a fine-grained code instrumentation tool in
an embedded environment. We would like a comprehensive tool that transparently creates interfaces between every executed instruction. Custom tools can then be integrated through these interfaces. In order for this tool to be maximally usable, it should be:

- **Transparent**
  
The tool should not inadvertently alter the behavior of any application program. Transparency is important especially when targeting applications in actual use. Unintended changes in architected state may have severe consequences and even crash the whole application. Any behavior changes in the native application would obsolete the profiling information, which depends on the execution correctness of the native application.

- **Comprehensive**
  
The tool must be able to observe and modify any and all executed instructions, so that maximal instrumentation flexibility is guaranteed.

- **Practical**
  
  To be useful, the tool must work on existing, relevant, unmodified commodity embedded platforms.

- **Universal**
  
The tool should be capable of working on every application, not only including hand-crafted assembly programs, but also large, complex, commercial programs.

- **Customizable**
  
The tool should supply a platform for construction of custom analysis tools.
The tool should constrain its overhead to avoid excessive slowdowns. Near-native performance is required especially for real-time embedded platforms.

These goals define the characteristics of our instrumentation tool. Some of them are complementary: transparent and universal work together making the tool work on as many applications as possible. Others conflict, such as being comprehensive and practical at the same time as maintaining efficiency. This thesis is about optimally realizing the combination of these goals.

1.2 Contributions of Thesis

In our work, we describe DSPInst, a binary instrumentation tool targeting Analog Devices’ Blackfin family of DSPs. DSPInst provides an infrastructure wherein embedded system developers can design a wide range of customized analysis tools that operate at an instruction granularity. In DSPInst we have adopted a static instrumentation approach, and modeled our tool after ATOM [48], a link-time instrumentation toolkit developed for the DEC Alpha architecture.

DSPInst utilizes static instrumentation. The main contributions of DSPInst are:

- This is the first tool-building system targeting the Blackfin DSP processors. Analysis tools to perform code profiling or code modification can be easily built.
- DSPInst allows for selective instrumentation; the user can specify on instruction boundaries when to turn profiling or code modifications on/off.
• DSPInst instruments object code, versus source or assembly. DSPInst decouples the user from having to provide source code of the application to be instrumented.

The basic philosophy of DSPInst is to replace instructions in the original binary with trampoline code that jumps to analysis procedures. Despite the presence of variable-length instructions in the Blackfin ISA, our tool is able to achieve transparency for all the instructions.

1.3 Organization of Thesis

The remainder of the thesis is organized as follows:

Chapter 2 provides background information about embedded systems and instrumentation techniques. Related research work on binary instrumentation and related techniques are surveyed.

Chapter 3 presents the design methodology that has been used in DSPInst on the Analog Devices Blackfin BF548 platform. The chapter also presents implementation considerations and details of DSPInst, such as code discovery, transparency, and code generation.

Chapter 4 talks about the experiment setup, including the discussion about the Blackfin architecture, hardware platform and benchmark applications we use.

In Chapter 5, we illustrate the usage of DSPInst by presenting three example analysis tools. The first one is a utility counting the occurrence of 64-bit instructions executed in applications; the second one is to profile information for computationally intensive loop execution in applications; the third one uses voltage and frequency
scaling on any instruction boundary to balance power and performance for an application.

Finally, Chapter 6 presents the conclusion of this thesis, as well as discussion of future directions of DSPInst.
Chapter 2

Background

This chapter goes through a brief background into the areas that are related to the research performed in this thesis. The first section takes a look at embedded systems, their characteristics, and current trends. The second section introduces instrumentation, different methodologies to apply instrumentation, and their pros and cons. In the last section, we survey the related instrumentation tools or frameworks, as well as the related technologies.

2.1 Embedded Systems

An embedded system is usually defined as a computer system that has pre-defined, dedicated functions with computing constraints. Embedded systems are everywhere in our life, cell phones, MP3 players, global positioning systems (GPS), even refrigerators, microwave ovens, etc. They are usually designed to perform specific functions under strict constraints. For example, a portable MP3 player is an embedded system. It has several readily apparent functions: storing, organizing and playing audio files, perhaps recording voices and an alarm. It also has some resources constrains. The
processing chip cannot be very large, or no one would carry it. The power consump-
tion must be limited, since only a small battery can be placed in the portable player,
and the battery should last as long as possible. Each embedded system is designed
to satisfy a combination of various functions and constraints.

This is different from general purpose systems, such as desktops or workstations. General purpose systems are designed to perform many different tasks, depending on the running software, as opposed to an embedded system, that is built to perform specific tasks with strict parameters.

There are three fundamental characteristics of an embedded system. First, an embedded system is designed to perform a dedicated task or a tightly scheduled set of tasks. Second, power, cost and reliability are often important factors affecting the design of an embedded system. Third, an embedded system can function as a sub-system of a larger device, and is not always a standalone device.

Embedded systems are evolving with technology breakthroughs and development of design methodologies. A very important trend is the recent emergence of multi-core processors which offer the benefits of reducing power consumption, at the same time significantly improving performance. These multi-core processors are especially suited to the embedded systems which take performance-per-watt as the most important design consideration. Another clear trend is the growth of reconfigurable computing, which is driven by an ever-increasing need for computational power with tight budgets and space, weight and power challenges. Last but not least, with the increasing complexity of embedded system design, the methodology of hardware/software codesign is gaining more and more acceptance. Hardware and software designers are working together on the specification of the embedded system and the partition between hardware and software. This concurrent design methodology has increasingly displaced
2.2 Instrumentation

In hardware or software performance analysis, there are two major techniques that influence the overall design flow. The first technique is *profiling*, which keeps the user-defined statistical information about a program or an architecture at runtime. The second technique is *tracing*, which records a complete log of all activities performed by an execution to a trace file. The profiling result is usually presented to the user immediately after the program has finished execution, and shows a compact and high-level view of the program execution on a certain architecture; while tracing usually results in a large trace file, especially for a long-time execution. However, with more detailed information, tracing can be used to reconstruct the exact behavior of an execution, and can be thought as a more general performance analysis technique.

A very important stage for both of these two techniques are *instrumentation*. During instrumentation, an instrumentation entity (either a software tool or the programmer) adds extra code into an application at interesting points, such as a certain function is called or a certain address is written. Instrumentation may be accomplished in three ways: through source instrumentation, through the use of wrapper libraries, or through binary instrumentation. While most tools only use one of these three ways, it is not impossible to use the combination of more than one of these techniques to instrument an application.

Source instrumentation adds instrumentation code directly into the source code of an application. While it achieves an easy and accurate instrumentation for special
regions of source code, changing the source code may interfere with compiler optimization. Furthermore, source instrumentation is limited to the parts only in which the source code is available.

Wrapper libraries can be used on calls made to libraries. The developer links a library provided by an instrumentation tool instead of the original library. When the library is called, it is intercepted by the instrumentation tool library. The instrumentation tool library first collects information about this call, then forwards the call to the original library. Wrapper libraries do not need source code and can be convenient by only re-linking an application against the new libraries provided by the instrumentation tool. However, wrapper libraries are limited to collecting information for library calls. One example is the wrapper libraries implemented in the Multi-Processing Interface (MPI) [20, 46] profiling interface, which are used by many performance analysis tools to collect information about MPI communication.

Binary instrumentation moves the high technical burden to instrumentation tool writers, and provides a convenient yet powerful instrumentation technique for programmers. With this technique, the executables are modified directly to insert the instrumentation code. The instrumentation may be performed statically before runtime, or dynamically at runtime. Since binary instrumentation is working directly on the executable, it decouples the instrumentation from the source code. However, to support different architectures with completely different binary file formats and instruction sets, it generally requires substantial changes for binary instrumentation tools.
2.3 Related Work

Many researching areas involve modifying binaries of applications or building tools that operate on binaries. In this section, we will survey this related work and discuss related technology features presented in these systems.

2.3.1 Hardware Simulation and Emulation

Fast architecture simulators and emulators are widely used to help profile and analyze both hardware and software systems. These systems include Shade[14] and whole-system simulators like SimICS[36] and SimOS[42]. Shade performs cross-architecture simulation, and simulates and traces the target machine. Shade is targeting SPARC (Version 8 and 9) and MIPS 1 instruction sets, which are simpler than Blackfin. Shade has a simulation interface, but no interface for building other types of tools. It specializes in trace generation and simulation, and is not extensible to other profiling and analysis tools.

Software virtual machines, including Denali[52], VMWare[10], and Xen[7], are able to run large, complex programs, such as commercial operating systems. However, they only worry about privileged instructions, and ignore all the user-mode instruction as much as possible. This means that they can not be used as tools to study application behavior except the behavior has interactions with underlying operating system. Furthermore, all of them are targeting general purpose architecture.

The primary goal of these simulators and virtual machines is to study whole-system behavior by performing the execution of the operating systems and their workloads or execution of multiple operating systems simultaneously on the same hardware. In contrast, DSPInst’s goal is to help the user build customized tools that
analyze a single application in a lightweight manner.

2.3.2 Binary Optimization

Binary optimization systems operate at the post-link stage on binaries. It burdens the code optimization task which is conventionally the domain of the compiler. OM[49] is an optimizer for executables initially implemented for DEC-stations running Ultrix/MIPS, and later ported to Digital Unix/Alpha. OM was designed as a separate pass on the result of linking, and it can also make use of profiling information. Some optimizations performed by OM include code size reduction by unreachable code removal, compaction of the memory area that holds compile time constants by elimination of unused and duplicate constants, and profile guided code positioning and alignment.

Spike[15] is an implementation of OM on the Windows NT/Alpha platform. Although Spike consists of an instrumentation part and an optimization part, the instrumentation only serves for optimization. Both parts are embedded in the Spike optimization environment(SOE), which transparently handles the task of collecting and managing profiling information for the user. The instrumentation part provides basic block and control flow edge execution frequency counts, and the important transformations of the optimization part are profile-driven.

FDPR[38] is also a feedback-based post-link optimization tool. The main optimizations of FDPR include inter-procedural register re-allocation, improving instruction scheduling, data prefetching, function inlining, and global data reordering etc. FDPR is operational on AIX4.1 and can optimize XCOFF format program binaries compiled with IBM’s XL compiler.

Dynamo[5, 4] is a dynamic optimization system for PA-RISC. Dynamo focuses on
optimization. However, it does not completely solve the transparency problem, and
does not return control to native execution if it is not performing well. It was never
scaled up to run complex applications, nor did it export an interface for customization.

Ispike[34] is a post-link optimization system on the Intel Itanium Processor Family
processors under the Linux operating system. Ispike provides a wide range of optim-
izations and uses hardware-counter profiles. By utilizing a hardware performance
counter, Ispike effectively constrains its profiling overhead. However, the profiling
capability is highly dependent on the performance counters design.

Most of above binary optimization systems are targeting general purpose or high
performance architecture. Their primary purpose is to statically or dynamically opti-
mize program binaries for corresponding architectures. Therefore, they put less focus
on the profiling interface design and the capability to design customized analysis tools
is very constrained.

2.3.3 Static Binary Instrumentation

Static instrumentation technology was pioneered by ATOM [48]. It originated in the
OM project but has since then diverged. ATOM provides a common infrastructure
and organizes the final executable of the application program and the separate user’s
analysis routines to run in the same address space at link time. The instrumentation
code is linked with an instrumentation engine to generate the instrumentation tool.
This tool will parse the executables and insert function calls at the procedure level, the
basic block level or the instruction level. Registers modified by an analysis procedure
are saved to the stack and later restored. ATOM is probably the most commonly
used static binary instrumentation toolset targeting the Digital Alpha processor.

EEL [32] is similar with ATOM in that they both supply a library that tries to
hide much of the complexity and system specific detail of modifying object files. It was designed at the University of Wisconsin-Madison and runs on Solaris/Sparc and Unix/MIPS. EEL allows a tool to analyze and modify executable programs independent of instruction sets, executable formats, or consequences of deleting existing code and adding foreign code. Theoretically, tool builders should be able to modify an executable without being aware of the details of the underlying architecture. But instrumentation code consists of concrete instructions have to be rewritten in machine language for different architectures. EEL has been used to implement the qp and qpt tools[6], which are used to obtain path profiles. However, EEL still has a problem in achieving transparency, especially when dealing with indirect jumps and calls.

The Morph [53] system is a combination of operating system and compiler technology to effectively profile and optimize the application program targeting Digital UNIX 4.0 on Digital Alpha processor. By applying statistical sampling as a profiling method, Morph achieves low-overhead continuous optimization. However, Morph does not supply the interface for user-defined profiling or optimization.

Etch[41] targets Windows NT/x86 executables. It was developed jointly at the University of Washington and Harvard University and it was strongly influenced by ATOM. To instrument an executable, Etch is invoked with the name of the executable file and a dynamic linked library (DLL). The DLL file contains the analysis procedure in the form of callback functions that are called by Etch when modifying the executable.

The instrumentation interfaces of these systems are the model of DSPInst, as well as more recent dynamic instrumentation tools. The most critical drawback of static instrumentation is lacking of support for dynamic application behavior. In another words, it is difficult for static tools to instrument the modules dynamically loaded.
Etch partially addresses this problem by a initial pass of executing the program to discover the code boundaries and dynamically linked libraries, and then instrumenting the now-known code. But there is no guarantee that all code was seen in the initial run. Similarly, DSPInst only instruments program binaries and static linked libraries. At the current stage, DSPInst is only targeting embedded applications running without operating system, and does not support instrumentation for the dynamically loaded modules.

### 2.3.4 Dynamic Binary Instrumentation

Several dynamic instrumentation systems have been developed in the last few years. They are implemented in two ways: (1) binary modification and (2) recompilation of the application at run-time. Binary modification overwrites the original program instructions with probes or trampolines. These modified instructions will invoke special code to handle the process of instrumentation. The recompilation of the application does not touch the original code. Instead, it decodes the original program binary and encodes them into a separate area of memory commonly referred to as the code cache. Instrumentation is inserted at the specified points in the code cache.

Dyninst [9], Kerninst [50], Detours [23], and Vulcan [19] modify the binary to apply instrumentation. All of them target IA-32, along with other architectures. Dyninst and Kerninst are based on dynamic instrumentation technology [22] developed as part of the Paradyn Parallel Performance Tools project, while Detours and Vulcan rely on special features of the Windows operating system. These tools are successful at observing modern, dynamic applications, but they suffer from transparency problems. Additionally, extensive modification of the code quickly becomes unwieldy through these mechanisms, especially in the face of variable-length instructions.
Recent run-time tools use code caches, giving greater control than trampoline-based methods. Pin[30], Valgrind[39], Strata[45], DynamoRIO[8] and DIOTA[35] are systems relying on the recompilation at run-time strategy to perform instrumentation. All above systems target general purpose or high performance architecture.

2.3.5 Instrumentation on Embedded System

Pin has been extended to the ARM architecture [21]. However, the authors do not discuss Thumb, the 16-bit ISA extension to ARM, because they do not deal with variable length instructions.

Another close binary instrumentation system for embedded environment is the DELI [18] developed by Hewlett-Packard and ST Microelectronics. It translates and optimizes code for LX/ST210 embedded processor. However, DELI clearly takes optimization as its goal, while DSPInst is targeting instrumentation. Furthermore, they are designed for different architectures.

DSPTune [43] is a toolset similar to DSPInst targeting the Analog Devices SHARC architecture. DSPTune can only instrument C or assembly code, which limits its usefulness when source is not available.

2.3.6 Binary Translation

Static binary translation systems are aimed at architectural compatibility. DEC’s Alpha migration tools translate VAX and MIPS to Alpha AXP architecture.[47] DEC’s FX!32[11] makes use of static translation combined with dynamic profiling and emulation to translate IA-32 Windows application to Alpha. FX!32 deliberately avoids runtime translation for the considerations of performance. Instead, it relies on ahead-of-time offline translation. UQBT[12] is designed to be able to easily migrate software
from various source to multiple target machines. The primary goal of UQBT is a general adaptable binary translation framework.

Dynamic translation systems manipulate code at runtime and at the same time translate from one instruction set to another. The mechanism is similar to instruction set emulators. They include Aries[54] for translating PA-RISC to IA-64 and Walkabout[13] translating IA-32 to SPARC. Dynamic translators do not have the pressure for performance, since they are not competing against a statically-optimized binary.

Binary translation systems involve binary rewriting. Some dynamic translation systems use similar technique with DSPInst, which jumps to the trampoline code translating and executing the target instructions. However, the primary goal of these systems is translation from one ISA to another. Therefore, none of them provides an interface for customization.
Chapter 3

Design and Implementation Details

3.1 Design Overview

The design of DSPInst allows us to add calls to analysis routines before or after any instruction in a program. A good example is the zero-overhead loop setup instruction. By detecting this instruction sequence we can detect the beginning of a computationally intensive code area (i.e. loop bodies). Identifying these structures will help to guide hot/cold optimization [15].

A transparent instruction-level instrumentation tool allows customized analysis procedures to be added at any point within the program, as specified by the user. DSPInst views a program as a linear collection of procedures, a procedure as a collection of basic blocks, and a basic block as a collection of instructions. DSPInst provides a platform for users to build their own custom analysis procedure and then will insert them at points specified by the user.

We have implemented DSPInst on top of the Analog Devices VisualDSP++ tools [27]. We utilize VisualDSP build tools to generate the final executable for
the Blackfin processor. The build tools are composed of an optimizing C compiler, Blackfin assembler, linker, loader and cycle-accurate simulator.

Internally, DSPInst works in two steps, as illustrated in Figure 3.1. In the first step, the binary executable and libraries of the application program are parsed, and instructions at user-defined address points are replaced by trampoline code. DSPInst uses the FORCE INTERRUPT instruction available in the Blackfin ISA to initiate the trampoline.

Figure 3.1: The instrumentation process
Our next task is to preserve the ISA state and then carry out the analysis prescribed in the analysis file. To modify the interrupt vector table so that control transfers to our desired analysis code, we execute a startup routine to do proper initialization.

In the second step, we generate the instrumented binary application by assembling the user’s analysis function, wrapping it with proper register protection, and linking it with the instrumented application program and startup routine.

Figure 3.2 shows an instrumented application program example. In this program, 64-bit VLIW instructions are replaced by 16-bit interrupt instructions. The interrupt instructions effect a jump from the original program to the instrumentation procedure.
To maintain transparency (in terms of addressing), we insert three NOP instructions to pad the interrupt instruction to 64 bits.

### 3.2 Code Discovery

Blackfin uses Executable and Linking Format (ELF)[16], a common standard file format for executables, object code, shared libraries, and core dumps. Figure 3.3 is a file organization of ELF files.

![Figure 3.3: Executable and Linking Format](image)

An ELF header resides at the very beginning and holds a “road map” describing the file’s organization. The Program Header defines offsets and sizes of the segments and tables and their locations in the address space. It also contains the code address
where execution starts. A program header table, if present, tells the system how to create a process image. Files used to build a process image (execute a program) must have a program header table. A section header table contains the information about the file’s sections. Every section has an entry in the table; each entry contains the information such as the section name, section size, offset address, etc. Sections hold the bulk of object file information for the linking view: program code, data, symbol table, relocation information, and so on. Various sections hold program and control information. For example, the text section holds the executable instructions of a program; the data section holds initialized data that contributes to the program’s memory image; the bss section holds zero initialized read/writable data and is therefore reduced to an address/size pair in the file representation; the debug section holds information for symbolic debugging.

Code discovery tries to locate the parts of the object file or executable that contains the instructions we are interested in. Locating code is not always straightforward. Usually read-only data, such as string constants, jump tables, floating point constants, and program code are mixed together in the text segment. This problem is aggravated if the architecture has variable length instructions (like the Intel x86 architecture). In this case, one has to be very careful when decoding instructions. Commonly, code discovery is applied with control flow graph construction. For example, decoding only starts when a new branch target address is identified and stops only when it encounters another flow-control instruction.

Fortunately, on the Blackfin platform, compilers are very disciplined, and program code and data are placed into separate areas. However, DSPInst still has to handle variable lengths of instructions (16-bit, 32-bit and 64-bit) on Blackfin.

If the program is dynamically linked, code discovery may also try to identify the
shared libraries used by the program and to parse them as well. DSPInst currently does not support dynamically linked code.

3.3 Startup Routine

Since we are utilizing the existing hardware unit in the Blackfin processor, we have to properly initialize the unit for our instrumentation. We execute a startup routine to do this initialization. This code performs the following:

- Initialize the proper entry point of the interrupt vector table according to the application binary executable,
- Initialize a memory buffer to store analysis data, and
- Create a table, in which the address of each modified instruction is stored with its corresponding index.

There are a number of issues that the startup routine needs to consider. First, the startup routine has to run in supervisor mode in order to configure the interrupt vector. Second, a memory buffer needs to be reserved to store the table of the original instructions. Third, we have to make sure that the instrumentation points are not located in areas where system priority will become an issue.

3.4 Transparency

DSPInst must avoid interfering with program execution. Transparency is critical for instrumentation. If the original program execution has been modified, all profiling information will be invalid.
3.4.1 Trampoline Instruction Selection

In terms of instruction length, the majority of instructions in Blackfin ISA are 16-bit instructions. There are also 32-bit instructions. The Blackfin processor is not superscalar; it does not execute multiple instructions at once. However, it does permit up to three instructions to be issued in parallel with some limitations. A multi-issue instruction is 64 bits in length and consists of one 32-bit instruction and two 16-bit instructions. Only a limited type of 32-bit Arithmetic Logic Unit (ALU) or Multiply Accumulate Unit (MAC) operations can be in a parallel instruction. Table 3.4.1 shows the parallel issue combinations.

<table>
<thead>
<tr>
<th>32-bit ALU/MAC instruction</th>
<th>16-bit instruction</th>
<th>16-bit instruction</th>
</tr>
</thead>
</table>

Table 3.1: Parallel Issue Combinations

To achieve transparency, we use the Blackfin FORCE INTERRUPT instruction to effect a jump from a user application to the instrumentation routine. The FORCE INTERRUPT instruction is a 16-bit instruction and is the smallest instruction in the Blackfin ISA. By using this instruction, we can replace any instructions without pushing away following instructions from their original addresses.

We could have elected to utilize the SHORT JUMP instruction, though the displacement provided would place limits on the size of the binary that we could effectively instrument. Instrumenting binaries with the FORCE INTERRUPT instruction removes this limitation.

However, for replacing 32-bit instructions or 64-bit instructions, we have to pad the FORCE INTERRUPT instruction to 32 bits or 64 bits. Otherwise, the next instruction will not hold its original address and be pulled into the blank due to different instruction lengths. Figure 3.4 shows the examples to pad the FORCE INTERRUPT instruction.
3.4.2 Program Flow Control Instructions

Some instructions can be moved to another address and still execute without disturbing original program behavior. However, not all instructions can be replaced as straightforwardly. All the instructions with relative address, such as program flow control instructions, cannot be simply moved to another address.

PC Related Instruction

Since the original instruction was copied to a table, and then executed at this new location, we need to make sure that any instructions that are PC-relative will utilize the original PC.
The Blackfin architecture can only reference the PC as a source register; the PC is only used in branch instructions (JUMP and CALL). When we instrument an instruction with the PC as a source register, we replace the original instruction with a sequence of instructions. This involves a series of steps. First, we protect the data registers that the instrumentation routine will overwrite. Second, we need to save the interrupt return register RETI value to a temporary data register. Next, we calculate the original PC value from the temporary register and the length of the original instruction. Then, we calculate the destination address from the PC value and related registers. Finally, we issue the RETI and restore the register values. When the interrupt return instruction (RTI) is executed at the end of the instrumentation routine, control will return to the correct destination address. Figure 3.5 is an example of instrumenting PC related instructions.

```
JUMP ( PC + P0 );

<table>
<thead>
<tr>
<th>Addr1 = RETI - 2 ;</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETI = Addr1 + P0 ;</td>
</tr>
<tr>
<td>RTI ;</td>
</tr>
</tbody>
</table>
```

Figure 3.5: Instrumenting PC related instruction

**JUMP, RTS and Conditional Branch Instruction**

To avoid nested interrupts, we have to make sure that the interrupt return instruction (RTI) is the only exit from the instrumentation routine. When we instrument a JUMP, a subroutine return instruction (RTS) or a conditional branch instruction, we
replace the original instruction with a sequence of instructions. In this sequence, the
destination address is calculated from the current program state so that the condition
code status flag and register values are preserved. The result will be updated by the
RETI. We properly protect any registers used in computations for the instructions,
thus the original program behavior is effectively unmodified. Figure 3.6 shows an
example of instrumenting the JUMP instruction.

<table>
<thead>
<tr>
<th>JUMP.S Addr1 ;</th>
<th>Addr2 = RETI + Addr1 – 2 ;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RETI = Addr2 ;</td>
</tr>
<tr>
<td></td>
<td>RTI ;</td>
</tr>
</tbody>
</table>

Figure 3.6: Instrumenting the JUMP instruction

CALL Instruction

In the instrumentation routine, the original CALL instruction is also replaced by an
instruction sequence. To instrument CALL instructions, we copy the address value
held in the interrupt return register (RETI) to the subroutine return register (RETS)
right before updating the destination address in RETI. By doing this, we guarantee
not only that the instrumentation routine returns correctly, but also that the called
subroutine also returns correctly. Figure 3.7 is an example of instrumenting the CALL
instruction.
Zero-overhead Loop Setup Instruction

The zero-overhead loop setup instruction provided in the Blackfin ISA is a counter-based, hardware-loop mechanism that does not incur a performance penalty when executed. This instruction also reduces code size when loops are needed. The zero-overhead loop setup instruction will initialize three registers, which are the Loop Top Register, the Loop Bottom Register and the Loop Count Register, in a single instruction. The address value initialized in Loop Top and Bottom Register are PC-relative offset. Therefore, for instrumenting this instruction, we have to replace the original instruction with a sequence that initializes the three registers accordingly. Figure 3.8 shows an example sequence for instrumenting the zero-overhead loop setup instruction.

3.4.3 Program Execution Status Protection

Before entering an analysis function, the registers are preserved by being pushed onto the stack. Before executing the original instruction, the protected register state is restored from the stack. This ensures that the analysis procedure does not perturb the original program state.
3.5 Code Generation

After we locate and properly replace the instructions we are interested, we need to generate a new version of the executable. We attach the extra code to the end of the text segment, and link the analysis procedure into the new executable.

3.5.1 Address Translation

Address translation has historically been a major problem for systems modifying a program binary. It is a consequence of the fact that the code addresses, the subroutine start addresses in particular, will usually have changed after modifying the object binary code. Several solutions have been proposed and implemented[51].

One approach is to remember the original address for each basic block. PC-relative branches and subroutine calls are easily handled statically, so are branches and subroutine calls to absolute addresses. Targets of these branches and calls are, by definition, beginnings of basic blocks. Therefore, all the system can translate old
addresses to the new addresses when generating new code. Other control transfer instructions, i.e., indirect control transfers, are handled by run-time address translation. Here a code snippet is inserted before an indirect control transfer, which translates old addresses to new addresses with the help of an additional table at runtime. The table is an array mapping new addresses to the old addresses, and is appended to the text segment. If the snippet cannot find an address in the table, it leaves it unchanged. This will allow run-time generated code to work properly. This approach has been used in prior instrumentation work[51, 31].

Another approach does all the address translation statically. This approach assumes that indirect control transfers will branch to addresses that have ultimately been loaded from memory. Hence all the static translator has to do is to find the memory locations containing code addresses and replace them with the corresponding new addresses. These memory locations are identified using relocation information.

The third approach - the one used by DSPInst - is to avoid address translation by allowing only transformations that do not change code addresses, such as deleting old code or inserting new code in the middle of the text segment. We are allowed only to substitute code. In DSPInst, we substitute an instruction with the FORCE INTERRUPT instruction, and the FORCE INTERRUPT instruction will branch to the interrupt service routine where the original instruction is executed after doing some extra work.

3.5.2 Segment Growth

Binary code modification will usually change the size of the text segment. In DSPInst, since extra code is attached to the end, the size of text segment grows. We have to
properly modify the program header and segment header to reflect these modifications.

Usually growing a segment does not create a problem if we map the text segment to the external memory. However, we have to be careful that the new text segment does not exceed the memory size when we map the text segment to the internal memory on Blackfin.
Chapter 4

Experimental Setup

4.1 Analog Devices Blackfin Architecture

The infrastructure discussed in this thesis targets the Analog Devices Blackfin family of DSPs. The specific DSP used in the results chapter of this thesis is ADSP-BF548. We next will provide a brief overview of the Blackfin family and its core architecture. More detailed information is available in the Analog Devices programmer’s reference manuals [28, 24, 29, 26].

The Blackfin DSP is a Micro Signal Architecture (MSA) based architecture developed jointly by Analog Devices and Intel Corporation. The architecture combines a dual 16-bit Multiply Accumulate (MAC) signal processing engine, flexible 32-bit Single Instruction Multiple Data (SIMD) capabilities, an orthogonal RISC-like instruction set and multimedia features into a single instruction set architecture. Combining DSP and microcontrol functionality into a single instruction set architecture enables the Blackfin to perform equally well in either signal processing or control-intensive applications.

Some of the Blackfin Instruction Set Architecture (ISA) features include:
• Two 16-bit multipliers, two 40-bit accumulators, two 40-bit arithmetic logic units (ALUs), four 8-bit video ALUs and a 40-bit shifter,

• Two Data Address Generator (DAG) units,

• 16-bit instructions (which are the most frequently used instructions),

• Complex DSP instructions are encoded into 32-bit opcodes as multifunction instructions,

• Support limited multi-issue capabilities in a Very Long Instruction Word (VLIW) fashion, where a 32-bit instruction can be issued in parallel with two 16-bit instructions, and

• A fixed-point processor, offering 8, 16, 32-bit signed or unsigned traditional data types, as well as 16 or 32-bit signed fractional data types.

The Blackfin processor supports a modified Harvard architecture in combination with a hierarchical memory structure, which is organized into two levels. The L1 memory can be accessed at the core clock frequency in a single clock cycle. The L2 memory is slightly slower, but still faster than external memory. The L1 memory can be configured as cache and/or SRAM, giving Blackfin the flexibility to satisfy a range of application requirements [44].

The Blackfin Processor was designed to be a low-power processor, and is equipped with a Dynamic Power Management Controller (DPMC). The specific frequency and voltage varies depending on the particular processor chosen in the family, but in general the processor can operate at a core clock frequency of up to 600MHz, 1.2V. Users can modify the operating frequency and voltage quickly and easily by configuring various power modes with DPMC. The normal operating mode is the Full-On mode,
in which the processor runs at the highest performance and also with the highest power cost. The next power-consuming mode is the Active Mode. At this mode, the core and system clock are forced to switch to an externally provided clock frequency. Therefore, the processor saves considerable power as well as maintains the system state and keeps monitoring its peripherals. In the Sleep mode, the core of the processor is shut down, and only peripherals and external memory are still active. The processor saves more power in this mode when not performing any work, but waiting for a peripheral. The final mode is the Deep Sleep mode. In this mode, the processor is completely disabled and can be awakened only by an interrupt, saving significant power consumption when the processor is not being used. Besides various power modes, DPMC is able to work together with the Phase Locked Loop (PLL) circuitry, allowing the user to scale both frequency and voltage, to arrive at the best power/performance frequency/voltage operating point for the target application. By cooperating DPMC and PLL, the processor is able to tune its power consumption in a fine grained manner. This diverse power management functionality allows the dynamic power consumption to be easily managed from a system level.

The Blackfin Processor has a built-in performance monitor unit (PMU) that monitors internal resources unintrusively. The PMU covers a wide range of events, including pipeline and memory stalls, and includes penalties associated with these events. Developers can use the PMU to count processor events during program execution. This kind of profiling can be utilized to better understand performance bottlenecks and opportunities for power/frequency scaling. The PMU provides a more efficient debugging utility as opposed to recreating those events in a simulation environment.

In addition to the high performance architecture, low power operation options and monitoring facilities, the Blackfin is supported by a plethora of tools, ensuring
easy and quick programming. The main development environment is VisualDSP++, which supplies a multitude of debugging and optimizing capabilities by collecting all the necessary tools required to program on a Blackfin processor. Besides this basic development environment, National Instruments has also extended its Labview product to the Blackfin processor. Users can program on a Blackfin processor through National Instruments’ graphical interface [3]. A handful of other third party toolsets are also available from companies such as the Green Hills environment [2]. There are also open source tools available for the Blackfin, including support for the GCC tool chain, uClinux kernel, and debugging tools [1]. This variety of options help programmers use Blackfin effectively and efficiently.

4.2 ADSP-BF548 EZ-KIT Lite Evaluation System

The main platform for our experiments is the ADSP-BF548 EZ-KIT Lite evaluation board. The board is designed to demonstrate the capabilities of the ADSP-BF548 Blackfin processors. It is designed to be used in conjunction with the VisualDSP++ development environment giving the designer the ability to perform advanced application code development and debugging, such as:

- Create, compile, assemble, and link C, C++ or Blackfin assembly application programs.
- Load, run, step, halt, and set breakpoints in application programs.
- Read and write data or program memory
- Read and write core or peripheral registers
- Plot memory
Figure 4.1 is the product image of ADSP-BF548 EZ-KIT Lite evaluation board. Detailed information can be found in ADSP-BF548 EZ-KIT Lite evaluation system manual[25]. Here is a brief overview of board features:

- Analog Devices ADSP-BF548 Blackfin processor, with core performance up to 600 MHz, external bus performance up to 133 MHz
- 64MB double data rate (DDR) SDRAM
- 32MB (16M x 16-bit) burst flash memory
- 2Gb NAND flash memory
- 16Mb SPI flash memory
- Advanced technology attachment packet interface (ATAPI) with 40GB HDD
- Analog audio interface
- 4.3” TFT LCD display with touchscreen
- Ethernet interface
- Keypad and push buttons
- Universal asynchronous receiver/transmitter (UART)
- Expansion interface: all ADSP-BF548 processor signals

4.3 EEMBC Benchmarks

EEMBC benchmarks target real-world applications and demands that embedded systems encounter. We use a subset of one category, the consumer applications, of the EEMBC benchmark applications[17] to apply DSPInst. They are all heavily loop based with small working set sizes and instruction footprints.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>AES encryption</td>
</tr>
<tr>
<td>CJPEG</td>
<td>JPEG encoding</td>
</tr>
<tr>
<td>DJPEG</td>
<td>JPEG decoding</td>
</tr>
<tr>
<td>HUFFDE</td>
<td>Huffman decoding</td>
</tr>
<tr>
<td>MPEG DEC</td>
<td>MPEG2 decoding</td>
</tr>
<tr>
<td>FILTER</td>
<td>RGB high pass gray filter</td>
</tr>
<tr>
<td>MP3PLAYER</td>
<td>MP3 player</td>
</tr>
</tbody>
</table>

Table 4.1: Subset of EEMBC benchmark applications
Chapter 5

Build Customized Tools: Examples

In this section, we illustrate the utility of DSPInst to build customized tools for embedded systems research. We provide three examples. All examples are developed on the ADSP-BF548 EZ-KIT Lite evaluation board, with a Blackfin ADSP-BF548 processor. We use a subset of the EEMBC Consumer Benchmark suite. Each benchmark application is run as a dedicated task without any operating system support.

5.1 Counting the occurrence of 64-bit instructions

Our first example counts the occurrences of 64-bit VLIW instructions in application programs. The Blackfin processor allows up to three instructions (one 32-bit and two 16-bit) to be issued in parallel as one 64-bit VLIW instruction (with some limitations). All three-op VLIW instructions execute all ops concurrently and commit at a rate of the slowest of the three.

To instrument a binary program, the user provides three inputs to DSPInst: 1) the application program binary that will be instrumented, which is a .dxe file, 2) the analysis procedure, which is an assembly file, and 3) an instrumentation configuration,
which defines which part of program we are interested and if we want to apply the instrumentation before or after this part of program. For this example, we instrument the program after each 64-bit instruction.

Counter disassembled:

Counter:

```
[ -- SP ] = I1 ;  //protect I1 in stack
[ -- SP ] = R7 ;  //protect R7 in stack
I1.L = 0x3ff0 ;
I1.H = 0xff90 ;  //assign I1 = 0xFF90_3FF0
R7 = [ I1 ] ;    //read the value at I1 address
R7 += 1 ;        //increase the value by 1
[ I1 ] = R7 ;    //store back the result
R7 = [ SP ++ ] ; //restore R7
I1 = [ SP ++ ] ; //restore I1
RTS ;           //return
```

Figure 5.1: An instruction-counting analysis function

Figure 5.1 shows the implementation of the counter in the analysis procedure. We first protect the registers (I1 and R7 in this example) that we are going to use in the analysis function by pushing them into the stack; then we read the current counter value from memory, increase the value by one, and write back the value to memory; at the end of the routine, the counter is saved and the original register values are restored.
Figure 5.2: The instrumentation overhead counting 64-bit instructions

Figure 5.2 shows the overhead introduced by performing instruction-counting of all 64-bit ops. We can see that the overhead varies as a function of the benchmark application. For instance, CJPEG runs with very little overhead (close to native execution). In contrast, DJPEG incurs a large overhead, and is 207x slower than native execution. This overhead has a direct relation to the instruction mix and the number of 64-bit instructions present in the code. We present this information in Table 5.1.

We further pinpoint the overhead by comparing instrumentation that tracks the instructions, but does no accounting of them. We can see in Figure 5.2 that only a very small portion of the overhead is incurred by the analysis procedure. A very large
<table>
<thead>
<tr>
<th>Application</th>
<th>Execution time (cycles)</th>
<th>VLIW Number</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>4605M</td>
<td>222</td>
<td>35M</td>
</tr>
<tr>
<td>CJPEG</td>
<td>776M</td>
<td>97</td>
<td>2K</td>
</tr>
<tr>
<td>DJPEG</td>
<td>1157M</td>
<td>755</td>
<td>5M</td>
</tr>
<tr>
<td>HUFFDE</td>
<td>7M</td>
<td>51</td>
<td>1M</td>
</tr>
<tr>
<td>MPEG DEC</td>
<td>7174M</td>
<td>746</td>
<td>96M</td>
</tr>
<tr>
<td>FILTER</td>
<td>316M</td>
<td>113</td>
<td>3M</td>
</tr>
<tr>
<td>MP3PLAYER</td>
<td>65991M</td>
<td>450</td>
<td>426M</td>
</tr>
</tbody>
</table>

Table 5.1: Counting the occurrence of 64-bit instructions in the EEMBC benchmark.

portion of the overhead is attributable to the instrumentation routine section that searches the instruction table and tries to find the index of the original instruction.

5.2 Profiling Loop Execution

When analyzing program performance, the programmer usually gives special attention to the most active, or hot, code regions, and the most active code regions are usually implemented as loops or nested loops. Moreover, in the current multi-threaded and multi-core era, an increasing amount of research focuses on parallelizing loops for better performance[33, 40]. Loop-centric profiling tools[37] help the developer collect information about loop execution and parallelize the loops. It is important for the instrumentation tools to identify and profile information for loop execution.

In the second example, we present how to utilize DSPInst to profile information for loop execution in the application, and find the most computationally expensive loop by analyzing the collected information.

The Blackfin architecture has dedicated registers and hardware units for loops, and there are zero-overhead loop setup instructions to configure these registers. By instrumenting these zero-overhead loop setup instructions, we can collect information
for each loop in the application.

In the instrumented application, program flow will be branched to an analysis function after the execution of each zero-overhead loop setup instruction. Figure 5.3 is the design flow of the analysis function.

![Design flow of loop analysis function](image)

**Figure 5.3: Design flow of loop analysis function**

From executing zero-overhead loop setup instructions, we dynamically obtain the
loop start address, the loop end address and the initial value of the loop counter. We define:

\[ \text{loop\_size} = (\text{loop\_end\_address} - \text{loop\_start\_address}) \times \text{loop\_counter} \]

The bigger the loop size is, the more likely the program will spend more time at this loop. However, loop size calculated by this equation is not always accurate. It is possible that there are multiple entries or exits in this loop. For example, the Blackfin compiler will compile a “while” instruction into a loop with 0xFFFFFFFF as the loop counter value. In the real execution, the program will leave this loop though the loop counter has not decreased to zero. In this case, the extremely large value in the loop counter only indicates a possible infinite loop, instead of a real active code region. In our loop analysis function, we already identify this infinite loop and do not count it as a possible candidate.

<table>
<thead>
<tr>
<th>Loop range</th>
<th>32 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop counter value</td>
<td>115200</td>
</tr>
<tr>
<td>Loop size</td>
<td>3686400</td>
</tr>
<tr>
<td>% of whole execution time</td>
<td>64.61%</td>
</tr>
</tbody>
</table>

Table 5.2: Profiling information for loop execution in HUFFDE

Table 5.2 lists the loop execution information in HUFFDE. This loop has the biggest loop size, and it takes up 64.61% of the execution time of the whole application.
5.3 Performing Dynamic Voltage and Frequency Scaling

Dynamic voltage and frequency scaling (DVFS) has been shown to be an effective method to reduce energy consumption. Power/energy management is important in many embedded system applications. The main idea behind DVFS is to dynamically scale down the voltage and slow down the frequency of microprocessor to a “just enough” clock speed to process the workload while still meeting the timing constraints. By doing so, the system incurs little or no performance penalty, and reduces energy consumption.

In our next demonstration, we present how to utilize DSPInst to dynamically scale voltage and frequency on the Blackfin processor. The goal is to balance performance and power consumption.

<table>
<thead>
<tr>
<th></th>
<th>without voltage/frequency scaling</th>
<th>with voltage/frequency scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>500MHz</td>
<td>250MHz</td>
</tr>
<tr>
<td>voltage</td>
<td>1.20V</td>
<td>1.00V</td>
</tr>
<tr>
<td>cycles</td>
<td>138.9K</td>
<td>72.7K</td>
</tr>
<tr>
<td>stall cycles</td>
<td>131.2K</td>
<td>63.7K</td>
</tr>
</tbody>
</table>

Table 5.3: Configurations with and without voltage/frequency scaling

As in the previous example, the user specifies the application binary executable and the analysis code to scale the voltage and frequency. Selecting the appropriate points in the binary to apply voltage and frequency scaling further illustrates the value of good instrumentation tools. We attempt to scale the voltage and frequency at two user-specified points in the DJPEG benchmark application. The first point is at the beginning of a memory intensive region that accesses external memory in the application program. At this point, if we can slow down the DSP core’s frequency
from 500MHz to 250MHz, while maintaining the external memory speed of 125MHz, we should be able to save some energy, while not impacting performance severely. To accomplish this, we also scale down the DSP core’s operating voltage from 1.20V to the lowest possible voltage at 1.00V [26]. Further, we configure the performance monitor unit to record how long the processor stalls due to memory delays. According to the monitor, 94.5% of the processor cycles are spent waiting on memory operations to complete within this specific code region.
The second point in the execution where we want to insert a call to our analysis routine is immediately after the memory intensive region. We then reset the voltage and frequency to their original values at this point.

Figure 5.4 shows the instrumented DJPEG benchmark application. Program execution is interrupted by a “FORCE INTERRUPT” instruction at the selected instrumentation point. In the interrupt service routine, voltage and frequency are then scaled.

Figure 5.5: Execution Time and Energy Estimation with or without VFS

Figure 5.5 shows the normalized performance and energy estimation in the instrumented code region and the impact on the overall DJPEG application. External memory is much slower than the processor speed. Since the processor will stall frequently due to the memory operations, if we scale voltage/frequency during this
memory intensive code region and this code region is long enough, we can save power and at the same time amortize the cost of scaling. For the selected code region in DJPEG, we incur an overhead of 4.6% in execution time while saving an estimated 63.7% in energy consumption when compared with the full-speed execution. Since the instrumented code region constitutes 22.3% of DJPEG’s entire dynamic execution time, we save 14.2% in energy consumption with a 1% performance penalty (again compared to full-speed execution). In this example we utilize DSPInst to both identify the selected code region, as well as perform scaling. This demonstrates the value of providing sophisticated instrumentation tools in an embedded system.

For this application, when we scale the voltage/frequency and reconfigure the PLL within the processor, multiple clock cycles are consumed to allow the PLL to stabilize. In our example, approximately 1,000 processor cycles were consumed whenever we adjusted the voltage/frequency. Although our instrumentation tool enables the user to perform fine-grained instrumentation, the nature of voltage/frequency scaling is coarse grained, and thus does not take full advantage of the capabilities of DSPInst.
Chapter 6

Conclusions and Future Work

6.1 Conclusion

As embedded systems are deployed in a rapidly increasing number of applications, and as the sophistication of these applications continues to grow, the need for high quality instrumentation and analysis tools also grows. Until recently, binary instrumentation tools targeting embedded systems have not been made readily available.

In this thesis, we present the design, implementation, and performance of DSPInst, a static binary instrumentation toolset for the Analog Devices Blackfin family of DSPs. DSPInst achieves transparent and comprehensive instrumentation for all types of executed instructions on the Blackfin architecture. We further provided three examples to illustrate how to use DSPInst to apply instrumentation to EEMBC benchmark applications on the ADSP-BF548 EZ KIT Lite evaluation platform. In these examples, we also illustrate how a user can build efficient, customized tools with DSPInst to collect run-time profiles, to dynamically analyze profiling data and to dynamically adjust system operating conditions.
6.2 Future Work

DSPInst opens up new opportunities for software analysis and design space exploration on the Blackfin architecture. Designers can use this toolset to rapidly prototype new ideas and debug bottlenecks. In this chapter, we will discuss several potential research directions based on DSPInst.

6.2.1 Software Analysis on Blackfin

Understanding software applications is the key to effective embedded platform design. Conversely, understanding the architecture is the key to effectively migrating the existing software applications. DSPInst allows its user to design analysis tools that gather the desired information on Blackfin architecture. These analysis tools are able to be applied to other applications or libraries, regardless of whether or not the source code of the applications is available. DSPInst was designed to meet the need for software analysis on the Blackfin architecture.

6.2.2 Compiler Design

Instrumentation tools also enable compiler developers to understand and optimize code generation techniques. DSPInst can help Blackfin compiler developers to analyze the generated object module, and to find the efficient and inefficient parts. By feeding back these analyses, DSPInst helps to improve the compiler.

6.2.3 Architectural Compatibility

The Blackfin architecture is constantly under extension and refinement. Within the past several years, we have seen more than thirty Blackfin DSPs coming out to meet
diverse needs from the market. The main difference between these processors may be memory management, ability to multiprocess, or media support. It is not necessary that the instruction sets of these processors are compatible with each other. Incompatible instructions will result in an illegal exception when executed on non-compatible processors. One possible application of DSPInst is to detect and emulate any incompatible instructions, so that the application program can be reused on other Blackfin platforms. Another interesting application is to use DSPInst to emulate and evaluate the effects of proposed new ISA extensions before they are actually implemented on any existing hardware.
Bibliography


conference on Compilers, architecture and synthesis for embedded systems, pages 261–270, New York, NY, USA, 2006. ACM.


