A Binary Instrumentation Tool for the Blackfin Processor

Enqiang Sun
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA, U.S.A
esun@ece.neu.edu

David Kaeli
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA, U.S.A
kaeli@ece.neu.edu

ABSTRACT

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 sensitive to performance bottlenecks, memory leaks, and software inefficiencies. There is a growing need to develop more sophisticated profiling and instrumentation tools in this rapidly growing design space.

In this paper we describe, DSPInst, a binary instrumentation tool for the Analog Device's Blackfin family of Digital Signal Processors (DSPs). DSPInst provides for fine-grained control over the execution of programs. Instrumentation tool users are able to gain transparent access to the processor and memory state at instruction boundaries, without perturbing the architected program state. DSPInst provides a platform for building a wide range of customized analysis tools at an instruction level granularity. To demonstrate the utility of this toolset, we provide an example analysis and optimization tool that performs dynamic voltage and frequency scaling to balance performance and power.

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Code generation, Optimization

General Terms

Languages, Management, Measurement, Performance

Keywords

DSPInst, Binary instrumentation, Embedded architectures, Dynamic voltage and frequency scaling

1. INTRODUCTION

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 *dynamic instrumentation*; instrumentation applied at compile time or link time is called *static instrumentation*.

Instrumentation techniques in the embedded domain are relatively immature, though due to the increased sophistication of recent embedded systems, 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 need for improved instrumentation frameworks for digital signal processing systems continues to grow [18, 11].

In this paper 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 modified the binary executables of the applications before run-time.

The remainder of the paper is organized as follows. Section 2 introduces the Blackfin DSP architecture used in our work. Section 3 presents an survey of related instrumentation tools. Section 4 presents the design details of DSPInst. In section 5, we illustrate the usage of DSPInst by presenting an example utility. Section 6 concludes the paper and discusses future directions.

2. ANALOG DEVICES BLACKFIN DSP ARCHITECTURE

The infrastructure discussed in this paper targets the Analog Devices Blackfin family of DSPs. The specific DSP used in our example application is the ADSP-BF548. We start by providing a brief overview of the Blackfin family and its core architecture. More detailed information is available in the Analog Devices Programmers Reference Manual [4, 1, 5, 2].

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-

WBIA '09, Dec 12, New York City, NY

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 in a single instruction set 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 represent 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 [19].

The Blackfin Processor was designed to be a low-power processor, and is equipped with a Dynamic Power Management Controller (DPMC). The DPMC works 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.

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 voltage/frequency scaling. The PMU provides a more efficient debugging utility as opposed to recreating those events in a simulation environment.

3. RELATED WORK

High quality profiling and instrumentation techniques have been developed to support general computing platforms. Tools that operate on a binary format can be further divided into two categories based on *when* instrumentation is applied. Probably the most commonly used static binary instrumentation toolset ever developed was ATOM, which targeted the Digital Alpha processor [21]. Other commonly used instrumentation toolsets include EEL [14], Etch [17] and Morph [24].

The second class of binary instrumentation tools we consider are dynamic instrumentation. A number of high quality dynamic instrumentation tools have been developed in recent years, and include Dyninst [7], Kerninst [22], Detours [12] and Vulcan [10]. These tools dynamically modify the original code in memory during execution in order to insert instrumentation *trampolines* (i.e., a mechanism that jumps betweeen the instrumented code and analysis code, and back again). Most of these dynamic instrumentation systems do not address instrumentation transparency, preserving the architected state of the processor. Other dynamic instrumentation systems use caches and dynamic compilation of the binary, and include Valgrind [16], Strata [20], DynamoRIO [6], Diota [15] and Pin [13]. Most of these tools target general purpose architectures.

Instrumentation tools targeting embedded systems need to satisfy a different set of requirements [11]. Some of these constraints include power efficiency and lack of operating system support. Many of these issues force us to take a minimalists approach to instrumentation (i.e., less is more). The DELI [9] developed by Hewlett-Packard and ST Microelectronics provides utilities to manipulate VLIW instructions on the LX/ST210 embedded processor. DSPTune [18] is a toolset similar to DSPInst targeting the Analog Devices SHARC architecture. However, DSPTune can only instrument C or assembly code, which limits its usefulness when source is not available.

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 built easily.
- 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. In the next section we will delve into the implementation details of DSPInst.

4. DESIGN AND IMPLEMENTATION DETAILS

4.1 Design Overview

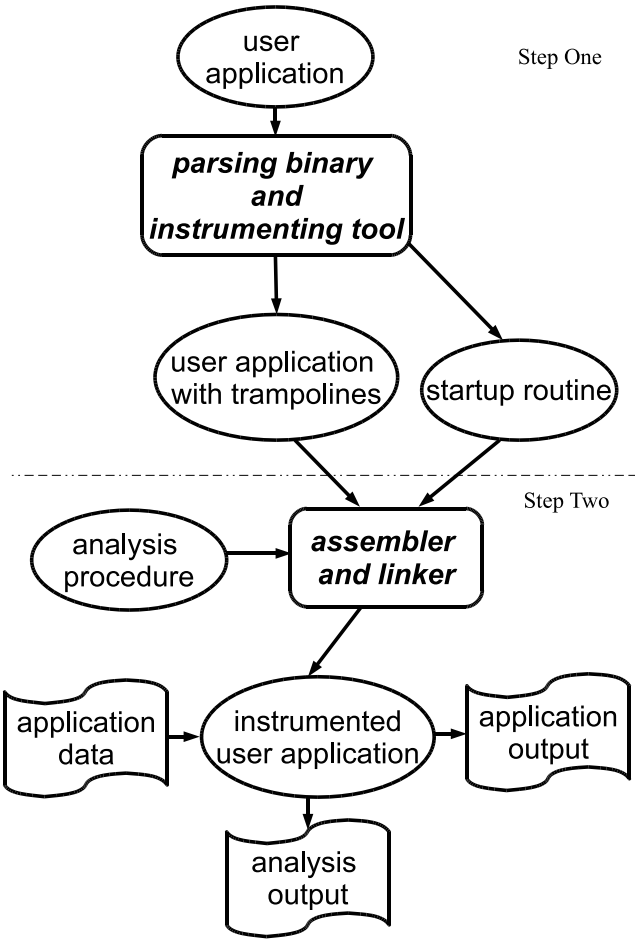

Figure 1: The instrumentation process

The design of DSPInst allows us to insert calls to analysis routines before or after any instruction in a program. For example, if we wanted to perform some analysis on the execution of every loop in a program on a Blackfin DSP, we would choose to instrument every zero-overhead loop setup instruction sequence. By detecting loop setups, we can detect the beginning of computationally-intensive loop bodies. Identifying these structures can help guide hot/cold optimization [8].

A transparent instruction-level instrumentation tool allows customized analysis procedures to be added at any points 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 procedures that can be inserted at any point specified by the user.

We have implemented DSPInst on top of the Analog Devices VisualDSP++ tools [3]. We utilize VisualDSP's 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 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.

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. This code performs the following:

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

There are two 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, we have to make sure that the instrumentation points are not located in program areas where system priority will become an issue.

In the second step, we generate the instrumented binary application by assembling the user's analysis functions, *wrapping* them with the proper register protection, and linking them with the instrumented application program and startup routine.

Figure 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 out to a full 64 bits.

The instrumentation procedure runs in the same address space as the application. This helps to insure that a precise hardware state and program information are preserved and presented to the analysis function at all times.

4.2 Transparency

DSPInst must avoid interfering with program execution, since transparency is critical for instrumentation. If the original program execution has been modified, the integrity of the profiled state will be compromised.

4.2.1 Trampoline Instruction Selection

In terms of instruction length, the majority of instructions in Blackfin ISA are 16-bit instructions. There are also 32-bit

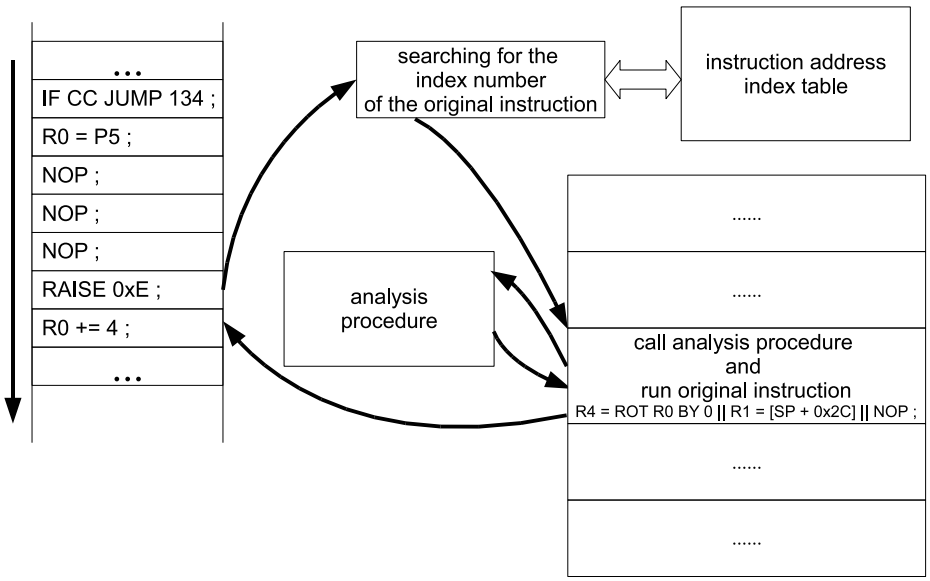

Figure 2: An instrumented application

instructions. The Blackfin processor is not a 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. Figure 3 shows the parallel issue combinations.

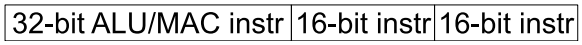

Figure 3: Parrallel issue combinations

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 16-bit instruction, we can replace any instruction without impacting the address of any other instruction in the binary.

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 remain at its original address. Figure 4 shows examples of how to pad the FORCE INTERRUPT instruction.

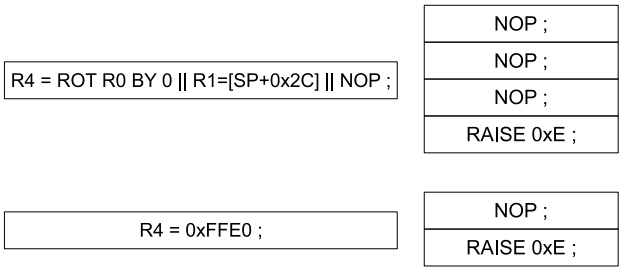

Figure 4: Padding the FORCE INTERRUPT instruction

4.2.2 Program Flow Control Instructions

Some instructions can be moved to another address and still execute without disturbing the original program behavior. However, not all instructions can be replaced as straightforwardly. All instructions that use relative addressing e.g., program flow control instructions are not easily moved to another address.

PC related instruction. The Blackfin architecture can only reference the PC as a source register; the PC is only used in branch instructions (JUMPs and CALLs). 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.

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 either 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.

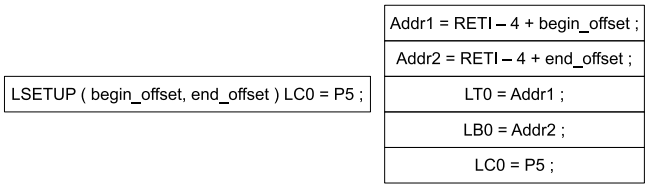

Figure 5: Instrumenting the Zero-overhead Loop Setup instruction

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 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 that not only the instrumentation routine returns correctly, but that the called subroutine also returns correctly.

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 Loop Top Register, Loop Bottom Register and Loop Count Register, in a single instruction. The address values initialized in the Loop Top and the Bottom Register are PC-relative offsets. Therefore, for instrumenting this instruction, we have to replace the original instruction with a sequence that initializes the three registers, accordingly.

4.2.3 Program Execution Status Protection

Before entering an analysis procedure, 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.

Any deviation from native execution will introduce references into the instruction and data footprint in caches, especially when we map the analysis function and instruction search table to the external memory. We either avoid profiling the cache during these periods, or map analysis functions and the instruction search table to the internal L1 SRAM. The total on-chip L1 memory for ADSP-BF548 processor is 80 KB, including 48 KB of instruction SRAM and 32 KB of data SRAM. Besides the configurable 16 KB of instruction cache/SRAM and 32 KB of data cache/SRAM, we have extra space that can be allocated for instrumentation. Instruction and data accesses to the internal L1 SRAM will not perturb the cache state.

To obtain accurate profiling information, we can protect hardware performance counters before entering any analysis procedure by disabling them. If necessary, the events occurring during the execution of an analysis procedure will be ignored and are not counted.

5. BUILDING A CUSTOMIZED TOOL: PERFORMING DYNAMIC VOLTAGE AND FREQUENCY SCALING

In this section, we illustrate the utility of DSPInst to build customized profiling and optimization tools for embedded system research. In this example, we utilize DSPInst to identify code regions that incur significant stalls during execution. If these stalls are due to cache misses, we have the ability to slow down the processor to better match the latency of the off-chip SDRAM. This scaling can save power, while not sacrificing performance (if performance is impacted, so will the energy budget). We begin by presenting our profile-guided dynamic voltage and frequency scaling(DVFS) algorithm. Then we provide an implementation of this algorithm on the Blackfin ADSP-BF548 EZ-KIT platform and discuss the benefits obtained.

5.1 DVFS decision algorithm

The advantage of static instrumentation driven DVFS is that the optimization decision is made offline before the actual application run. The decision process avoids consuming valuable native execution cycles. Furthermore, additional offline analysis has a chance to be conducted. One example is that we can merge the neighboring code regions with same DVFS decision to further constrain the overhead. The disadvantage of a static instrumentation driven approach is that the profiling data and decision is based on a static program behavior and so cannot react to changes in application behavior.

In our static instrumentation driven DVFS, we profile the

application and make DVFS decisions in the first run. As soon as the decisions are made, DVFS instructions are inserted as an instrumentation procedure in the application program to balance performance and power consumption.

Like other instrumentation based optimization techniques [23], it is important to select the best candidate code regions. To be cost effective, we want to select code regions that are frequently executed. At the same time, the regions need to consume enough time to amortize the cost of scaling; voltage and frequency scaling is a relatively slow process, taking about 1000 clock cycles on the Blackfin for the Phase Lock Loop to stabilize.

In our design, we construct a profiling tool on DSPInst to apply instrumentation at each *Zero-overhead Loop Setup* instruction. By instrumenting these instructions, we divide program execution into phases that have a loop execution in the middle and Zero-overhead Loop Setup instructions on the boundaries. The analysis function of the profiling tool configures the hardware performance counters and collects information about performance counter events, execution time, start address and end address, etc. This provides us with the necessary information to decide when to apply voltage/frequency scaling.

In order to be beneficial for DVFS, the selected code region first needs to run long enough to amortize the scaling overhead. But it is not sufficient that the code regions is long running. We need other metrics to complete the criterion by analyzing the DVFS decision model.

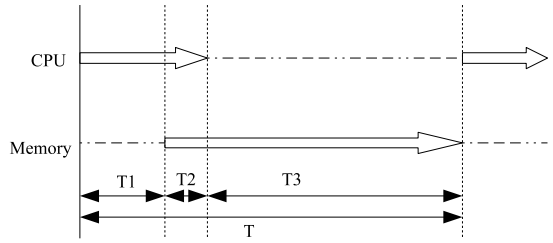

Figure 6: Our DVFS decision model

Figure 6 is our DVFS decision model. T1 is a CPU bound execution phase, and T2 and T3 are memory bound execution phases. It is possible that the CPU and memory are executing concurrently due to multiple issued instructions, such as in T2. Due to the long latency of external memory accesses, T3 is still significantly long, although the CPU and memory is possible to run concurrently. In memory bound code regions, the CPU is stalled waiting for data cache misses to be serviced from external memory. For the whole execution time T, T3 takes up a significant portion.

In general, scaling down the CPU voltage and frequency will surely decrease processor power consumption, but it will also slow down the CPU execution speed in phase T1 and T2. However, the memory system clock is usually independent of CPU clock and so it is possible to maintain the memory speed as fast as possible, so that T3 doesn't change much. From these observation, we can scale down the voltage and frequency of the processor and reduce the number of CPU stalls during long latency memory operations, such as T3, thus saving energy without incurring a high performance penalty.

One class of events identifying phase T3 are data cache misses, which can be directly obtained from hardware performance counters on the Blackfin. Assuming, on average, a data cache miss take N CPU clock cycles, the percentage of CPU slack time during time period T is:

$$\beta = \frac{data\ cache\ misses \cdot N}{execution\ time}$$

In this equation, execution time is represented in CPU (versus SDRAM or board clock) cycles.

Therefore, we define this new metric:

$$misses\ per\ cycle = \frac{data\ cache\ misses}{execution\ time}$$

The higher the data cache misses per cycle, the greater percentage of time the processor will be stalled waiting for data from external memory.

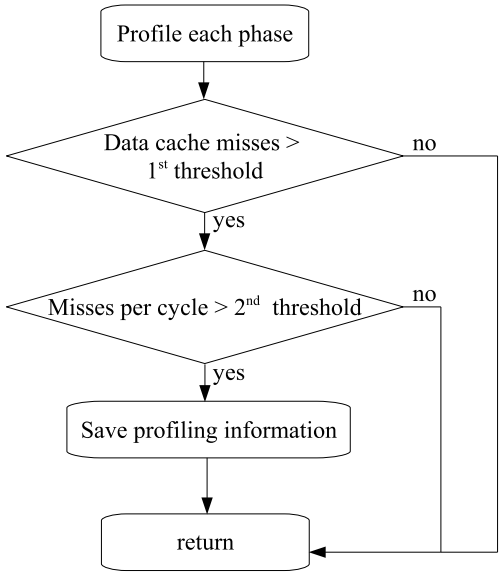

Figure 7: The DVFS decision algorithm.

Figure 7 shows a flow diagram of the DVFS decision algorithm. We set two thresholds in the decision algorithm to filter ineligible execution phases. The first threshold is the data cache miss number incurred during each program phase

– phases with a low number of misses will be pruned. The second threshold is the number of misses per cycle, which has to be larger than a set threshold. With these two thresholds, we filter the ineligible phases and select only program execution phases that contribute significantly to the overall CPU slack time.

Once we find the candidate code regions, DVFS instructions will be inserted at every entry point of the code region to scale down the voltage and frequency, as well as at the exit points in the code region to restore the voltage and frequency level. We merge the neighboring candidate code regions to further reduce the overhead.

5.2 Experimental Result

We implement the example on the ADSP-BF548 EZ-KIT Lite, with a Blackfin ADSP-BF548 processor. Figure 8 is the product image of ADSP-BF548 EZ-KIT Lite evaluation system. We configure 16 KB L1 instruction cache and 32 KB L1 data cache, and all the program code and data are mapped to external SDRAM memory. We use the consumer applications subset of the EEMBC Consumer benchmark suite as benchmark applications.

Figure 8: Product image of ADSP-BF548 EZ-KIT Lite Evaluation System.

DJPEG	JPEG decoding
MPEG4 DEC	MPEG4 decoding
MPEG4 ENC	MPEG4 encoding
MP3PLAYER	MP3 player

Table 1: Consumer applications subset of EEMBC benchmark suite

We take MPEG4 DEC as an example. The analysis function selected 21 program phases from from potential candidates (1.5 million in this case), and identified the start and ending addresses for each program phase. We further confirm the output with an offline analysis of the profiling information in Figure 9.

Figure 9 shows the number of dynamic data cache misses occurring in MPEG4 DEC. We can easily see that there are significant opportunities to apply voltage/frequency scaling in this application. Based on the selected program phases,

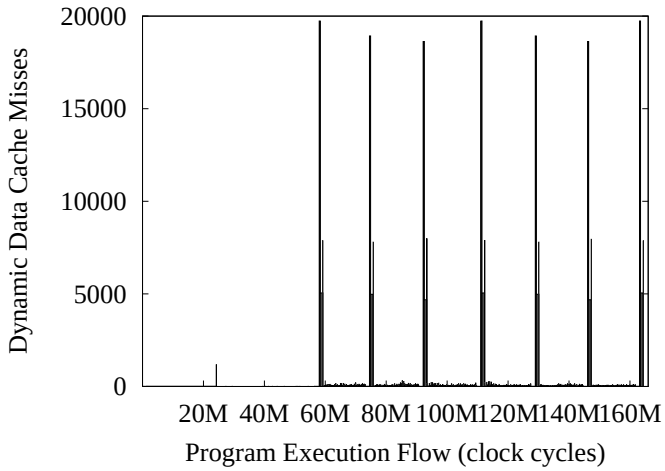

Figure 9: Dynamic Data Cache Misses in MP4DECODER.

we then use an optimizing tool to apply DVFS as the configurations in the Table 2 .

	without VFS	with VFS
frequency	500MHz	250MHz
voltage	1.20V	1.00V

Table 2: Configurations with and without voltage/frequency scaling.

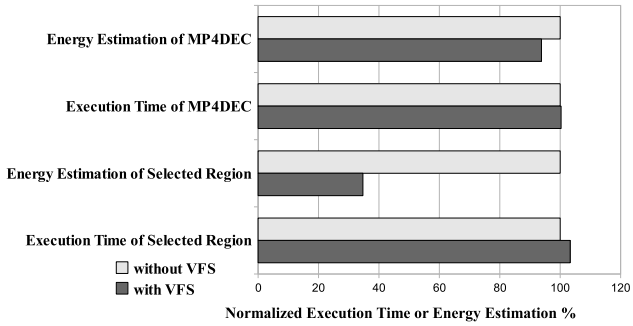

Figure 10: Execution Time and Energy Estimation with or without VFS.

Figure 10 shows the normalized performance and energy usage in the instrumented code region and the impact on the overall MPEG4 DEC application. For the selected code region in MPEG4 DEC, we incur a 3.3% overhead in execution time while reducing energy consumption by 65.3% when compared to the full-speed execution. Since the instrumented code region constitutes 9.5% of MPEG4 DEC's entire dynamic execution time, we save 6.2% in energy consumption with a 0.3% performance penalty.

We applied the same profiling and optimization tools to the other consumer applications. On average, we saved 7.6% in energy consumption, with only a 0.8% performance impact.

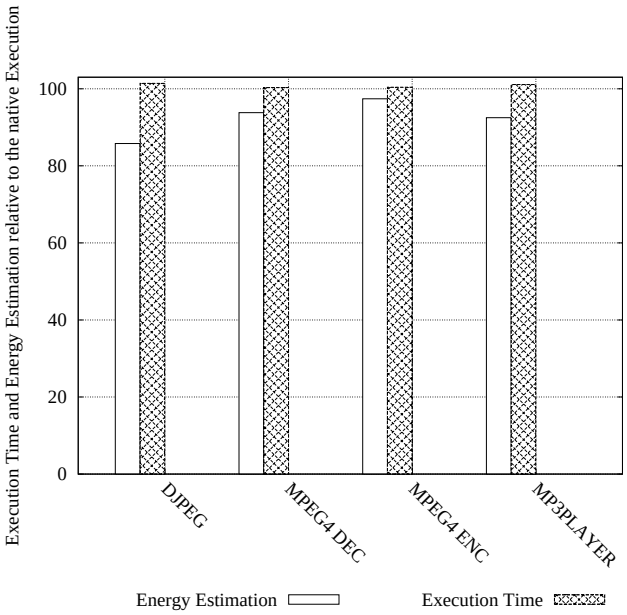

Figure 11: Execution time and energy estimation with or without VFS of consumer applications.

6. CONCLUSIONS

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 readily available. In this paper, we present design, implementation, and performance analysis of DSPInst, a static binary instrumentation toolset for the Analog Devices Blackfin family of DSPs. We have provided an example utility to illustrate how a user can build customized tools with DSPInst to both collect run-time profiles, and to dynamically adjust system operating conditions.

DSPInst opens up new opportunities for software analysis and design space exploration on the Blackfin architecture. In future work we plan to consider how DSPInst can generate standard profiles that can be used during compilation. We will also consider how best to integrate the toolset into the ADI VisualDSP framework.

7. ACKNOWLEDGMENTS

The authors would like to thank Richard Gentile and Kaushal Sanghai at Analog Devices for their help and guidance with the Blackfin tools. The Northeastern University Computer Architecture Laboratory is supported by a generous research gift from Analog Devices.

8. REFERENCES

[1] Analog Devices Inc. Blackfin processors webpage. http://www.analog.com/embedded-processing-dsp/processors/en/index.html.

[2] Analog Devices Inc. Blackfin Embedded Processor ADSP-BF542/ADSP-BF544/ADSP-BF547/ADSP-BF548/ADSP-BF549 Data Sheet, 2009.

[3] Analog Devices Inc. VDSP++ 5.0 Product Release Bulletin., August 2007.

[4] Analog Devices Inc. Adsp-bf54x blackfin processor hardware reference., August 2008.

[5] Analog Devices Inc. Blackfin processor programming reference., September 2008.

[6] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Cambridge, MA, USA, 2004. Supervisor-Amarasinghe,, Saman.

[7] B. Buck and J. K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[8] R. Cohn and G. Lowney. Hot cold optimization of large windows/nt applications. In *Proc. MICRO29*, pages 80–89, 1996.

[9] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: a new run-time control point. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 257–268, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[10] A. Edwards, H. Vo, A. Srivastava, and A. Srivastava. Vulcan: Binary transformation in a distributed environment. Technical report, 2001.

[11] K. Hazelwood and A. Klauser. A dynamic binary instrumentation engine for the arm architecture. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 261–270, New York, NY, USA, 2006. ACM.

[12] G. Hunt and D. Brubacher. Detours: binary interception of win32 functions. In *WINSYM'99: Proceedings of the 3rd conference on USENIX Windows NT Symposium*, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[13] C. keung Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *In Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[14] J. R. Larus and E. Schnarr. Eel: machine-independent executable editing. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 291–300, New York, NY, USA, 1995. ACM.

[15] J. Maebe, M. Ronsse, and K. D. Bosschere. Diota: Dynamic instrumentation, optimization and transformation of applications. In *In Proc. 4th Workshop on Binary Translation (WBTą́02)*, 2002.

[16] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[17] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of win32/intel executables using etch. In *NT'97: Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, pages 1–1, Berkeley, CA, USA, 1997. USENIX Association.

[18] S. Sair, G. Olivadoti, D. Kaeli, and J. Fridman. Dsptune: A performance evaluation toolset for the

sharc signal processor. In *SS '00: Proceedings of the 33rd Annual Simulation Symposium*, page 51, Washington, DC, USA, 2000. IEEE Computer Society.

[19] K. Sanghai, D. Kaeli, A. Raikman, and K. Butler. A code layout framework for embedded processors with configurable memory hierarchy. In *Proceedings of the 5th Workshop on Optimizations for DSP and Embedded Systems*, pages 29–38, 2007.

[20] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society.

[21] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM.

[22] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 117–130, Berkeley, CA, USA, 1999. USENIX Association.

[23] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 271–282, Washington, DC, USA, 2005. IEEE Computer Society.

[24] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automatic profiling and optimization. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 15–26, New York, NY, USA, 1997. ACM.