Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 507896, 7 pages
http://dx.doi.org/10.1155/2015/507896

Research Article

Implementing and Optimizing of Entire System Toolkit of VLIW DSP Processors for Embedded Sensor-Based Systems

Xu Yang,[1] Mingbin Zeng,[2] and Yanjun Zhang[3]

[1]*School of Software, Beijing Institute of Technology, Beijing 100081, China*
[2]*College of Technology Management, University of Chinese Academy of Sciences, Beijing 100081, China*
[3]*School of Information and Electronics, Beijing Institute of Technology, Beijing 100081, China*

Correspondence should be addressed to Xu Yang; yangxu@tsinghua.edu.cn

Received 12 March 2015; Accepted 5 July 2015

Academic Editor: Ivan Lanese

VLIW DSPs can largely enhance the Instruction-Level Parallelism, providing the capacity to meet the performance and energy efficiency requirement of sensor-based systems. However, the exploiting of VLIW DSPs in sensor-based domain has imposed a heavy challenge on software toolkit design. In this paper, we present our methods and experiences to develop system toolkit flows for a VLIW DSP, which is designed dedicated to sensor-based systems. Our system toolkit includes compiler, assembler, linker, debugger, and simulator. We have presented our experimental results in the compiler framework by incorporating several state-of-the-art optimization techniques for this VLIW DSP. The results indicate that our framework can largely enhance the performance and energy consumption against the code generated without it.

1. Introduction

Very Long Instruction Word (VLIW) architecture [1], which first appeared in 1972, typically has multiple functional units (FUs) and is capable of executing several instructions in parallel within one single clock cycle and thus is granted the ability to largely improve the Instruction-Level Parallelism (ILP).

VLIW architecture is now widely used in commercial process designs, such as NXP's TriMedia media processors, Analog Devices' SHARC DSP, Texas Instruments' C6000 DSP family, STMicroelectronics' T200 family based on the Lx architecture, Tensilica's Xtensa LX2 processor, and Intel's Itanium IA-64 EPIC, in both the embedded domain and the nonembedded domain.

While it can be exploited to largely improve the ILP, it also brings a large challenge for the development of software toolkit for VLIW architecture. As compiler is the main responsibility for the code generation of VLIW architecture, VLIW's advantages come largely from having an intelligent compiler that can schedule as many instructions as possible into parallelization to maximize the total ILP [2]. Thus, the design of compiler for VLIW is most critical. A full design toolkit is also very important, both for the consideration of time-to-market of embedded systems and for the convenience of application programming and performance verification.

In this paper, we presented our methods and experiences to develop system toolkit flows for a VLIW DSP architecture. This VLIW DSP architecture is a scalable VLIW DSP architecture. Our system toolkit consists of compiler, assembler, linker, debugger, and simulator. The compiler is retargeted from Open64 compiler, and various issues are addressed to support optimization including software pipeline and SIMD code autogeneration. Assembler, linker, and debugger are developed based on Binutils. Finally, a cycle-approximate simulator has been developed based on Gem5. Benchmarks are evaluated on this framework, and results are presented. The results indicate that our framework can largely enhance the performance.

The remainder of this paper is organized as follows: Section 2 describes the target VLIW DSP architecture. In Section 3, the methods and experiences concerning the development of the whole toolkit are presented. In Section 4,

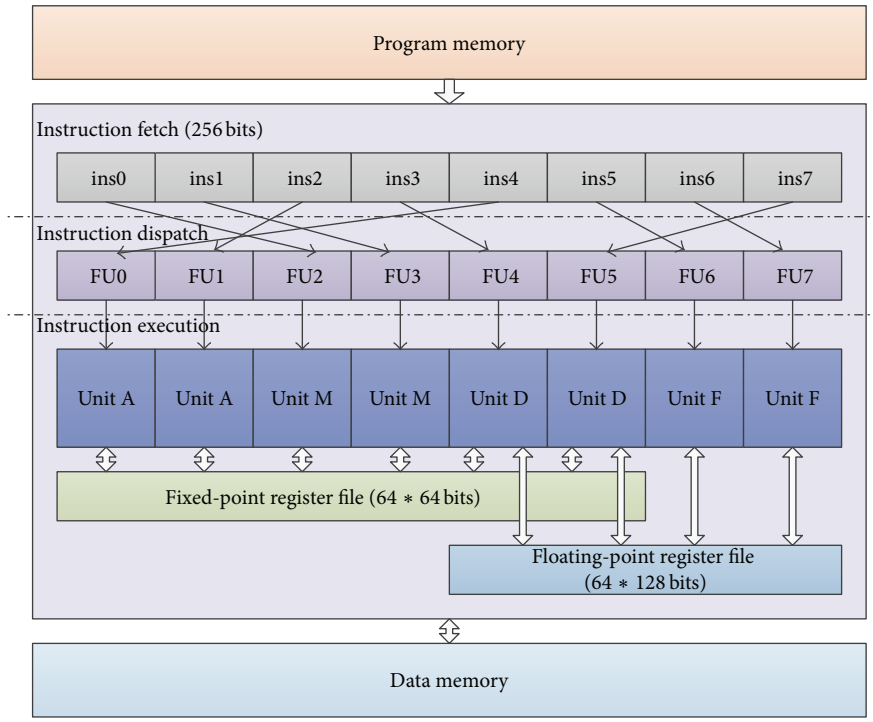

Figure 1: An example of Magnolia architecture.

related works are discussed. We present experimental results in Section 5 and conclude this work in Section 6.

2. The Target VLIW DSP Architecture

Magnolia is designed for sensor-based embedded systems [3]. It aimed at high performance and low energy consumption. Magnolia is a totally scalable VLIW DSP architecture, including the type of functional units (FUs), the number of register files, the number of registers in each register file, the number of clusters, the number of FUs and register files in each cluster, the type of instructions, and the execution time of instructions in each type of FU.

A FU library has already been developed, which includes four different types of FUs, which are Unit A, Unit M, Unit D, and Unit F, respectively. Unit A, Unit M, and Unit D are fixed-point units, while Unit F is a floating-point unit. Unit A is dedicated to executing arithmetic operations, logical operations, and shift operations. Unit M can execute multiplication operations, as well as some arithmetic and logical operations. Unit D is in charge of memory access and process controlling, as well as some arithmetic and logical operations. Unit F carries out all the floating-point operations, including the floating-point vector operations [3].

Magnolia architecture supports both fixed-point instructions and floating-point instructions. The instruction width is 32 bits. In order to meet the ever increasing requirement of computation power of embedded applications, SIMD instructions are supported in Magnolia architecture to largely enhance the DLP (Data Level Parallelism). Special purpose instructions are also developed for acceleration of certain sensor-based embedded applications.

Magnolia architecture supports both fixed-point register file and floating-point register file. The registers in the fixed-point register file are 128 bits, while the floating-point registers are 256 bits. The number of registers in each type of register file is scalable. Both fixed-point register file and floating-point register file are programmer visible.

Traditionally, during the register allocation phase of compilation, if registers are not enough, additional save and load operations must be created and inserted into the original instruction queue by the compiler, to spill the data of a symbolic register to memory and restore it back to the register later. However, accessing memory is much slower than accessing registers and would slow down the execution speed. As in VLIW architectures, compiler needs to enhance the ILP, thus potentially increasing the pressure of register, which means spilling often happens.

Thus a mechanism called spill register file [3] is built in Magnolia architecture. When spilling happens, the data is first transferred into spill register file. Only when spill register file is full, then does the data have to be saved into memory. This mechanism can largely reduce the number of memory access incidents. Spill register file cannot be accessed by the programmer.

Figure 1 gives an instantiation of the Magnolia architecture. There are only one cluster and 2 sets of each functional unit, which means it can simultaneously execute 8 instructions in a single clock cycle. There are 64 fixed-point registers in the general register file and can be accessed by Unit A, Unit M, and Unit D. The floating-point register file

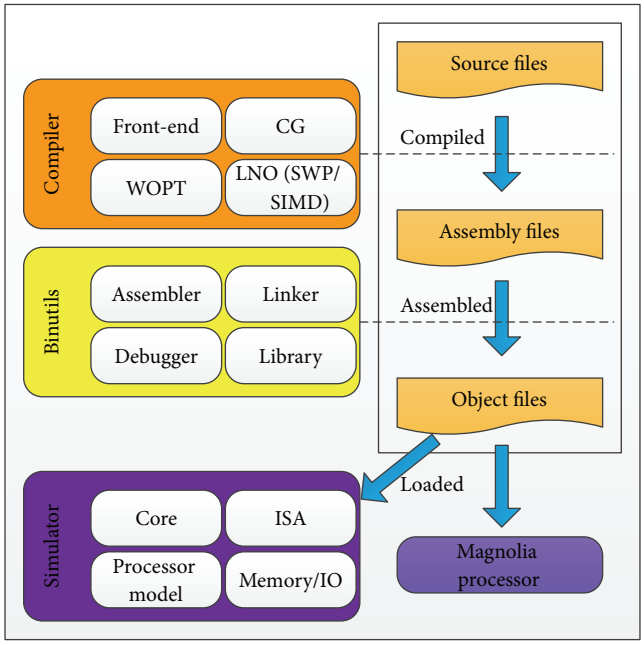

Figure 2: The software flow.

consists of 64 registers and can be accessed only by Unit D and Unit F. Unit D is responsible for the data conversion between fixed-point data and floating-point data.

3. Development of the System Toolkit

3.1. Overview of the Software Flow. The software flow is illustrated in Figure 2, which consists of compiler, assembler, linker, debugger, and simulator. Source files written with high-level languages are first compiled by the compiler into assembly files. Then the assembly files are assembled by the assembler and linked with libraries by the linker into executable files. These executable files can be run on the Magnolia VLIW DSP or on the simulator.

3.2. Compiler. The compiler for Magnolia VLIW DSP architecture is designed based on Open64 [4] originally derived from the SGI compiler, which is designed for MIPS R10000 processor, called MIPSPro. It was released under the GNU GPL in 2000 and is an open source, optimized compiler [5]. It includes many state-of-the-art optimization techniques for generating high performance codes.

In order to retarget Open64 to support compiling for Magnolia architecture, three major works must be done: (1) implementing of machine description files related to Magnolia architecture; (2) constructing of a code generator for Magnolia architecture; (3) accomplishing of optimization techniques for Magnolia architecture.

3.2.1. Implementing Machine Description Files. The retargetablity of Open64 compiler comes from the introduction of machine description files. Three main categories of information about the target architecture are described in the machine description files, including the following:

(1) Information about the Instruction Set Architecture (ISA) describes the details of instructions in the instruction set, such as functions, number of operands, data type, assembly code format, and addressing mode.

(2) Information about the Application Binary Interface (ABI) describes the details of the interface between an application program and the libraries or other parts of the application program, such as data type, data size, data alignment, and the calling convention.

(3) Information about the processor model describes the details of resources in the target architecture, such as functions of each kind of functional units and number of each kind of functional units.

So in order to support Magnolia architecture, machine description files related to Magnolia architecture must be generated.

3.2.2. Constructing Code Generator. The Open64 compiler is mainly composed of three parts: the Front-End, the Middle-End, and the Back-End.

The Front-End translates programs written in high-level language into the intermediate representation of Open64, which is WHIRL (Winning Hierarchical Intermediate Representation Language). The Front-End of Open64 supports C/C++/Fortran. The Middle-End is composed of several phases, each of which performs a target-machine-independent optimization on the WHIRL. And the Back-End is in charge of code generation, which builds assembly codes according to WHIRL.

As the Front-End is totally target-machine-free, it needs no modification to support for Magnolia architecture. And the retargeting work of the Middle-End is discussed in next section.

The Back-End of Open64 is retargeted to support code generation for Magnolia architecture. Our compiler can be roughly divided into three phases: Code Expansion, Resource Binding, and Code Emission. The details of the implementation of those three phases can be found in [3].

The Code Expansion phase analyzes the WHIRL structure and translates operations on WHIRL structure into instructions from the Magnolia architecture. So, during the implementation of this phase, two major works must be done: (1) constructing the corresponding relation between operations from WHIRL structure and instructions from machine description files of Magnolia architecture; (2) building correct Magnolia instruction format according to the machine description files.

The Resource Binding phase binds instructions to certain resources in the architecture, such as execution cycle, execution FU, and registers for operands, too. During the implementation of this phase, proper order of intersteps must be carefully arranged to avoid deadlock, as there may be conflicts caused by binding instruction to different resources. Also, in this phase, the cooperation mechanism with machine description file must be dealt with.

The Code Emission phase translates the bind instructions into assembly format. So, in this phase, correct assembly format of instructions must be extracted from the machine description files for Magnolia architecture.

3.2.3. Accomplishing Optimization Techniques. The Middle-End of Open64 is retargeted to fit for Magnolia architecture, and our compiler's Middle-End is mainly composed of two parts: loop optimizer and global optimizer. The loop optimizer performs transformation on loops to optimize the compiling code. The global optimizer uses Static Single Assignment (SSA) as the program representation and performs def-use analysis, alias classification and pointer analysis, induction variable recognition and elimination, copy propagation, dead code elimination, partial redundancy elimination, and other typical optimizations. The details of the implementation of those two optimization phases can be found in [3]. And, in this work, we focused on the implementation of the SIMD code autogeneration technique.

A lot of existing approaches in researches perform SIMD code autogeneration at a late stage of the compilation process, because more information is available at late stage. However, the disadvantage is that the data parallelism in loops cannot be effectively exploited by these techniques, so the code quality can be less optimal.

So, in this work, a high-level technique is used to generate SIMD code with examining of loop code. The SIMD code generation is in the early stage of the compilation process, just after the input source file has been transformed into the WHIRL structure. This approach only needs simple knowledge of the target-machine's ISA, so it is easily retargetable. The data packing work is done at the same time as the SIMD code is generated; thus, it can ease the work of Resource Binding, especially register allocation, in the Back-End stage.

Our SIMD code autogeneration technique is focused on loops. A preprocessing engine is introduced in the SIMD code autogeneration process. It is responsible for filtering out loops that do not suit our SIMD code autogeneration technique. Several directive rules have been introduced to choose right candidates for the following process.

After a candidate loop is selected, the compiler will go through the loop to annotate operations that could be candidates to be grouped into SIMD code. Then, all the candidates will be evaluated. Several rules are defined to conduct the evaluation process. After the evaluation, the compiler will reconstruct the WHIRL structure and replace those candidate operations in the loop into SIMD operations according to the evaluation result. Also, the data will be aligned and regrouped into packed data for the SIMD operations.

The SIMD operations in the WHIRL structure will finally be translated into SIMD instructions from Magnolia ISA in Code Expansion phase of our compiler. And the data for SIMD instructions will be prepared in Resource Binding phase.

3.3. Assembler/Linker/Debugger. The assembler, linker, and debugger of Magnolia architecture are developed based on the open source toolkit GNU Binary Utilities. There are two major issues which need to be solved:

(1) Maintaining correct instruction parallelism: according to the definition of the Magnolia assembly format, the instructions which are executed paralleled in one clock cycle must be arranged in a certain pattern where the functional units of these instructions are in an ascending order. Otherwise, the assembler cannot identify the instruction parallelism correctly. Thus, when generating assembly code, Magnolia compiler must check and rearrange the issue order of instructions, so that the information about the instruction parallelism can be delivered to the assembler in a right way. The assembler has been designed so that it can identify and recognize these pieces of information to issue correct instruction queue.

(2) Avoiding real-time errors: as Magnolia architecture is designed dedicated to embedded systems, where real-time errors must be avoided, the linker is designed to perform static linking.

3.4. Simulator. Simulator provides a platform to validate the design of software toolkit, evaluate the processor architecture, and accelerate the hardware development progress. Efficient modeling of the processor architecture and fast simulation are critical for the development of both hardware and software of VLIW architecture. The simulator for Magnolia architecture is designed based on Gem5 simulator.

Gem5 [6, 7] is an open source platform for computer system architecture research. And the simulator for Magnolia architecture is developed based on Gem5 simulator. The framework of our simulator consists of two main parts, simulation object and simulation core, and some auxiliary module. The simulator takes the application programs and the configuration script as input. The application programs are of ELF binary file format prepared by the linker. And the configuration script is coded in python language including system architecture configuration information and simulation parameters of simulated core. The simulator generates simulation report in the form of a text file including configurable trace information.

Gem5 provides plenty of simulation object models with implementation details, including memory, CPU models, bus, cache, and physical, and so forth. However, Gem5 simulator does not support the EPIC architecture processor models and VLIW ISA simulation. In order to construct our simulator, we have created a processor model with VLIW features and added the description of Magnolia ISA to the original Gem5 system. The original Gem5 loader and simulation core were also modified. The Magnolia ISA is implemented using the Gem5 ISA description system, by generating a decoder function, which analyzes the Magnolia ISA description and generates a C++ instruction object. The C++ instruction object is then treated as a basic data type of the simulator and utilized by the simulation core and other simulation objects.

The most important issue of the implementation of our simulator is to enable simulation of parallel execution of

instructions. In the original Gem5 design, instructions are processed in sequence. Thus, when processing VLIW architectures, it might lead to conflicts among the instructions operating on the same register. To avoid the conflicts, in our simulator, the register file is duplicated. The processor reads register from the original register file. And the duplicated one is used for writing. When the instructions in a dispatching packet are all processed, a register file updating function is invoked, to update all the register values, so as to maintain the coherence of the register data and thus to enable the simulation of parallel execution of instructions.

As mentioned before, Magnolia architecture uses the order of function units to indicate the instruction parallelism. If the function units of two adjacent instructions are in ascending order, then the two instructions will be issued concurrently. Or else, the issue of the latter instruction will be delayed to next cycle. The benefit is that we can save 1 bit to double the encoding space of the instruction sets. However, it is not compatible with RISC execution style. So, during the designing of our simulator, we have added instruction parallelism judgement mechanism in original Gem5 to support this feature.

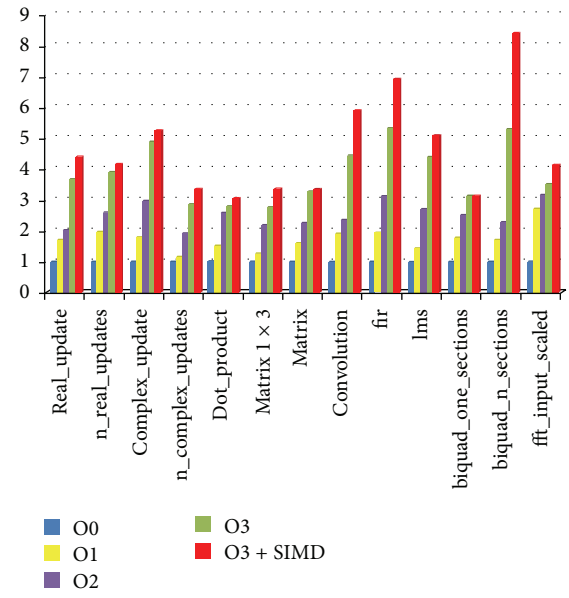

Figure 3: Performance results measured on Magnolia simulator.

4. Related Works

Chapman et al. [5] presented an interactive tool called Dragon, which provides detailed information about a C/Fortran77/Fortran90 program that may contain OpenMP/MPI constructs. It takes advantage of Open64 analysis and capabilities. The basic information displayed in Dragon's graphical tool is general-purpose and could be employed in many situations, from analyzing legacy sequential code to helping users reconstruct parallel code.

Wu et al. [8] presented methods and experiences to develop software and toolkit flows for PAC (Parallel Architecture Core) VLIW DSP processors, which is a five-way VLIW DSP processor with distributed register cluster files and multibank register architectures. The presented toolkits include compilers, assemblers, debugger, and DSP microkernels.

Chang et al. [9] presented a software framework based on Android and multicore embedded systems. In that framework, they have integrated the compiler toolkit chain for multicore programming environment which includes DSP C/C++ compilers, streaming RPC programming model, debugger, ESL simulator, and power management models.

Wittenburg et al. [10] presented architecture and software development toolkit of a parallel VLIW RISC processor, called HiPAR-DSP. They have discussed their procedure for high-level language support on that DSP.

Several works have discussed the utilization and improving of TI company's DSP starter kit, including [11–14]. There are also some works about research based on Qualcomm Company's Binary Runtime Environment for Wireless (BREW) toolkit, such as [15–17].

Steiger and Grentzinger [18] presented a platform for signal processing experiments. And they also presented an associated software toolchain to support the development of DSP applications on that hardware board. The software toolchain is custom-made and dedicated to a specific application.

Several simulators for VLIW architecture simulation have already existed, such as VLIWDLX [19], VLIW-sim [20], and Simple-VLIW [21]. Multithread and multiprocessor systems become more and more popular, thus requiring the simulator to be capable of running multithread workload on multiprocessors. However, all the simulators mentioned before can only perform single processor simulation.

Gem5 [6, 7] is an open source platform for computer system architecture research, encompassing system-level architecture as well as processor microarchitecture. Gem5 is written in C++ and python language. It has several interchangeable CPU models and can support the simulating of multiprocessor systems. Gem5 has event-driven memory systems. However, Gem5 simulator does not support the EPIC architecture processor models and VLIW ISA simulation.

5. Experimental Results

Experiments were done by running programs from DSPstone [22] benchmark. Those programs in the DSPstone benchmark suite, such as matrix, fir, lms, and fft, represent quite a broad spectrum of the possibilities of using DSP in an embedded sensor-based system. Figure 3 shows the results of performance for programs of DSPstone benchmark suite estimated on Magnolia simulator.

These programs are first compiled by the Magnolia compiler and then assembled and linked and finally loaded to the simulator to get the measurement of performance. Blue bar showed the performance generated from the compiler without any optimization. Yellow bar showed the performance generated by the compiler with some basic instruction

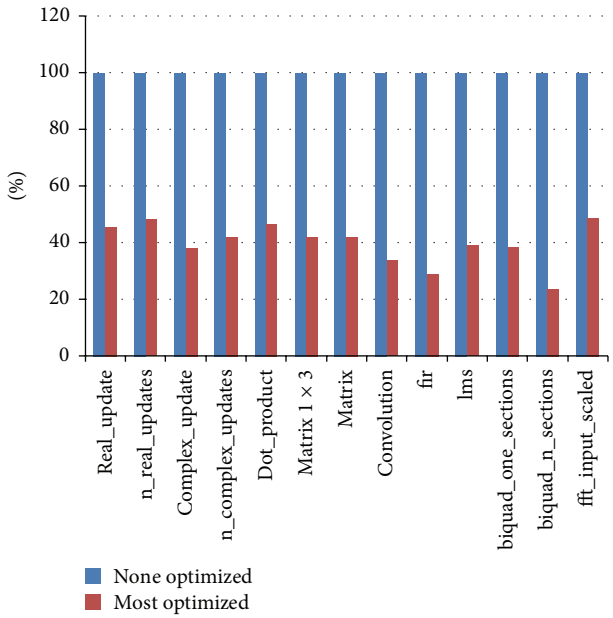

Figure 4: Energy consumption results measured on Magnolia simulator.

scheduling optimization. Purple bar showed the performance generated by the compiler with optimizations such as EBO and WOPT. Green bar showed the performance generated by the compiler with loop optimizations. Finally, red bar showed the performance generated using the compiler with SIMD code autogeneration technique.

The results indicated that our compiler can gain speedup up to around 5 times compared with the code without any optimizations. In some cases, SIMD autogeneration technique does not bring any further performance improvement. That is because, in these cases, either there is no loop in these benchmarks or the number of loop iterations does not satisfy our preprocessing rules. In that situation, our SIMD autogeneration technique does not work. In our future work, we will try to find a way to improve the applicability of our SIMD autogeneration technique, both for more levels of nested loop and for loops that have more sophisticated structures.

Figure 4 shows the results of energy consumption for programs of DSPstone benchmark suite estimated on Magnolia simulator. The energy model used in this work is based on the instruction-level energy model in [23]. In this model, the energy associated with an instruction is assumed to be dependent on its own properties as well as on its execution context, and different blocks may have different contribution factors when considering the energy consumption with the variation of number of clusters. This energy model has been validated to have an average absolute error of 1.9% and a standard deviation on the error of 5.8%, which shows a very high level of accuracy. A series of RTL-simulations have been carried out, using Cadence EDA tool chain to extract the parameters needed for construction of the energy model.

Clearly, our optimization can significantly reduce the energy consumption related to the execution of those programs, thus making it more suitable for sensor-based application usage.

6. Conclusion

In this paper, we have presented our methods and experiences on developing system toolkit for a VLIW DSP architecture. Our entire system toolkit includes compiler, assembler, linker, debugger, and simulator. We presented our methods to develop those tools and also presented the experiences of dealing with issues encountered in the process. Results evaluated using DSPstone benchmarks indicated significant optimization on performance and energy consumption. The experiences presented in this paper might benefit the architecture designers and toolkit developers who are interested in similar VLIW DSP architectures.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This paper is supported by the National Natural Science Foundation of China (no. 61201182).

References

[1] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *Proceedings of the 10th Annual International Symposium on Computer Architecture (ICSA '83)*, pp. 140–150, Stockholm, Sweden, June 1983.

[2] B. C. Schafer, Y. H. Lee, and T. Kim, "Temperature-aware compilation for VLIWProcessors," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '07)*, pp. 426–431, IEEE, Daegu, Republic of Korea, August 2007.

[3] X. Yang and H. He, "An advanced compiler designed for a VLIW DSP for sensors-based systems," *Sensors*, vol. 12, no. 4, pp. 4466–4478, 2012.

[4] 2012, http://www.open64.net/.

[5] B. Chapman, O. Hernandez, L. Huang et al., "Dragon: an Open64-based interactive program analysis tool for large applications," in *Proceedings of the Parallel and Distributed Computing, Applications and Technologies (PDCAT '03)*, pp. 792–796, August 2003.

[6] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006.

[7] 2012, http://gem5.org/Main_Page.

[8] C. Wu, K.-Y. Hsieh, Y.-C. Lin et al., "Integrating compiler and system toolkit flow for embedded VLIW DSP processors," in *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '06)*, pp. 215–222, August 2006.

[9] Y.-H. Chang, C.-B. Kuan, C.-Y. Lin et al., "Support of software framework for embedded multi-core systems with android

environments," in *Proceedings of the 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia '11)*, pp. 2–8, IEEE, Taipei, Taiwan, October 2011.

[10] J. P. Wittenburg, M. Ohmacht, J. Kneip, W. Hinrichs, and P. Pirsch, "HiPAR-DSP: a parallel VLIW RISC processor for real time image processing applications," in *Proceedings of the 3rd International Conference on Algorithms and Architectures for Parallel Processing (ICAPP '97)*, vol. 97, pp. 155–162, Melbourne, VIC, Australia, December 1997.

[11] A. Z. R. Langi, "Rapid prototyping of DSP systems using DSP starter kit," in *Proceedings of the Joint International Conference on Rural Information & Communication Technology and Electric-Vehicle Technology (rICT & ICeV-T '13)*, vol. , no, pp. 1–5, November 2013.

[12] M. D. Galanis, A. Papazacharias, and E. Zigouris, "A DSP course for real-time systems design and implementation based on the TMS320C6211 DSK," in *Proceedings of the 14th International Conference on Digital Signal Processing (DSP '02)*, vol. 2, pp. 853–856, IEEE, 2002.

[13] M. G. Morrow, T. B. Welch, and C. H. G. Wright, "A host port interface board to enhance the TMS320C6713 DSK," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '06)*, vol. 2, p. 2, Toulouse, France, May 2006.

[14] Texas Instruments, *TMS320C64x Technical Overview*, Texas Instruments, 2000.

[15] S. Zhou, Z. Liu, and Y. Fan, "The study of UIONE phone developing technology based on BREW platform," in *Proceedings of the International Asia Symposium on Intelligent Interaction and Affective Computing (ASIA '09)*, pp. 130–132, December 2009.

[16] Z. Li, W. Jin, D. F. David, and S. Lansun, "Wireless media streaming system over CDMA networks," in *Proceedings of the 9th International Conference on Advanced Communication Technology*, vol. 3, pp. 2226–2230, February 2007.

[17] Z. He, Y. Teng, and Q. Dou, "Design and development of meteorological information inquiry system based on BREW," in *Proceedings of the International Forum on Information Technology and Applications (IFITA '10)*, vol. 2, pp. 34–37, IEEE, Kunming, China, July 2010.

[18] O. Steiger and B. Grentzinger, "A generic, cheap and portable platform for signal processing experiments," in *Proceedings of the IEEE International Conference on Industrial Technology (ICIT '12)*, pp. 1009–1013, March 2012.

[19] M. Bečvářn and S. Kahánek, "VLIW-DLX simulator for educational purposes," in *Proceedings of the Workshop on Computer Architecture Education (WCAE '07)*, pp. 8–13, ACM, San Diego, Calif, USA, June 2007.

[20] I. Barbieri, M. Bariani, A. Cabitto, and M. Raggio, "Multimedia-application-driven instruction set architecture simulation," in *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME '02)*, pp. 169–172, August 2002.

[21] P. Wang, J. X. Yang, and B. Wang, "Simple-VLIW: a fundamental VLIW architectural simulation platform," in *Proceedings of the 7th International Conference on System Simulation and Scientific Computing (SSSC '08)*, pp. 1258–1266, Beijing, China, October 2008.

[22] V. Zivojnovic, J. M. Velarde, C. Schlager, and H. Meyr, "DSPstone: a DSP-oriented benchmarking methodology," in *Proceedings of the International Conference on Signal Processing Applications and Technology (SPAT '94)*, pp. 715–720, October 1994.

[23] A. Bona, M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon, "Reducing the complexity of instruction-level power models for VLIW processors," *Design Automation for Embedded Systems*, vol. 10, no. 1, pp. 49–67, 2005.