Research Article

Effort at Constructing Big Data Sensor Networks for Monitoring Greenhouse Gas Emission

Haijing Tang, Xu Yang, and Yanjun Zhang

1 School of Software, Beijing Institute of Technology, Beijing 100081, China
2 School of Information and Electronics, Beijing Institute of Technology, Beijing 100081, China

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

Received 13 March 2014; Accepted 24 June 2014; Published 7 July 2014

Academic Editor: Jesús Carretero

Copyright © 2014 Haijing Tang et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Global warming is an important environment issue rapidly becoming a part of popular culture. Scientists are 95–100% certain that it is primarily caused by increasing concentrations of greenhouse gases produced by human activities. Policies dealing with global warming depend on accurate and precise monitoring of the emissions of greenhouse gases. However, most greenhouse gas emissions management comes from bottom-up reporting, in which a company reports its energy use and other resource consumption metrics and maps out its carbon footprint. This is not a sufficient good way. A big data network of sensors has been constructed by us in Neimenggu province of China, which will gather huge amounts of more accurate information on the dynamics of greenhouse gases in local areas and measure changes over time. Application specific instruction set processor is integrated in those sensor devices, to perform preliminary process of collected data. In this work, we have presented our efforts at constructing the big data sensor network for monitoring the greenhouse gases emission and mainly focused on building the ASIP’s retargetable compiler framework designed and dedicated for our sensor network.

1. Introduction

Since the early 20th century, Earth’s mean surface temperature has increased by about 0.8°C (1.4°F), with about two-thirds of the increase occurring since 1980 [1]. Global warming is an important environment issue which is rapidly becoming a part of popular culture [2]. Warming of the climate system is unequivocal, and scientists are 95–100% certain that it is primarily caused by increasing concentrations of greenhouse gases produced by human activities such as the burning of fossil fuels and deforestation. These findings are recognized by the national science academies of all major industrialized nations.

Proposed policy responses to global warming include mitigation by emissions reduction, adaptation to its effects, and possible future geoengineering. Most countries are parties to the United Nations Framework Convention on Climate Change (UNFCCC), whose ultimate objective is to prevent dangerous anthropogenic (i.e., human-induced) climate change.

Parties to the UNFCCC have adopted a range of policies designed to reduce greenhouse gas emissions and to assist in adaptation to global warming. They have also agreed that deep cuts in emissions are required and that future global warming should be limited to below 2.0°C (3.6°F) relative to the preindustrial level. However, reports published in 2011 by the United Nations Environment Programme and the International Energy Agency suggest that efforts as of the early 21st century to reduce emissions may be inadequate to meet the UNFCCC’s 2°C target.

China has been one of the earliest members of UNFCCC since 1992. China’s greenhouse gas emissions are growing rapidly and, even with policies adopted by China, are expected to rise until at least 2030. The emissions growth is driven by China’s rapid economic and industrial growth and its reliance on fossil fuels despite measures to raise the shares of non-fossil energy sources. In response to the UNFCCC process investigating the technical issues surrounding the ability to reduce greenhouse gas emissions in developing countries, over the past two decades, strong government
directives and investments have dramatically reduced the energy and greenhouse gas intensities of China’s economy [3].

Policies dealing with global warming depend on accurate and precise monitoring of the emissions of greenhouse gases [4, 5]. However, most greenhouse gas emissions management comes from bottom-up reporting, in which a company reports its energy use and other resource consumption metrics and maps out its carbon footprint. This is not a sufficient good way to monitor the greenhouse gases emission.

So it is essential to construct a network of sensors to monitor the emissions of greenhouse gases. Thus it can provide another layer of top-down data that can confirm, supplement, or contradict reporting from companies.

We have constructed a big data network of sensors in Neimenggu province of China. This network of sensors will gather huge amount of more accurate information on the dynamics of greenhouse gases in local areas and measure changes over time. Two hundred sensors will be placed and all of the greenhouse gas monitoring sensor equipment is designed by us.

This network will help us to collect increasing amounts of data from around the grasslands and forest in Neimenggu province. Such a province level greenhouse gas network would collect massive amounts of real-time emissions data about atmospheric carbon dioxide and methane emissions (greenhouse gases that are contributing to climate change) from locations throughout Neimenggu. The data would enable us to monitor specific areas that are contributing a high amount of emissions and watch those areas over time for growth or improvement and will enable us to further refine our scientific studies on where greenhouse gases originate, how they circulate around our atmosphere, and how they move from one area to another.

The sensor stations will be linked via data connections and the data will be processed preliminarily on those sensor devices and then sent to the data center, so researchers can use tools to further analyze the data and draw trends and information from it. Application Specific Instruction Set Processor (ASIP) is integrated in the sensor device, to perform preliminary process of collected data.

In this work, we have presented our efforts at constructing the big data sensor network for monitoring the greenhouse gases emission and mainly focused on building the ASIP’s retargetable compiler framework designed and dedicated for our sensor network, which is based on the open resource compiler (ORC).

The following of this paper is organized as follows: Section 2 will discuss some background content; in Section 3, the architecture of our ASIP is described, while the detail of our retargetable compiler framework is presented in Section 4; Section 5 talks about related works, while experimental result is shown in Section 6; and finally the conclusion is given in Section 7.

2. Background

2.1. Application Specific Instruction Set Processor. An Application Specific Instruction Set Processor (ASIP) is a processor designed for one particular application, or a set of specific applications. An ASIP exploits characteristics of various pieces of hardware to meet the desired performance, cost, and power requirements of specific applications. ASIPs are a balanced solution between two extremes: Application Specific Integrated Circuits (ASICs) and General Programmable Processors [6–8]. Since an ASIC is specifically designed for one behavior, it is difficult to make any change at a later stage. In such a situation, the ASIPs offer the required flexibility at lower cost than General Programmable Processors [9].

According to Liem et al. [9], generally, ASIP design can be divided into five main steps as follows (Figure 1 [9]).

1. Application Analysis. Applications are analyzed to get the desired characteristics/requirements which can guide the hardware synthesis as well as instruction set generation. Analyzed information is stored in some suitable intermediate format for usage in the subsequent steps.

2. Architectural Design Space Exploration. Performance of possible architectures is estimated, using output of step 1 as well as the given design constraints, and suitable architecture satisfying performance and power constraints and having minimum hardware cost is selected.

3. Instruction Set Generation. Instruction set is generated for that particular application and for the architecture selected. This will be used during the code synthesis and hardware synthesis steps.

4. Code Synthesis. Compiler generator or retargetable code generator is used to synthesis code for the particular application or for a set of applications.

5. Hardware Synthesis. Hardware is synthesized using the ASIP architectural template and instruction set architecture starting from a description in VHDL/VERILOG using standard tools.

2.2. Open Resource Compiler (ORC). Open resource compiler is developed initially for Itanium architecture. It uses WHIRL as intermediate language, and the using of WHIRL enables ORC to support different high-level languages and ASIP targets. The frontend of ORC translates different kinds of high-level language into WHIRL and the backend of ORC processes WHIRL to generate target codes. The using of WHIRL and the architecture of ORC make it possible that only the backend of ORC needs to be retargeted when porting ORC to an ASIP architecture. However, the backend of ORC is still very sophisticated and is comprised of several phases including code generator (CG) and some optimization components, such as interprocedural analyzer and optimizer (IPA), loop-nest optimizer (LNO), and global scalar optimizer (WOPT).

Compilation can be viewed as a process of gradual transition from the high-level language constructs to the low-level machine instructions, so WHIRL has several levels, as shown in Figure 2. The closer it is to the source high-level language, the higher its level is. And the more it resembles the
machine instructions, the lower its level is. When compiling moves on, WHIRL is down in its level. In WHIRL, source codes are organized as a tree, after being processed by the frontend of ORC. And every node of this tree is called an “op,” which represents an operation. Every op has some fields that store the information of that op, such as operate code, and operand. Several ops can be assembled into a “bb,” which represents a basic block that is a combination of several operations. Several “bbs” can be assembled into a “region,” which is a hyperform of several basic blocks.

ORC has some predefined optimization levels, named as O0, O1, O2, and so on. O0 level is the lowest level. Under this level, ORC only goes through the basic phases when compiling source code, such as code selection, local schedule, local register allocation, and code emission, and has no optimization. In O1 and O2 levels, some optimizations like LNO, IPA, and WOPT are included in the compilation process to make the compilation more effective.

3. Architecture of the Designed ASIP

A Register-File Connected Clustered ASIP architecture has been developed by us, to be used in our sensor devices. And a retargetable compiler framework has been generated based on ORC, to support this new ASIP architecture.

One example of this ASIP architecture is shown in Figure 3. There are eight function units, which are grouped into four clusters. Each cluster has its own local register file which is named A, B, C, and D, respectively [10]. And all clusters share a global register file named G. Both the local register file and the global register file contain 16 registers with width of 32 bits [10]. In this architecture, each function unit can only access its corresponding local register file through its own access port. However, all the function units can access the global register file, and so comes the name Register-File Connected. And the two function units in the same cluster can access the global register file by different access ports [10].

There are four types of function units in this architecture. They are arithmetic and logic unit (AL), arithmetic and branch unit (AB), multiplication unit (MP), and load and store unit (LS). Function unit AL can be used to perform most of the arithmetic and logic operations; function unit AB is mainly used to execute branch operations and it can also perform some arithmetic operations; function unit MP can only be used to perform multiplication operations; and the main function of LS is to load/store data from/to memory. In order to achieve high performance, each type of function unit is duplicated in this architecture [10].

The main goal of grouping these function units is to reduce the communication between different clusters; that is, it is required that the results of one function unit are mostly consumed by the other function unit in the same cluster [10].

4. The Retargetable Compiler Framework

ASIP cannot work well without compiler’s support and compiler needs to know the necessary hardware information to work well. However, ASIP architectures can be various. If the designer has to develop compiler for each different ASIP architecture, the work amount will be huge. So, retargetable compiler is the inevitable choice.

ORC has some advantages to be a footstone of a retargetable compiler. In ORC, all the hardware information is grouped in the machine description files, including operate code information, operate attribute information, register information, operand information, and function unit information. So, when porting ORC to different hardware architecture, only the hardware information in the machine description files needs to be changed.

We have modified the information about the function units, the information about the registers, the information about the operation, the information about the operand types in this architecture, and some other information, in the machine description files, to make ORC suit our ASIP architecture.

In the machine description files, the registers in our ASIP architecture are classified into five register subclasses under the integer register class defined initially in ORC. Also, the attributes of each kind of register have to be defined. For example, we defined 2 registers of attribute “callee” and 12 registers of attribute “caller” in both local register file and global register file.

The flow of the backend of ORC under optimization level O0 only is composed of the code generation part which includes 4 main phases, which are code selection, local schedule, local register allocation, and code emission, respectively.
In the code selection phase, WHIRL is converted into operation list. And this operation list is scheduled in the local schedule phase. Registers for all the operations are allocated in local register allocation phase. Finally the assemble code is generated in code emission phase.

The original code of code selection phase is modified to add the operations from our ASIP architecture and the algorithm of local schedule is also modified to improve the utilities of local register files too.

For Itanium architecture, when performing register allocation, at first, ORC will select a register node. Then the register set that can be used to this node will be evaluated and registers in the available set are searched one by one to find the suitable one that has not been assigned to any node. When a suitable register is found, it will be allocated to the register node.

However, our ASIP architecture is different from Itanium architecture. In Itanium architecture, every function unit can access any register. While, in our ASIP architecture, each function unit can only access its corresponding local register file and the public global register file, which means that it is necessary to know which function unit executes the register node in the register allocation phase. In the original ORC compiler, the function unit is not necessary information when compiling, so a new field needs to be added to note this information.

In the local schedule phase, this new added field of every op is assigned a number which represents its executing function unit. In the register allocation phase, the registers that an op can use can be decided according to this field.

When dealing with the register allocation phase, there are two more problems also needed to be considered. One came out during the process of finding the can-use register set for a register node. Because a “bb” is regarded as an atom element for processing in the register allocation phase and the ops in the “bb” are processed in the reversed order, in ORC, all ops involved in one register node’s live-range need to be considered when allocating register for this register node. Here, the live-range represents the range from the location where a node is defined at the first time to the location where it is used for the last time.

However, considering the characteristic of our ASIP architecture, this will limit the range of registers that can be used to only global register file in most cases. So the ops are considered only when they treated the register node as source operand or destination operand when allocating register.

Another problem appeared when there were not enough registers for allocation. In general, when register allocation encounters this situation, compilers often automatically spill the value into memory. The spill method will also add one op to store the value into memory after it is used and add one op to load it from memory before it is used in some time later. However, in our situation, the information about the executing function unit of the spill op, both the store one and the load one, needs to be known and then the right register
set for these ops can be found out. Otherwise, there will be a mistake, and harm to the performance is done.

A new trace algorithm is improved to guide the trace through the can-use register set to find a free register that can be allocated to a register node after the can-use register set of this node is given. According to the function unit that executes the register node, a start location for trace can be given; then a trace can be carried on. This algorithm can largely improve the efficiency of finding a register from the can-use register set for a target register node.

At last, the code of code emission is modified to get assemble code in the format defined by our ASIP architecture, so it can be used on our assembler.

5. Related Work

Clustering has become a common trend in processor designing; there has already been a lot of work concerning the compiling of clustered architectures.

Codina et al. [11] have exploited a concept of virtual cluster to assist the instruction scheduling for clustered architecture. Aleta et al. [12] have presented a graph-based approach, which is named AGAMOS, to schedule loops on clustered architectures, which can reduce the number of intercluster communications. Arafath and Ajayan [13] have implemented a modified list scheduling algorithm using the amount of clock cycles followed by each instruction and the number of successors of an instruction to prioritize the instructions, to perform an integrated instruction partitioning and scheduling for clustered architectures.

Zalamea et al. [14] have focused on integrating register spill with instruction scheduling for clustered architecture, to provide additional possibilities for obtaining high throughput schedules with low spill code requirements. Huang et al. [15] have presented a worst-case-execution-time-aware rescheduling register allocation (WRRA) approach, for clustered architecture. The novelty of this work is that the WCET problem and the phase ordering problem of the clustered architecture are taken into account jointly.

6. Experimental Result

As described earlier, our architecture has five register files, including four local register files and one global register file. All function units can access the global register file, but they can only access their corresponding local register file. And all the register files have 16 registers with width of 32 bits [10].

In order to see the influence of register number on the performance and to find out the optimal proportion between the local register files and global register files, some experiments were done. First, the numbers of registers of the global register file and local register file were changed, keeping the total number of registers in our architecture constant. Then, the numbers of all the register files were reduced to make a comparison. The results are illustrated in Figure 4. In situation A, all the register files have 16 registers. In situation B, there are 8 registers in every local register files and 48 registers in the global register file. In situation C, each register file has 8 registers.

In this figure, y-axis represents the execution cycles needed for processing the benchmark [16]. The data had been pretreated to be clearly displayed in one picture. There is not much difference between situation A and situation B for all the benchmarks. This is mainly because the utilities of local register files are lower than the global register file. Therefore the reduction of the number of local registers does little influence on the performance.
But obviously the reduction of the number of global registers has considerable impact on the performance, as compared to the situation C and the two others.

So, if the number of registers is increased, the performance improved, but the trend will flatten out when the number of registers becomes large.

7. Conclusion

In this paper, we have presented our effort at building a retargetable compiler framework of an ASIP architecture dedicated for construction of a big data sensor network for monitoring the greenhouse gases emission. This retargetable compiler framework is built based on ORC.

Some research has been done to find out the influence of register number on the performance. The result shows that the more the register number is, the higher the performance can get. But when the register number grows larger, the performance growth will flatten out. And since the utilities of local register files are less than the global register file, the influence of local register number is less than the global register file.

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] A. Carnesale and W. Chameides, Americas Climate Choices, The National Academies Press, Washington, DC, USA, 2011.

[2] M. L. Deaton and J. J. Winebrake, “Greenhouse gases and global warming,” in Dynamic Modeling of Environmental Systems, pp. 158–173, Springer, New York, NY, USA, 2000.

[3] J. A. Leggett, J. S. Logan, and A. Mackey, China’s Greenhouse Gas Emissions and Mitigation Policies, Congressional Research Service, Library of Congress, 2008.

[4] R. S. Eckaus, “Comparing the effects of greenhouse gas emissions on global warming,” The Energy Journal, vol. 13, no. 1, pp. 25–36, 1992.

[5] R. DeFries, F. Achard, S. Brown et al., “Earth observations for estimating greenhouse gas emissions from deforestation in developing countries,” Environmental Science and Policy, vol. 10, no. 4, pp. 385–394, 2007.

[6] J. R. Goodman and W-C. Hsu, "Code scheduling and register allocation in large basic blocks," in Proceedings of the International Conference on Supercomputing, pp. 442–452, St.Malo, France, July 1988.

[7] S. Novack and A. Nicolau, “Mutation scheduling: a unified approach to compiling for fine-grain parallelism,” in Proceedings of the of 7th Workshop on Languages and Compilers for Parallel Computing, vol. 892 of Lecture Notes in Computer Science, pp. 16–30, 1994.

[8] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, “Importance of prepass code scheduling for superscalar and superpipelined processors,” IEEE Transactions on Computers, vol. 44, no. 3, pp. 353–370, 1995.

[9] C. Liem, T. May, and P. Paulin, “Instruction-set matching and selection for DSP and ASIP code generation,” in Proceedings of the European Design and Test Conference (EURODAC’94), pp. 31–37, Paris, France, March 1994.

[10] Y. J. Zhang, H. Hu, and Y. H. Sun, “A new register file access architecture for software pipelining in VLIW processors,” in Proceedings of the ASP-DAC Asia and South Pacific Design Automation Conference, vol. 1, pp. 627–630, January 2005.

[11] J. M. Codina, J. Sánchez, and A. González, “Virtual cluster scheduling through the scheduling graph?” in Proceedings of the International Symposium on Code Generation and Optimization (CGO ’07), pp. 99–101, San Jose, Calif, USA, March 2007.

[12] A. Aleta, J. M. Codina, J. Sanchez, and A. Gonzalez, “Graphpartitioning based instruction scheduling for clustered processors,” in Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture (MICRO ’01), pp. 150–159, December 2001.

[13] K. M. I. Y. Arafath and K. K. Ajayan, “A novel instruction scheduling scheme for clustered VLIW architecture,” in Proceedings of the IEEE Recent Advances in Intelligent Computational Systems (RAICS ’11), pp. 783–787, Trivandrum, India, September 2011.

[14] J. Zalamea, J. Lloso, E. Ayguadé, and M. Valero, “Modulo scheduling with integrated register spilling for clustered VLIW architectures,” in Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO-34 ’01), pp. 160–169, Washington, DC, USA, December 2001.

[15] Y. Huang, M. Zhao, and C. J. Xue, “WCET-aware re-scheduling register allocation for real-time embedded systems with clustered VLIW architecture,” in Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages,
Compilers, Tools and Theory for Embedded Systems (LCTES’12), pp. 31–40, ACM, June 2012.

[16] C. Lee and M. Stoodley, “UTDSP BenchMark Suite,” 2012, http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html.
Submit your manuscripts at http://www.hindawi.com