Data-level Parallelism Implementation and Optimization in Multiple Aspects on Binary Translation with Multi-processors

Ting-Wei SU
Texas A&M University
College Station, United States
willytwsu@tamu.edu

ABSTRACT

Recently, Apple Inc. published the novel M1 CPU based on ARM structure. This epoch-making product amazed the industry by its incredible performance and energy efficiency. Meanwhile, to resolve the cross-architectural problem between reduced instruction set computer (RISC) and complex instruction set computer (CISC), they proposed a binary translator that enabled software running on both RISC and CISC. However, the performance loss caused by binary translation is unavoidable. How to decrease the loss and accelerate the translation procedure grow their importance nowadays.

In case to facilitate binary translation, various techniques are revealed, e.g. dynamic binary translation (DBT), code cache, and extended single instruction multiple data (SIMD) exploitation. Among these strategies, data-level parallelism (DLP) plays a vital role on whether the compiler and machine can perfectly utilize data stream (including assembly sequences, basic blocks, fetched information from sliding window) while translation. For the sake that binary translation has strict restrictions to verify semantically equivalent, DLP is the most efficient method to enhance performance without enormous hardware adjustment. Hence, we will discuss about DLP implementation and optimization in various domains to achieve the improvement in binary translation.

1 INTRODUCTION

Binary translation is the mechanic to address cross-architectural compatibility problems. Different architectures, e.g. ARM and Intel x86, have plenty of discrepancies in their assembly code. **Fig. 1** exhibits the discrepancies of assembly code from an identical source code. The most advanced machine today still can not decode these two kinds of computers simultaneously. The main function of binary translation is to translate the binary code from the guest processor to the host processor without recompilation the source code. During the processing of binary translation, the engine could dynamically alter parallelism through instruction level or data level.

Exploit DLP on binary translation not only leverages the translation completion, but also reduces the power consumption. To better reinforce the performance without substantial hardware adjustment, we focus our survey on the framework and algorithm optimizations. Sliding windows and code caches are universal strategies people dedicating to. These two additional frameworks augment the efficiency of translation. The fetched instructions are split into a plethora of basic blocks, which are viewed as a processing unit in sliding windows and code caches. The basic blocks are then stored in history tables and produces unique tags. Some papers propose to form basic blocks into pairs to better exploit spatial locality[1]. Once the repeated translation sequences happens, machine can fetch the instructions by tags and reduces the time searching pages in main memory.

Algorithm optimizations can be achieved by weighted edge scheduling (WES)[2] or task arrangement (TA)[3]. The data dependencies and number of tasks are first recorded through sliding windows as profiles. Afterwards, WES dynamically align the edges in a graph and assigns weights and dependencies on them based on profiles. The connections between edges are then calculated with optimal workload and performance. Task arrangement examines the time each data processing costs at first. Machine dynamically schedules the tasks to have the least execution time.

Besides these conventional improvement methods, techniques employing machine learning (ML) related algorithms will be introduced in our survey. They bring a prospective aspect dealing with binary translation although the results are not good enough to be implemented on industry products. Following their steps can we find some possible achievements that enables ML be applied on most binary translation issues.

Our survey firstly introduces the methods for information fetching before data processing. Details about data-level parallelism methods are discussed in section 3. Binary translation mechanics including both statically and dynamically are included in section 4. Our points of view are demonstrated in section 5. We propose some possible future works in the last part.

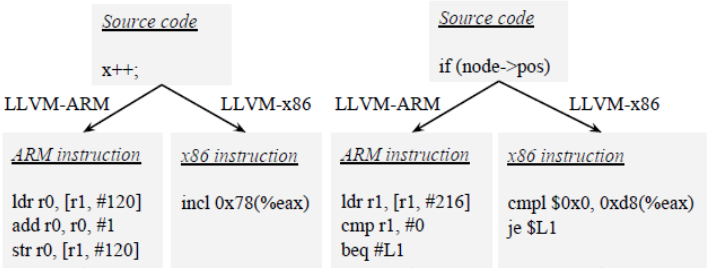

Fig. 1 ARM and x86 instructions are compiled through the same LLVM compiler. These two architecture apply totally different methods to have functionalized instructions. The discrepancies from the architectures make it difficult to validate the semantical equivalence after translation.

2 INFORMATION FETCHING BEFORE DATA PROCESSING

There is a critical task before translation is to fetch the information about the assembly code and its following instructions.

Code Cache. [4] suggests an approach applying code caches to leverage the translation efficiency. It disassembles the assembly

codes from guest machine into intermediate translation (IT) sequences. Then, putting IT sequences into the code caches to benefit following translations. Through memory flushing, this architecture enables recursive and bilateral translation between machines. In this fundamentals, [5] proposes pipelining structure with fibers, which are mostly employed in the device I/Os. Fibers excel at continuously data fetching. This characteristic augments the reading speed between BT processing units and code caches. In addition, pipelining increases the throughput of codes while the bandwidth is enough.

Sliding Window. People typically utilize sliding window (SW) to record data dependencies and properties. [6] identify three possible sliding window parallelism, inter-key, inter-window, and intra-window to enhance throughput. Inter-key and inter-window insert items into threads with replicas of key or window. Intra-window splits the items into small pieces and realigns them into threads, which best exploits the locality and balances the workload. Data accessed by SW can further compose a set of information named basic block[7]. Some researchers further constitute basic blocks to vectors[8] and super-words[9] to implement parallelism on powerful computers.

Data Partitioning. [10] transfers basic block into profile. This profile is viewed as a standard for DLP operations. There are number of tasks estimates and cache fetching references recorded in the profile. Machine determines which data combines together for better performance according to the profile. For example, data x1 and data x2 fetch the same cache lines but they are not adjacent instructions. Machine will attempt to assign them together lest repeated cache fetching and reduce overhead. Except for performance improvement, profile strengthen the classification of semantical equivalence between guest and host codes. To validate whether the translation is successful is incredibly difficult because each computer structure has its own assembly logics. Profile provides the traceable table that verifies the correctness of translation procedure.

From above-mentioned papers, intra-window sliding window responsible for profiles alongside with pipelining code caches are the ideal strategy to provide necessary information for binary translation afterwards. However, bandwidth and timing are two vital problems. Without substantial hardware adjustment, original bandwidth is not capable of handling additional pipeline stages with fibers. Binary translation usually executes millions of sequences at a time. It is convincing that intra-window SW can produce enough profiles in a couple cycles. It costs significant time when encountering irregular instructions (non-common instructions).

There is the trade-off between real situations and ideal performance. In our opinion, intra-window sliding window with partial profile (without task estimates) is the most suitable solution in computers nowadays. Intra-window requires several cycles to rearrange itself and achieve significant improvement. Partial profile keeps the traceable tables and data dependencies, which still reinforces the validation of semantical equivalence. Pipelining intra-window SW is a possible method to accomplish, but it demands efforts to inspect the results.

3 DATA-LEVEL PARALLELISM STRATEGIES IN MULTIPLE PERSPECTIVES

After collecting necessary information, we can now start with data-level parallelism operations. Scheduling optimization related to DLP has two main domains, task arrangement (TA)[3] and weighted edge scheduling (WES)[2].

Scheduling Optimization. Both TA and WES require profile (mentioned in section 3) to conduct following research. TA needs the execution time of each task and the current workload of processors. Machine dynamically schedule the tasks depending on the least execution time and workload. WES demands for data dependencies (relations between edges) and number of task estimates. Utilizing data dependencies reduces the overhead caused by cache fetching. Same as TA, WES decides which task works on which processor to balance the workload. **Fig. 2** displays the final graph after WES. Processing elements (PE) have similar loading through dependency and number of tasks evaluation.

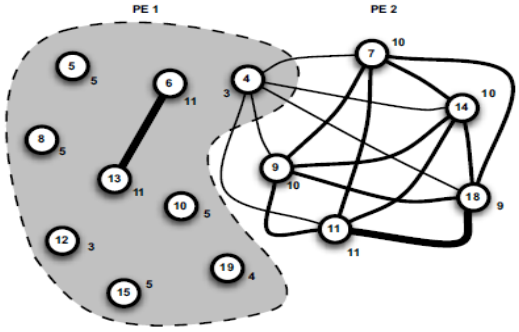

Fig. 2 Number nearby the edges are the amount of tasks when executing this edge. The thickness of connections exhibit the dependencies. Edges have low dependencies but high workload will be assigned to relatively idle PE. By this strategy, each PE can work on roughly the same task number and execution time.

Adaptive DLP. Except for scheduling optimization, [11] proposes a method applying different DLP approaches in specific situations. Although this paper mainly focuses on the kernel utilization, we can treat binary translation processing elements as enormous kernels inside. Data partitioning can be done according to the properties of basic blocks. If the basic blocks are comprised of small items, machine tends to adopt kernel partitioning to fit with the instructions. In contrary, basic blocks with big items are appropriate for inter-kernel strategy.

Kernel Improvement. [12] introduces a solution efficiently applying kernel computation. It duplicates the kernels with potential following computations (multiplication). Machine operates the sequential computation in parallel that decreases the time spending on recursive computations. The additional computation cost is acceptable in exchange of performance reinforcement. [11] also includes improvements about kernels. It demonstrates how to operate kernel partitioning. Small pieces of kernels help accelerate data fetching and computations due to shorter distance between PE and targets.

SIMD Exploitation. With the support of SIMD engine, such as ARM NEON and Intel AVX, the main processor could utilize

vectorized code regions to realize DLP. Along with the dynamic Assembler we could extend the DLP to a dynamically detection at the run-time execution. The approach could be achieve by vectorizing the instruction set architecture (ISA) or convert the short instruction into longer instruction.

Previously-mentioned methods are seemingly solutions to deal with assembly sequences. Machine should pay extra cycles to complete their preliminary works. This cause them not capable of universally adopted. In spite that there are approaches available to address the loss in translation, they do not have a in-depth research throughout the whole computer architecture. Current papers simulate the data stream without taking penalty and exceptions into account. They have deficiency though, we can still figure out more and more possibilities following their directions. Notable advances will be made if the algorithm of DLP is optimized and avoid exceptional hardware efforts.

4 MECHANICS WHILE EXECUTING BINARY TRANSLATION

Binary Translation Improvement. A Hybrid Multi-Target Binary Translator (HMTBT) was presented in [13] proposing the architecture of dynamically selecting the best performance before executing the Instruction-level Parallelism (ILP) engine or Data-level Parallelism (DLP) engine on the chosen target accelerator would create the adaptable solution for different application. ILP is exploited by using a CGRA engine through the process of (A) Dependency Analysis (B) Mapping (C) Configuration Build. DLP is exploited using the ARM NEON engine to operate the process of the binary translation on loop detection process when the iteration is unknown during compile time. After the ILP and DLP optimization of the code region (Mutual Regions), a dispather will receive CGRA configuration and SIMD instruction to decide which acceleration is dispatched in the Translation Cache. In addition, a History Table is used to stored the optimized code region for future use.

In [14], researcher also proposes a dynamic SIMD Assembler (DSA) generating the SIMD instruction to trigger the ARM NEON engine by detecting the vectorized code regions. The need of the DSA to detect the vectorized loop run-time is to compensated the weakness of the ARM auto-vectorization compiler count loop detection at compile time. Such condition as dynamic range loop, conditional loop and loop with a function call are three condition that the compiler couldn't possibly has optimized; thus, it will required the assistance from the run-time engine to parallelize the execution.

Combining the result in [14] and [13] both utilizing ARM NEON DLP engine to accomplish DLP, we could see that NEON couldn't have became effective without the co-existence of CGRA ILP engine. The effect of ARM NEON is only limited to certain scanerio and the performance enhancement is not as outstanding than the CGRA engine along. The CGRA engine could have operated in certain scenario with certain amount of optimization, but with the help of ARM NEON, the result could reach optimal in most of the scenario. That is to say, the combination of both of the ILP and DLP could create an effect better than both operates individually combined.

Different from [14] and [13], [15] proposed the binary translation between the AVX ISA, with 256-bit register wide, and the x86 SSE ISA, with 128-bit register wide, could become beneficial to software developer by eliminating the migration effort from older architecture to newer architecture. The paper proposed to use the static loop detection mechanism at compile time to translate the ISA to a lower-cost processor which create a future opportunity for software programmer to effortlessly migrate the old version application to the new version architecture when the technology would have dramatically improved.

An additional case has been taken into account when the loop exist the loop-carried dependency stating that such dependency will limit the parallelism factor for the program. Consider the following case

```
for (i = 0; i < 400;i++)
  a[i+4] = a[i] + 2;
```

The dependency will limit the maximum parallelism to a factor of 4 only.

Power Efficiency. Not only could we see that the the speed-up from exploiting DLP has outstanding effect but also energy saving has been reduced enormously especially with the highly data-dependent application in the [16]. Those research [15] which use static compiler analyzer to exploit DLP has not expressed their advantage over the energy consumption. Therefore, our survey is concluded that run-time dynamic binary translator could have a huge advantage on saving energy consumption.

Machine Learning Applications. Most binary translations are manually manipulated, including parameters, validation, and inspection. [1] demonstrates a novel perspective of binary translation that operating under unsupervised machine learning. Training and testing dataset is formed the same compiler. Input the dataset to recurrent neural network (RNN), which is universally used for natural language processing (NLP). After a couple training procedure, machine can determine the targeted assembly code at the accuracy of 95%. The performance is said to be enhanced if utilizing more powerful neural networks, e.g. long short-term memory (LSTM). Nevertheless, problems accompany with novel ideas. How to verify the semantical equivalence between two structures and how to compile useful dataset for training are two major concerns. We can imagine that the performance of binary translation will be significantly improved once the proper solution is proposed. It is a promising lecture that we can try to conquer.

We have presented the role of binary translator in different scenario ranging from the dynamically SIMD assembler, static loop detection mechanism and several application. The importance of binary translator in data-level parallelism is indispensable.

5 CONCLUSIONS

After evaluating the enhancements and implementation difficulties, we propose our view of possible solutions on this topic. Intra-window with partial profile provides enough information for machine to execute DLP operations. Dynamic scheduling is able to utilize the partial profile with data dependencies and balance the workload between processors. Lastly employs HMTBT to pursue significant translation performance. Integrating these three novel approaches into a whole machine requires huge efforts. We believe this concept could be a prospective direction to forward researches.

Our survey shows that DLP assists the system performance either in aspect of energy or speedup perspective. Considering computer structure limitations in overall could we find the best architecture to our targeted design. We aim to fulfill a DLP optimization with more practical constraints, e.g. memory dependencies, cold start overhead, data stream in reality with the proposed solutions. By doing so, unexpected conditions have a chance to be addressed. Probably, novel methods will be realized while we address them.

6 APPENDIX

Below is Authors and contributions

Chun-Sheng Wu: Beforehand Information Fetching, Data Partitioning Strategy, Kernel Partitioning Strategy, Scheduling Optimization, Adaptive DLP and Machine Learning Algorithm (RNN).

Ting-Wei Su: Binary Translation Optimization, Dynamic Binary Translation, SIMD Optimization.

7 REFERENCE

[1] Fei Zuo et al. "Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs". In: *Proceedings 2019 Network and Distributed System Security Symposium* (2019). DOI: 10.14722/ndss.2019.23492. URL: http://dx.doi.org/10.14722/ndss.2019.23492.

[2] Michael Chu, Rajiv Ravindran, and Scott Mahlke. "Data Access Partitioning for Fine-grain Parallelism on Multicore Architectures". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 2007, pp. 369–380. DOI: 10.1109/MICRO.2007.15.

[3] Zijun Han et al. "Exploit the data level parallelism and schedule dependent tasks on the multi-core processors". In: *Information Sciences* 585 (2022), pp. 382–394. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2021.10.072. URL: https://www.sciencedirect.com/science/article/pii/S0020025521010963.

[4] Emilio G. Cota and Luca P. Carloni. "Cross-ISA Machine Instrumentation Using Fast and Scalable Dynamic Binary Translation". In: *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE 2019. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 74–87. ISBN: 9781450360203. DOI: 10.1145/3313808.3313811. URL: https://doi.org/10.1145/3313808.3313811.

[5] Xuan Guo and Robert D. Mullins. "Accelerate Cycle-Level Full-System Simulation of Multi-Core RISC-V Systems with Binary Translation". In: *CoRR* abs/2005.11357 (2020). arXiv: 2005.11357. URL: https://arxiv.org/abs/2005.11357.

[6] Gabriele Mencagli et al. "Raising the Parallel Abstraction Level for Streaming Analytics Applications". In: *IEEE Access* 7 (2019), pp. 131944–131961. DOI: 10.1109/ACCESS.2019.2941183.

[7] Richard Littin et al. "Block Based Execution and Task Level Parallelism". In: 1998.

[8] Seyed A. Rooholamin and Sotirios G. Ziavras. "Modular vector processor architecture targeting at data-level parallelism". In: *Microprocessors and Microsystems* 39.4 (2015), pp. 237–249. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2015.04.007. URL: https://www.sciencedirect.com/science/article/pii/S0141933115000502.

[9] Charith Mendis and Saman Amarasinghe. "GoSLP: Globally Optimized Superword Level Parallelism Framework". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276480. URL: https://doi-org.srv-proxy1.library.tamu.edu/10.1145/3276480.

[10] Michael Chu, Rajiv Ravindran, and Scott Mahlke. "Data Access Partitioning for Fine-grain Parallelism on Multicore Architectures". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 2007, pp. 369–380. DOI: 10.1109/MICRO.2007.15.

[11] Lili Song et al. "C-Brain: A Deep Learning Accelerator That Tames the Diversity of CNNs through Adaptive Data-Level Parallelization". In: *Proceedings of the 53rd Annual Design Automation Conference*. DAC '16. Austin, Texas: Association for Computing Machinery, 2016. ISBN: 9781450342360. DOI: 10.1145/2897937.2897995. URL: https://doi.org/10.1145/2897937.2897995.

[12] Franjo Plavec, Zvonko Vranesic, and Stephen Brown. "Exploiting Task- and Data-Level Parallelism in Streaming Applications Implemented in FPGAs". In: *ACM Trans. Reconfigurable Technol. Syst.* 6.4 (Dec. 2013). ISSN: 1936-7406. DOI: 10.1145/2535932. URL: https://doi.org/10.1145/2535932.

[13] Tiago Knorst et al. "An energy efficient multi-target binary translator for instruction and data level parallelism exploitation". In: *Design Automation for Embedded Systems* (Jan. 2022), pp. 1–28. DOI: 10.1007/s10617-021-09258-6.

[14] Ding-Yong Hong et al. "Exploiting Longer SIMD Lanes in Dynamic Binary Translation". In: *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. 2016, pp. 853–860. DOI: 10.1109/ICPADS.2016.0115.

[15] Nabil Hallou et al. "Dynamic re-vectorization of binary code". In: *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2015, pp. 228–237. DOI: 10.1109/SAMOS.2015.7363680.

[16] Michael Guilherme Jordan et al. "Boosting SIMD Benefits through a Run-time and Energy Efficient DLP Detection". In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019, pp. 722–727. DOI: 10.23919/DATE.2019.8714826.