WPC: Whole-picture Workload Characterization Across Intermediate Representation, ISA, and Microarchitecture

This paper has been accepted by the IEEE Computer Architecture Letters, 2021

BenchCouncil: International Open Benchmarking Council
Chinese Academy of Sciences
Beijing, China
https://www.benchcouncil.org/WPC

Technical Report No. BenchCouncil-WPC-2021-1
June 15, 2021

WPC: Whole-picture Workload Characterization Across Intermediate Representation, ISA, and Microarchitecture

Lei Wang[1,2,3], Xingwang Xiong[1,3], Jianfeng Zhan*[1,2,3], Wanling Gao[1,2,3], Xu Wen[1,3], Guoxin Kang[1,3], and Fei Tang[1,3]

[1]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences , {wanglei_2011, zhanjianfeng, gaowanling}@ict.ac.cn
[2]BenchCouncil (International Open Benchmarking Council)
[3]University of Chinese Academy of Sciences

June 15, 2021

Abstract

This paper reveals that performing microarchitecture-dependent, or microarchitecture-independent, or ISA-independent workload characterization alone will lead to misleading or even erroneous conclusions. We present a whole-picture workload characterization (in short, WPC) methodology and the tool. WPC integrates microarchitecture-dependent, microarchitecture-independent, and ISA-independent characterization methodologies. It performs a whole-picture analysis on hierarchical profile data across Intermediate Representation (IR), ISA, and microarchitecture to sum up the inherent workload characteristics and understand the reasons behind the numbers. We contradict an influential observation using the WPC tool: having higher front-end stalls is an intrinsic characteristic of scale-out workloads. Our experiments show collecting microarchitecture metrics at only one level without understanding the reason behind the numbers might lead to erroneous conclusions. We open-source the WPC tool from the project homepage `https://www.benchcouncil.org/WPC`.

*Jianfeng Zhan is the corresponding author.

1 Introduction

The hardware renaissance witnesses 50,000-fold performance improvement since 1978 [5]. In this background, high-productivity languages like Java, Scala, JavaScript, and Python with complex programming frameworks like Hadoop and TensorFlow are gaining increasing popularity over performance-oriented languages like C and C++. Many programmers prefer high-productivity languages as their flat learning curve. As there are sophisticated performance optimizations across the full stacks, high-productivity languages with huge code sizes and complex stacks pose significant challenges to fundamental workload characterization approaches.

Much previous work presents microarchitecture-dependent workload characterization approaches and results, using specific processor performance counters to gain insights [8] [4]. For example, Intel VTune [2] can provide comprehensive microarchitecture data using a top-down characterization approach. Another approach is microarchitecture-independent (namely, ISA-dependent) workload characterization with run-time binary instrumentation [3] [10] [6]. Recently, Shao et al. [9] propose an ISA-independent workload characterization methodology, which uses an ILDJIT intermediate representation (IR) but only works for 32-bit operation instructions stream. However, we found that merely conducting microarchitecture-dependent or microarchitecture-independent or ISA-independent workload characterization will lead to misleading or erroneous conclusions.

The previous influential work—CloudSuite [4] performed a microarchitecture-dependent workload characterization of the scale-out workloads. The scale-out workloads refer to the data-level or request-level workloads that are often developed using high-productivity languages based on a distributed framework like Hadoop. Among the six benchmarks of CloudSuite v 1.0 in [4], five use high-productivity languages based on distributed frameworks. They concluded having a higher front-end stall is the inherent characteristic that places the scale-out workloads into a distinct workload class from desktop, parallel, and traditional server workloads.

We report microarchitecture-dependent, microarchitecture-independent, and ISA-independent workload characterization results on two typical scale-out workloads (Sort and Bayes, included in two influential benchmarks suites—CloudSuite and BigDataBench) with C, Java, Python, Hadoop (Java-based), TensorFlow (Python-based), MPI (C-based) implementations, and a traditional workload (Matrix Multiplication) with a C implementation. We also characterize but do not report the data of SPEC CPU 2017 due to the page limitation. We found that having a higher front-end stall is not the inherent characteristic of the scale-out workloads. Instead, the cascading side effect of high-productivity languages, frameworks, and compiler optimizations makes the scale-out workloads having higher front-end stalls. We confirm using the state-of-the-practice tool–Intel VTune, we can not uncover the root causes though it provides comprehensive microarchitectural data. More details can be found in Section 3.1.

Previous work [9] claimed performing ISA-independent characterization alone can avoid misleading conclusions. However, we found that it does not hold. For example, performing workload characterization on the IR stream of Sha256-C and Matmul-C (two workloads Sha256 and Matmul written in C language), we observe a vital metric—instruction mix are close. However, when performing two ISA-dependent approaches mentioned above, we notice larger deviations. The instruction characteristics similarity between two workload implementations at the IR level does not guarantee their instruction behavior characteristics will be similar at the other two levels. These observations indicate that collecting architecture metrics at only one level alone without understanding the reason behind the numbers might lead to erroneous conclusions.

This paper presents a WPC workload characterization methodology. WPC integrates the three types of traditional workload characterization approaches and performs a whole-picture analysis of hierarchical profile data. We do workload characterizations at each level and observe the same class of metrics' inconsistency at three levels to avoid the misleading or erroneous conclusions drawn through performing traditional workload characterization alone.

Fig.1 shows the framework of our methodology. First, we perform compiler-level analysis at the IR level, closest to the source code and independent from any ISAs. We use the LLVM compiler to translate the source codes into the IR codes to get the workload characterizations, assuming no cache

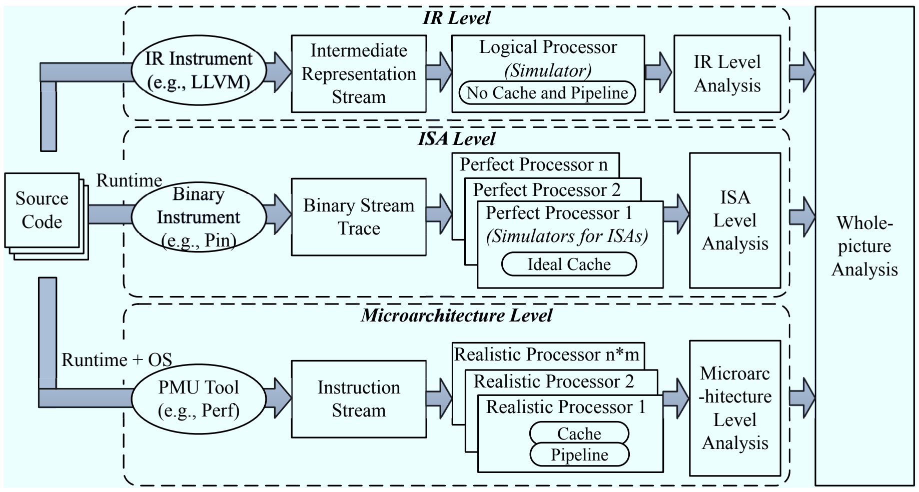

Figure 1: The Whole-picture Workload Characterization Methodology across IR, ISA, and Microarchitecture.

and pipeline. The IR-level analysis includes static and dynamic ones. The former analyzes the static IR codes, and the latter examines the IR stream. Second, we perform ISA-dependent characterization. The workloads translate into the binary machine codes and run as the OS process. At the ISA level, the binary codes stream analysis omits the effect of the specific microarchitecture decisions assuming no pipeline model but with a perfect cache model that always completes all memory references in a single cycle, and it is microarchitecture-independent. We collect the binary stream with the Pin tool. Third, we perform microarchitecture-dependent characterization. The workload runs on the processor as the instruction stream. We use a performance monitoring unit (PMU) tool like Perf to obtain microarchitecture metrics on specific processors. Overall, the IR-level analysis is ISA-independent, beyond the runtime environment and OS, while the ISA-level analysis is microarchitecture-independent, beyond the OS. The microarchitecture-level analysis is affected by the OS and microarchitecture-dependent. We collect a series of performance metrics at each level, e.g., instructions execution behavior, instruction locality, to depict workload characteristics in a combined and comprehensive way (Section 2.2). We build the WPC tool to collect, analyze, and visualize the automatically whole-picture hierarchical workload characterizations (Section 2.3).

2 Methodology and Tool

We present the WPC methodology and tool in this section. We introduce each level's method and the details of the whole-picture analysis method in the first and second subsections. Finally, we detail the tool.

2.1 The framework of our methodology

As shown in Fig. 1, our methodology covers three levels across IR, ISA, and microarchitecture. IR represents source code used in compilers. The other lower levels affect the architecture characteristics.

- **The IR Level**

The IR-level analysis includes static (static IR codes) and dynamic (IR stream) ones. The IR stream feeds to a presumed processor model with infinite registers but no cache and pipeline models for dynamic analysis. There are no pipeline and cache models; we can only obtain instruction mix, instruction locality, branch behaviors, and data locality metrics. We currently use LLVM IR, a virtual instruction set that hides the specific details of the ISA and microarchitecture [7].

- **The ISA Level**

At the ISA level, we perform ISA-specific analysis of the binary stream with a run-time binary instrument tool like Pin. We feed the trace into a perfect processor; it assumes a perfect cache model that always completes all memory references in a single cycle, but it has no pipeline model. The ISA-level workload profile contains more comprehensive performance metrics than the IR level, including instructions mix, perfect branch predictor behaviors, perfect cache behaviors, and parallelism behaviors.

- **The Microarchitecture Level**

At the microarchitecture level, we analyze an instruction stream on a specific processor microarchitecture. It uses a Perf performance monitoring unit (PMU) tool to obtain microarchitecture behaviors through accessing hardware performance counters. Compared to the two levels mentioned above, the most comprehensive microarchitecture characteristics obtained at this level include instruction mixes, branch predictor behaviors, cache behaviors, TLB behaviors, pipeline system behaviors, etc.

2.2 Whole-picture Analysis

We observe the same class of metrics' inconsistency at three levels to avoid the misleading or erroneous conclusions drawn through performing traditional workload characterization at one level alone. The metrics include instruction mix, instruction locality, data locality, branch predictability [9], and parallelism. We take the instruction mix and instruction locality as two examples to illustrate our methodology for space limitation.

2.2.1 Instruction Mix

We conclude five classes of instructions at three levels for whole-picture analysis, including Load, Store, Branch, Floating-point operations, and Integer operations. We only report the instruction mix of the IR stream at the IR level following the convention [9]. Furthermore, we use the cosine similarity to measure the similarity/dissimilarity of two metrics (Section 3.2).

2.2.2 Instruction Locality

The instruction locality is a vital metric [4]. At the IR level, we use instruction entropy, which calculates the probability of access frequency of each static instruction or dynamic stream instruction to measure the instruction locality. At the ISA and microarchitecture levels, we use the ICache miss ratio and the ICache MPKI (Misses per Kilo Instructions). Furthermore, we use a workload's normalized value relative to that of a referenced one to calculate the similarity/dissimilarity (Section 3.1).

2.3 The WPC Tool

We develop the WPC tool to collect, analyze, and visualize performance metrics. It mainly consists of two parts: multi-level profiler and performance data analyzer. The profiler profiles the workloads and gathers performance data, and then the performance data analyzer processes those data with different modules. The profiler transforms Java or Python bytecode into the LLVM bitcode and profiles the LLVM bitcode on the IR level. We use Pin and Perf tools at the ISA and microarchitecture levels, respectively. After running each workload, the performance data analyzer's collector module will collect all the data from the profiler and store them into the database. The analyzer reads raw data from the database and calculates metrics. The figure plotter plots different kinds of figures.

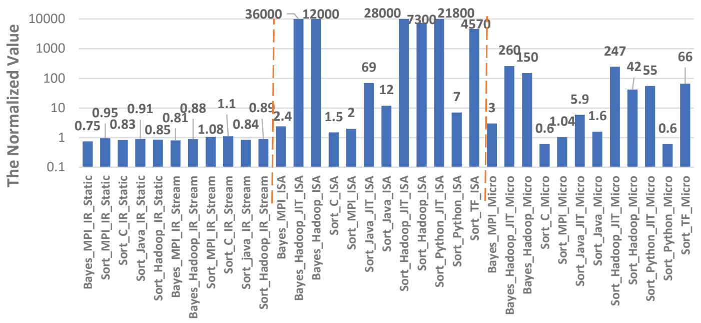

Figure 2: The instruction locality relative to Matmul-C at the IR, ISA, and Microarchitecture levels. The value of Matmul-C is 0.02 (ICache MPKI), 0.00004% (ICache miss ratio), 2.3 (instruction entropy of the IR stream), 2.4 (instruction entropy of static IR codes). The meaning of notations in the figure is as follows. Taking a Java implementation of Sort as an example, Sort_Java_JIT_ISA and Sort_Java_ISA in pair indicate turning on and off JIT, and report the ISA-level values.

3 Preliminary Evaluations

We deploy the workloads on the Intel Xeon CPU E5-2620 v4 processor equipped with 128 GB memory and 8 TB disk. The OS is Ubuntu 16.04. We use LLVM (version 9.0), Hotspot (version 1.8), TF-compiler (Version 1.2), Pin (Version 3.17), Perf (Version 4.4), and VTune (Version 2021.3.0) as the profile tools. At the ISA level, we set the cache size, cache block size, and replacement algorithm of the perfect cache model to 32 KB, 32 bytes, and Round-Robin algorithms.

We implement Matrix Multiplication (Matmul), Cryptographic Hash (Sha256), and Merge Sort (in short, Sort) with three languages (C, Java, Python, and Tensorflow). We use the MPI and Hadoop implementations of Bayes and Sort from BigDataBench [1]. The Matmul's input data is 1024*1024 double floating-point matrix, and the algorithm is the block matrix multiplication. The Sort input data is an array of 10E6 integers for each task. The Sha256's input is a random string with a length of 20M. The dataset of Bayes is 500 MB. We repeat each experiment three times and report the average values.

3.1 Is having a higher front-end stall an intrinsic characteristic of scale-out workloads?

The influential work–CloudSuite [4] claims that having a higher front-end stall is an inherent characteristic of the scale-out workloads. We use and compare WPC and VTune to certify this observation. In the rest of this paper, we use Matmul-C to refer to the C-implementation of Matmul. The other workloads use a similar notation. Sort-MPI, Sort-Hadoop, Bayes-MPI, and Bayes-Hadoop represent typical scale-out workloads, as they all run multiple concurrent tasks. And Matrix Multiplication (Matmul) represents a typical traditional workload. We want to investigate the effect of high-productivity languages like Java, Python. Also, we feel interested in the impacts of distributed frameworks when programming with high-productivity or high-performance languages: Hadoop (Java-based) vs. MPI (C-based).

If having a higher front-end stall is an intrinsic characteristic of scale-out workloads, their instruction locality should be poorer than the traditional workload. As shown in Fig.2, we use WPC to analyze the instruction locality of workloads at three levels. In Fig.2, we do not report the value of Sort-Python and Sort-TF at the IR level because the implementation of the Python profile tool at this level is ongoing. We do not report instruction entropy of the IR static codes of Bayes-Hadoop, because it is encapsulated in the Mahout algorithm package, hard to obtain its source codes.

We have the following observations. First, the instruction entropy of the static IR codes (the _IR_Static notations in Fig.2) is similar, and the gap is not more than 1.3 times. For the same workload with different implementations, the gap is not more than 1.2 times. This implied that the instruction locality of Sort, Bayes, and Matmul is similar too. Second, the instruction entropy of the IR stream (the _IR_Stream notations in Fig.2) is similar, and the gap is not more than 1.35 times. For the same workload, the gap is not more than 1.3 times. So, the compiler does not affect the locality. Third, the ICache miss ratio of Bayes-Hadoop, Sort-Java, Sort-Hadoop, Sort-Python, and Sort-TF implementations is two or three orders magnitude that of Matmul-C at the ISA level. On the other hand, the ICache miss ratio of Bayes-MPI, Sort-C, and Sort-MPI is similar to that of Matmul-C. It implies the instruction localities at the ISA level changes, which is impacted by the runtime environment, especially the high-productivity language. Forth, at the microarchitecture level, the ICache MPKI of the high-productivity language implementation is higher than that of Matmul-C. At the same time, Bayes-MPI, Sort-C, and Sort-MPI's values are similar to Matmul-C. Thus, the ISA level and microarchitecture level results are consistent, and the OS does not affect the locality (the ratio of user-mode of each workload is not less than 95%).

So, we speculate that a specific optimization of the runtime environment of the high-productivity languages causes the difference in instruction locality. We try turning off Java and Python's JIT (just-in-time) compilation option and change into an interpreter mode. We only turn off JIT of the ResourceManager, NodeManager, and HDFS for the Hadoop framework. We find that the entropy at the ISA changes accordingly. For example, the instruction entropy of Sort-Java varies from 1.8 to 1.7 at the ISA level, implying the instruction locality changes. We also see that their ICache MPKI at the microarchitecture level and ICache miss ratio at the ISA level reduce too, shown in Fig.2. But the performance of Sort-Java and Sort-Python deteriorates 6.4 and 2.2 times when turning off JIT. We notice the ICache MPKI of Sort-Hadoop and Bayes-Hadoop has a small change, and the performance of Sort-Hadoop and Bayes-Hadoop also deteriorates a little.

We conclude that scale-out workloads' instruction locality is similar to that of the traditional workload; having a higher front-end stall is not the inherent characteristic of the scale-out workload. Instead, the cascading side effect of high-productivity languages, frameworks, and compiler optimizations makes the scale-out workloads having higher front-end stalls.

Also, we use VTune to uncover the reason at the microarchitecture level. Besides analyzing ICache MPKI (the _Micro notations in Fig.2), VTune evaluates the pipeline efficiency of workloads, and classifies an issued micro-operation into retiring, bad speculation, frontend bound, and backend bound. Corroborating the observations in [4], the front-end bound percentages of Sort-Hadoop (22%) and Bayes-Hadoop (21%) are higher than Matmul-C (8%). Furthermore, the implementations using high productivity languages like Java, Python, or TensorFlow also have higher front-end bound percentages (the percentage ranges from 25% to 30%). Both the front-end bound percentage of Sort-C (9%) and Sort-MPI (12%) is low. Since the IPC of Bayes-MPI is high as 2.6, we did not evaluate the pipeline efficiency further. Indeed, the ICache MPKI of Bayes-MPI is only 0.06. So, using VTune, also we can conclude that the high-productivity languages and frameworks incur higher front-end stalls. The result of Sort-MPI and Bayes-MPI indicate the distributed framework is not the root cause. But we can not answer the other issues as follows. Whether the higher front-end stall is the inherent characteristic of the scale-out workload? Whether the scale-out workload's instruction locality is poor? And what results in the higher front-end stalls of high-productivity languages? This case motivates why WPC is essential.

3.2 What is the implication of WPC for architecture metrics

Instruction mix is one of the fundamental architecture metrics. This subsection takes instruction mix as an example to discuss the implication of WPC for an architecture metric. We measure the instruction mixes of all C and Java implementations workloads: Matmul, Sha256, and Sort at three levels and compare their pairwise similarity. Fig. 3 depicts the cosine distances of instruction mixes among each workload implementation at three levels. Furthermore, we subset these workloads using the metrics collected by WPC at the three levels. The metrics include instruction mix, instruction locality, data locality, branch predictability, and parallelism at three levels. We use the K-Means algorithm to cluster the workloads into four groups, which are [Sha256-C, Sha256-SSL-C, Sha256-Java], [Sort-C, Sort-Java], [Matmul-Java, Sort-Hadoop], and [Matmul-C].

We have the following observations. First, the instruction characteristics similarity between two workload implementations only at the ISA and microarchitecture levels do not guarantee their instruction behavior characteristics will be similar. For example, Matmul-Java and Sort-C have high similarity at

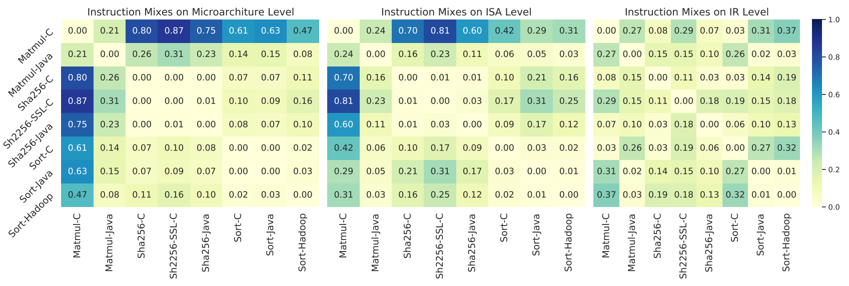

Figure 3: The symmetry matrices measure the cosine similarity of instruction mixes among different implementations of three workloads at three levels.

both ISA and microarchitecture levels, and their cosine distances are not more than 0.14. Still, there is a low similarity at the IR level, with the cosine distance being high as 0.26. Matmul-Java and Sort-C have different algorithms and runtime, and their instruction mix ratios are aggregated at ISA and microarchitecture levels. The subset result using WPC also validates it. Matmul-Java and Sort-C are not in the same subset group.

Second, the similarity of instruction characteristics between two workloads at the IR level is not necessarily consistent with those at the other two levels. For example, Sha256-C and Matmul-C are similar at the IR level (the cosine distance is 0.08) but not similar at the ISA and microarchitecture levels (0.698, 0.80, respectively). Indeed, their algorithms are different. Matmul-C has lots of floating-point operations, while Sha256-C is integer operation-intensive. The subset result using WPC also validates it. Matmul-C and Sha256-C are not in the same subset group.

Furthermore, the implementations of the same workloads calling a dynamic-link library versus directly implementing the functions in the source code have significantly different workload characteristics at the IR level. For example, there are two C language implementations of Sha256: Sha256-SSL-C and Sha256-C. Sha256-SSL-C calls an OpenSSL dynamic-link library, while Sha256-C directly implements SSL in the source code. We notice Sha256-C and Sort-C are similar, as their cosine distances are less than 0.1 at three levels. In fact, Sha256 and Sort's algorithms are similar in instruction mix as they both have intensive data access and integer operations. However, we observe Sha256-SSL-C and Sort-C are not similar at the IR level (the cosine distance is 0.19) but similar at the microarchitecture level (the cosine distance is 0.1). The reason is that IR only provides an entry to a function call of the dynamic-link library.

In conclusion, WPC is indispensable for us to understand the reason behind the numbers.

Also, we evaluate the analysis overhead at three levels and notice significant differences. For example, the instruction mix evaluation of the C implementation of the Sort workload consumes 31, 15, and 8 seconds at the IR, ISA, and microarchitecture levels, respectively.

4 Conclusion

This paper reveals that performing microarchitecture-dependent, or microarchitecture-independent, or ISA-independent workload characterization alone will lead to misleading or even erroneous conclusions. We propose the WPC methodology and tool to understand the reasons behind the numbers and avoid misleading conclusions.

References

[1] "Bigdatabench: https://www.benchcouncil.org/bigdatabench."

[2] "Intel vtune: https://software.intel.com."

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: Characterization and architectural implications,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[4] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the clouds: A study of emerging workloads on modern hardware,” *Architectural Support for Programming Languages and Operating Systems*, 2012.

[5] J. L. Hennessy and D. A. Patterson, “A new golden age for computer architecture,” *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[6] K. Hoste and L. Eeckhout, “Comparing benchmarks using key microarchitecture-independent characteristics,” in *2006 IEEE International Symposium on Workload Characterization*. IEEE, 2006, pp. 83–92.

[7] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[8] R. Panda, S. Song, J. Dean, and L. K. John, “Wait of a decade: Did spec cpu 2017 broaden the performance horizon?” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 271–282.

[9] Y. S. Shao and D. Brooks, “Isa-independent workload characterization and its implications for specialized architectures,” in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 245–255.

[10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The splash-2 programs: Characterization and methodological considerations,” *ACM SIGARCH computer architecture news*, vol. 23, no. 2, pp. 24–36, 1995.