# ZIP-IO: Architecture for Application-Specific Compression of Big Data

Sang Woo Jun*, Kermin E. Fleming*, Michael Adler† and Joel Emer†
*Computation Structures Group, CSAIL
Massachusetts Institute of Technology
{wjun, kfleming}@csail.mit.edu
†VSSAD Group
Intel Corporation
{michael.adler, joel.emer}@intel.com

***Abstract*—We have entered the "Big Data" age: systems are being asked to deal with exponentially increasing amounts of data. Moore's law continues to deliver cheaper and more plentiful transistors with which to process this data, but pin count, bandwidth, and frequency have not scaled as quickly, exacerbating the already painful I/O bottleneck. As data sets continue to scale in size, this bottleneck represents a serious concern in system architecture.**

**Compression is an effective way to deal with many large data sets. As a result, application-specific compression algorithms, which provide superior compression to general-purpose entropy compression algorithms, have become popular. Unfortunately, compression algorithms are often computationally difficult and can resulting in application-level slow-down when implemented in software. To address this issue, we investigate integrating a framework for FPGA-accelerated compression into a general-purpose system. Using this system we demonstrate that an unmodified industrial software workload can be accelerated 3x while simultaneously achieving more than 1000x compression in its data set.**

## I. Introduction

The evolution of computer networks and the increasing scale of electronic integration into our daily lives has lead to an explosion of data. Individuals post entire life-times worth [2] of photos to the internet; camera networks monitor traffic activiry across entire cities. Moreover, the size of this data is growing exponentially as networks and sensors become cheaper to deploy. One of the key challenges in computer architecture moving forward is operating efficiently on these large data sets.

Fortunately, Moore's law continues its steady march, and transistors continue to get cheaper and more plentiful, balancing the steady growth in data. However, the future is not completely bright. While the potential for computation continues to scale, memory, chip, and network bandwidths are not scaling nearly as quickly, exacerbating the already-painful I/O bottleneck. As we move farther into the "Big Data" age, the problem becomes not how to compute the data on chip, but how to get the data from storage or sensors, across the network, and onto the chip.

One solution to these burgeoning I/O and capacity problems is data compression: trading computation at the processor for increased bandwidth and capacity in network and storage systems. The idea of applying compression to I/O systems is not new and has been studied in several contexts. Many existing file systems [15], [19] support software-based data compression to increase apparent disk storage capacity. IBM developed hardware data compression to extend DRAM capacity [22], but this technology did not gain wide deployment. Recent FLASH-based storage systems [5] are thought to include automated support for compression to help hide the effect of write asymmetry in FLASH systems.

The main failing of these efforts is that general purpose compression schemes [24] do a a poor job of compressing generic data, resulting in limited performance gains [14]. To get higher levels of compression one must develop *application-specific* algorithms that take advantage of the intrinsic properties of a specific data set. Application-specific compression is also not new: compression algorithms for important data sets like video [8] have been well-studied. However, as problem sizes have grown, compression algorithms have been proposed for many high-performance workloads, including processor simulation [12], bioinformatics [13], and web search [7]. These algorithms seek to either increase the amount of data that can be stored locally (e.g. in fast DRAM) or to minimize the amount of traffic on the network, while improving performance metrics like run time.

The question, then, is how to implement these algorithms. One possibility, as general-purpose core counts scale, is to map compression algorithms onto dedicated cores. However, compression does not map well to general-purpose processors or GPUs. Compression algorithms typically feature complex, scheme-specific bit manipulations and highly variable control which are ill-suited to the wide, deep processing pipelines found in modern GPUs and CPUs. Moreover, most compression algorithms have tight feedback loops and strong data-dependencies, making core-level parallelization difficult. Using general purpose processors to implement compression and decompression operations may *slow* application codes layered on top of these algorithms to the point that using the best available compression scheme becomes a losing proposition in terms of run-time. As compression schemes become more complicated in an attempt to balance data volume, these performance issues worsen.

Compression implementations are better suited to computational structures supporting fine-grained bit manipulation, variable control, and fast feedback. As a result, application-specific compression schemes that require high performance, such as video, are typically implemented in hardware. However, due to cost, hardware implementation can only be considered for widely deployed applications, leaving a large set of applications in need of acceleration. Fortunately, most compression

schemes are also amenable to implementation on a fine-grained reconfigurable substrate, such as FPGA. As transistor counts increase, including such a substrate on die becomes both increasingly feasible and increasingly attractive [21], since fine-grained reconfigurable substrates capture different workloads than general purpose processors.

Compression maps well to FPGAs. However, integrating general-purpose software with FPGAs based accelerators, and, in general, implementing anything on FPGAs, is a difficult proposition. An important system-level consideration in constructing a general framework for application-specific compression is that we must integrate seamlessly with existing software. To ease this integration, we provide a novel FPGA-based implementation of the UNIX Standard I/O library, which permits the integration of user programs and FPGA-accelerated compression by way of commonly used shell commands.

Processor trace compression, which we will consider at length in this paper, is one example of a highly-compressible workload well-suited to decompression on the FPGA. Although recognized as computationally difficult, processor simulation is not typically thought of as a "Big Data" problem. However, to avoid the cost of running largely redundant functional simulations, detailed simulators may store a trace of instruction-by-instruction execution results. These execution traces are then used to drive a diverse set of simulators [9], [10], which can be orders of magnitude faster than the original detailed simulation. Traces are enormous: even for small programs, a detailed trace can consume, uncompressed, terabytes of storage. Moreover, typical production simulation systems consist of thousands of machines simultaneously accessing traces. This level of traffic can quickly cripple even high-end storage arrays. As a result, processor traces are a prime candidate for application-specific compression.

In this paper, we examine a prototype system architecture for *application-specific* compression, which we call ZIP-IO. This system consists of a processor and a tightly coupled reconfigurable substrate. Using trace-based processor simulation as an example, we will show that our system architecture not only provides the opportunity to deploy compression, and thereby network bandwidth and storage system utilization, but also that existing user-level applications can be accelerated by significant multiples as compared to a conventional software implementation.

## II. Background

### A. Trace-Based Simulation

Detailed simulation of processors is a computationally intensive task. Detailed software simulators operate at speeds of kilo-instructions per second, and complete benchmark executions take days to complete. Since architectural path-finding times are limited, architects frequently adopt a two-phased approach to simulation. Detailed functional simulations are run once, and during the run a trace of architectural updates is collected. This trace contains all architectural state changes, such as register writes and status flag updates, for each instruction executed. Traces are quite large, since each instruction takes tens of bytes to store and programs execute trillions of instructions. These traces are then repeatedly reused to drive many other simulators. For example, a full instruction trace can be used to extract memory traces, which can, in turn, be used to drive a cache hierarchy simulator. These secondary, trace-driven simulators run orders of magnitude faster than the original detailed simulation, permitting a broader range of architectural exploration.

The two-phased approach has a second benefit: parallelism. Once the traces have been collected, fast, parallel simulation is available. This parallelism is important because typical architectural studies will test a linear combination of parameters, for example, cache size and associativity, across a set of benchmark programs, such as SPEC [4]. Since each program/architecture experiment is independent, each can be run separately.

To exploit this parallelism, and to complete experiments quickly, industrial and academic researchers use large network batching systems. Although parallelism is abundant in this workload organization, sophisticated network infrastructure is needed to support all of the machines running simultaneously. As we remarked previously, detailed instruction traces are enormous. Parallel simulators making use of these traces do not have the disk capacity to store even a single full-length trace in their local disk. As a result, systems designed for processor simulation make use of large-scale, expensive, network storage. However, since many experiments run in parallel, this shared storage can represent a significant performance bottleneck. As figure 5(a) shows, even a handful of simulators simultaneously accessing a network store can result in large performance loss. Worse, industrial simulation deployments frequently have thousands to tens of thousands of experiments running in parallel.

### B. Trace Compression

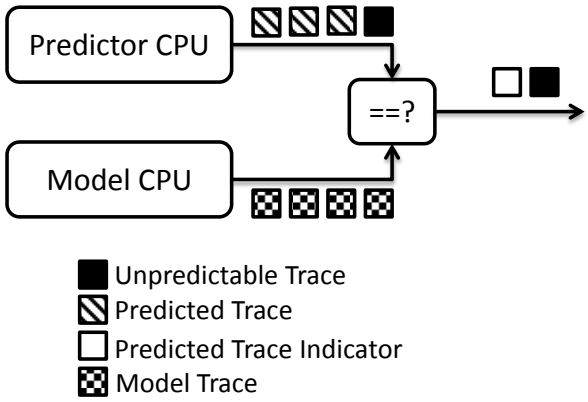



Fig. 1. Compression using Instruction Interpretation

Because traces are so large, significant effort has been spent in attempting to compress them. Most compression efforts [6], [16] have focused on specific kinds of traces, for example instruction streams. These specialized compressors can obtain

compression ratios hundreds to thousands of times better than conventional entropy compression. However, to minimize the need for detailed simulation, a general trace compression scheme, from which many different simulators can be driven, is needed. General trace compression [11] schemes have also been implemented using entropy-based methods tuned for traces. However, entropy based compression schemes are not attractive for high-performance implementation because they tend to be memory bound.

Cohn and Kanev [12] have shown that due to the characteristics of program execution, it is possible to implement highly efficient compression of instruction traces by emulating a simple CPU. Although modern processors are very complicated (even RISC machines have hundreds of instructions [3]), the majority of executed instructions are drawn from a small part of the ISA. Therefore, a large portion of program execution can be emulated using a simple *predictor CPU* that implements only the frequently used instructions. Using this observation, the execution trace can highly compressed by recording only those instructions not implemented by the predictor CPU. This algorithm, called *Zcompr*, is highly amenable to FPGA implementation because the core of the compression algorithm, the predictor CPU, is effectively hardware and largely streaming. This stands in contrast to traditional entropy trace compression schemes which operate on complicated, memory-based data structures to store portions of the trace. Because Zcompr is a general trace compression algorithm, it can be used to drive any trace-based simulation with minimal post-processing.

Figure 1 shows a block diagram of the compression process. During the detailed trace run, the behavior of the detailed model CPU and the predictor CPU are compared at each instruction. If the two CPUs produce matching outputs, then a marker is recorded in the compressed trace and the state update is omitted. However, when the predictor encounters an instruction that it does not implement, the complete state modification made by the detailed model must be recorded in the compressed output. Decompression is the inverse of compression: the predictor CPU will again run over the program, inflating the omission indicators. When the decompression predictor CPU encounters an unimplemented instruction, it injects the state update stored in the compressed trace stream and continues execution after patching its internal state to reflect the execution of the un-implemented instruction.

Figure 2 shows the percentage of executed instructions that are covered by our Zcompr implementation. The original Zcompr implementation did not include support for traps, protection management, and syscalls; neither does our implementation. Since we target the FPGA, we also omit floating point operations and certain complex memory operations. The remaining 10% of the instruction set accounts for more than 98% of the dynamic executions across the SPEC benchmarks, suggesting that 50 to 1 compression ratios should be achievable. In reality, the compression ratio is slightly less, both because, in addition to unimplemented instructions, the predictor CPU cannot correctly predict the effects of external processor I/O events and because some meta-data must be included in the compressed trace.

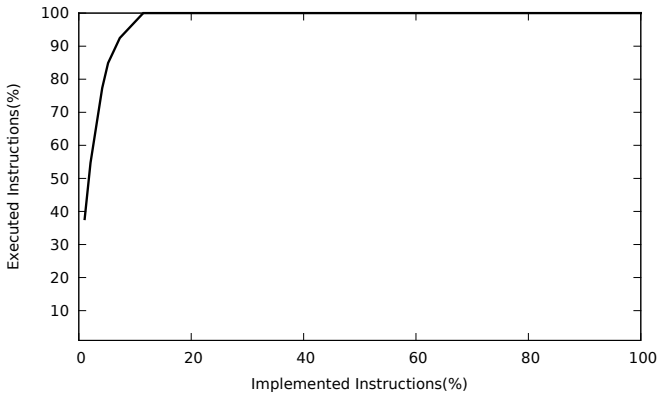



Fig. 2. Percentage of Implemented Instructions in Instruction Trace

### C. FPGA-Software Interfacing

The goal of ZIP-IO is to provide an easy-to-use and high-performance architecture for accelerating big data applications through application-specific compression. Good support for FPGA-software I/O is essential to ZIP-IO: it should be as easy to integrate ZIP-IO with existing software programs as it is to integrate other common Unix tools with them. Unfortunately, FPGAs generally have no easy-to-use software interfaces. Indeed, the problem of FPGA-software co-design has traditionally been viewed as difficult and error prone. This difficulty stems from the raw nature of physical devices, both in hardware and software, which require substantial expertise to incorporate into a design.

To address these issues, we have, over the past several years, developed the LEAP[1] FPGA operating system. The LEAP framework is designed to serve many of the roles of operating systems on general purpose processors, enabling portable application development across a variety of hardware. LEAP abstracts platform-specific device interfaces into a set of general interfaces, common across all FPGA platforms. For example, LEAP provides Host-FPGA I/O with an RPC-like mechanism, RRR [18]. To communicate with processes running on the host processor, developers instantiate simple communications stubs in their FPGA source. In this work, we extend LEAP to support a subset of the commonly used Unix STDIO library.

Other attempts at FPGA operating systems have been made. For example, BORPH [21], also addresses the I/O problem [20]. BORPH unifies the operating system file descriptor space across software and FPGAs, permitting communication between reconfigurable logic and general purpose processors using Unix pipes. However, BORPH's hardware implementations are largely static, for example BORPH does not admit of hardware opening arbitrary file handles dynamically, a functionality that we support in our STDIO implementation.

## III. ZIP-IO Architecture

### A. System Overview

Figure 3 shows the reference implementation of what we envision to be a typical deployment of ZIP-IO, namely a network store in which data resides and a tightly coupled FPGA-general purpose processor connected by a fast interconnect. On to this generic system, we have superimposed the data-flow of our Zcompr implementation to illustrate architectural support needed by a typical compression algorithm.

Our implementation incorporates three different methods to compress instruction traces: simple entropy compression of the full text format traces, an FPGA implementation of Zcompr, and gzip. These programs are connected in a processing pipeline and interposed between the compressed trace in the network store and the final consumer of the decompressed trace: a trace-driven simulator.

We augment Zcompr, which has been accelerated on the FPGA, with gzip to obtain even larger compression ratios. It may seem counter-intuitive that we have chosen to place gzip in software, since we have argued that compression algorithms are not well-suited to general purpose cores. However, in this case, gzip is operating on a significantly compressed data and does not represent a performance bottleneck. To the contrary, requiring a user to implement gzip, or a similar entropy compression scheme, in hardware for a marginal performance gain represents an unnecessary burden on the user.

### B. Handling I/O Complexity

One goal of ZIP-IO is that FPGA-based accelerators, like trace compression, should be as easy to integrate with existing software as existing Unix tools. Therefore, ZIP-IO accelerators must be able to connect to software applications using existing software interfaces, and users should be able to incorporate FPGA accelerators into natural program flows:

```
./runhw | ./runsw
```

The most common modality of interacting with compression algorithms in software is through file I/O and pipes. Therefore, we have extended LEAP to support a Unix-style STDIO library. The LEAP STDIO service implements a subset of STDIO methods, including allocation of file and pipe handles, formatted printing and raw I/O. Although it provides much of the functionality of Unix STDIO, the LEAP STDIO library has a different, FPGA-tuned interface.

LEAP STDIO relies on software support. Effectively all STDIO operations are carried over RRR to the host processor, where they are executed using the host STDIO implementation. Because LEAP STDIO relies on software support, operations that require a meaningful response, such as fread, can have long latencies relative to the speed of the user hardware. To help hide the latency of these operations, we separate them into distinct request and response operations, rather than the blocking calls found in C. This spereration permits user programs to issue multiple outstanding requests, overlapping the latency of the requests.

A critical step in realizing an STDIO implementation in FPGA is handling strings well, because strings are the primary means of interacting with many basic STDIO operations. Unfortunately, strings create problems in hardware because hardware is static and fixed in size, while strings are intrinsically unbounded structures. We solve the strings problem by creating a new string handle primitive in our HDL. Strings are never passed directly from hardware to software. Instead, a global string table is constructed at compile time and shared between hardware and software. Only string handles, effectively pointers into the table, are passed as printf format strings, file names in fopen, etc. Although the string table is statically initialized, it can be dynamically modified. For example, LEAP STDIO provides an sprintf function that returns a dynamically allocated string handle, permitting dynamic construction of new string references in the hardware itself.

ZIP-IO implementations use the LEAP STDIO library both consumes the compressed input trace and writes back the uncompressed result. For trace compression, the input is a software pipe produced by gunzip and the output is a pipe to an architectural simulator. ZIP-IO also uses STDIO to write log files useful for debugging.

### C. Interfacing with Programs

Because ZIP-IO provides such a simple interface to programmers, integrating it into an industrial simulation flow is easy. Decompressed data is fed using a common Unix FIFO to the input of the trace-driven simulator. Allowing us to conduct complete end-to-end system using unmodified real-world applications. The entire system execution can be invoked using the following sequence of commands:

```
mkfifo compressed.bin
mkfifo decompressed.log
gunzip -c /[network]/compressed.gz \
        > compressed.bin \&
./run_hardware.sh compressed.bin \
        > decompressed.log \&
./simulator [configuration] <\
decompressed.log
```

It is worth noting that the simulator used in this script requires neither modification nor recompilation to integrate with ZIP-IO. However, we typically need to insert a program to adapt the output of ZIP-IO to the format expected by the simulator. This trivial program is needed even for conventional software implementations.

## IV. Implementing Zcompr

### A. ZCompr Microarchitecture

The blowup on the right of Figure 3 shows the microarchitectural details of the hardware Zcompr implementation. The main component of the system is the predictor CPU, which consists of a prediction core and the core controller. The prediction core implements a subset of the MIPS ISA. This core is similar to a normal pipelined RISC core, except that it is augmented with extra control interfaces used to

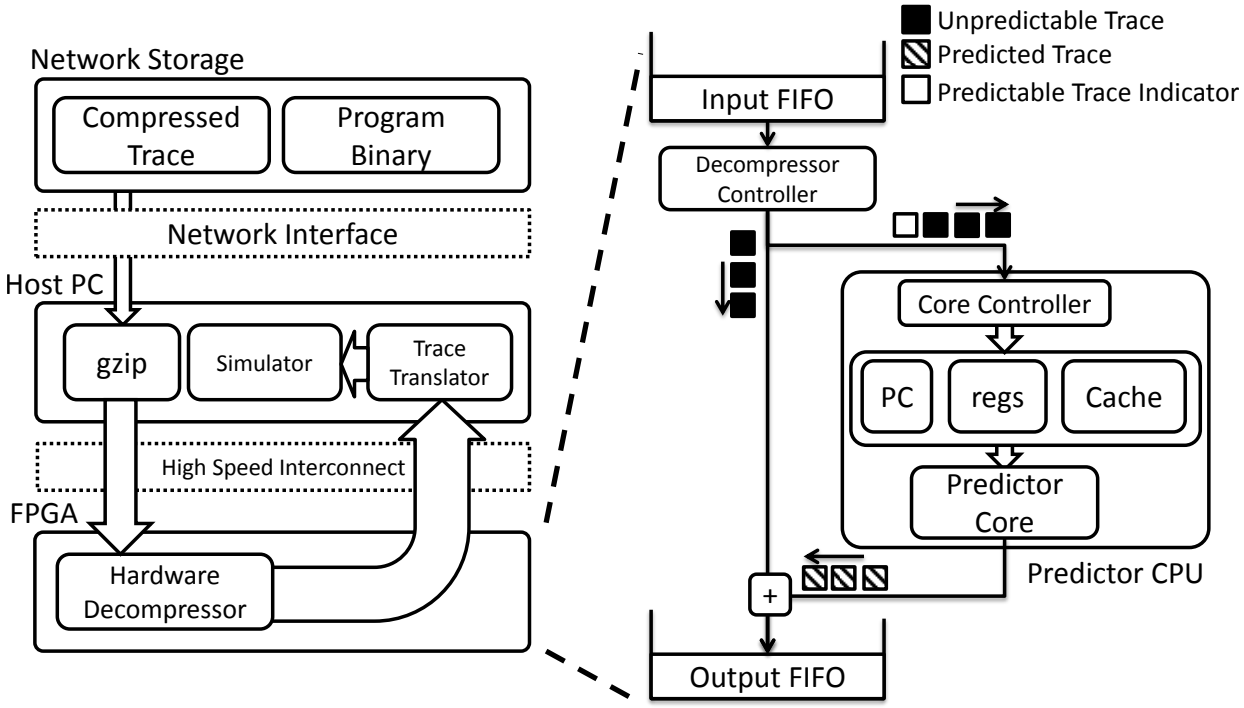



Fig. 3. Structure of the ZIP-IO System

handle unimplemented instructions and to export state updates. The first control interface halts processor execution before the issue of a particular instruction. The controller core uses this interface to stop the processor from beginning to execute unimplemented instructions. The state elements of the predictor CPU - the PC, register file, and memory system are modifiable by way of a second control interface. To the predictor core, these structures appear unmodified. However, the controller core can modify any value in the state space, permitting the controller to correct the state of the predictor CPU in the case of an unimplemented instruction. Finally, the predictor CPU outputs state updates as a part of its final pipeline stage.

During operation, a compressed trace is streamed from the host PC via LEAP STDIO. There are two possibilities for the compressed trace: either it is a correctly predicted instruction or it is an unimplemented instruction. When the controller core recieves a correctly predicted marker, it feeds a control token to the core controller and predictor CPU. This allows the predictor CPU to start inflating, or executing, the instruction corresponding to the marker. When the core controller receives an unimplemented instruction, the core controller waits for the predictor core to complete inflating its current set of trace steps. The core controller then halts the predictor core, modifies the state elements according to the result of the unimplemented instruction, and then restarts the predictor core on the next block of correctly predicted trace steps. In the case of a correctly predicted instruction, the predictor CPU will output the state updates into the output FIFO, but in the case of an unimplemented instruction, the inflated state updates will come directly from the input controller. Data in the output FIFO is streamed back to the host processor over STDIO.

On the host side, we introduce a software translation layer to convert the output of Zcompr in to the format required by the trace-driven simulator. This format is, itself, lightly encoded in a C-like binary format to simplify transmission to the host and processing on the host side. We choose to implement the final translation layer, which can be viewed as a sort of middleware, in software to allow rapid integration with a diverse set of trace-driven simulators.

### *B. Improved Trace Compression*

The original Zcompr algorithm does a good job of compression by capturing the majority of trace behavior in a manner that provides excellent local compression at the cost of increased, but parallel computation. Zcompr then relies on traditional entropy coding in the form of gzip to recover additional compression by discovering repetitions in the trace output, for example loops and function calls.

However, better compression can be achieved by recognizing that many of the unimplemented instructions in the trace have very similar behavior, in terms of state modification. Common modifications include increasing the program counter by four, repeatedly reading the same value (e.g. 0) from an unknown memory addresses, and writing the same value to a continuous region of memory. In these cases, the behavior of the unimplemented instruction *relative* to another unimplemented instruction can be summarized succinctly: the new unimplemented instruction can be described as a pointer into a memory containing previous unimplemented instructions. However, general entropy schemes are oblivious to these

structured relationships between instructions and suffer sub-optimal compression as a result.

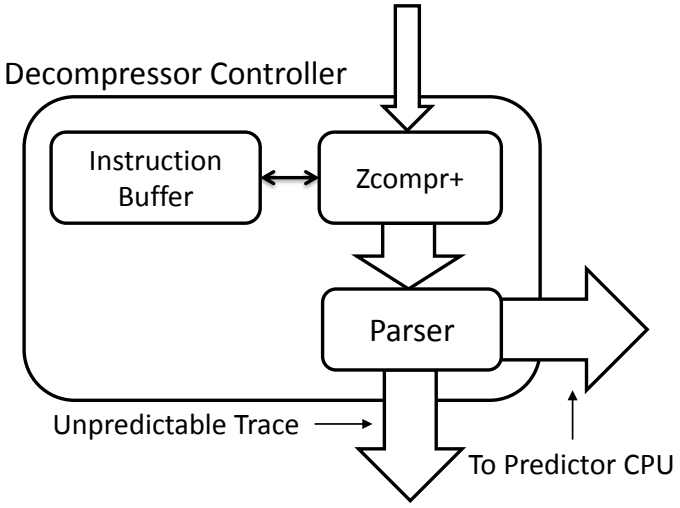



Fig. 4. Zcompr+, in the context of the Zcompr pipeline

In an ideal case, we would be able to maintain a full history of previously seen, unimplemented instruction to use for as a reference, as in the PDATs [11] compression algorithm. However, this sort of implementation is infeasible when trace lengths scale to multiple terabytes. Instead, we accomplish a similar, specialized entropy compression by maintaining a small cache of previously-encountered unimplemented instructions and their effects on system state. We then reduce "predictable" unimplemented instructions in the compress scheme to indices into this table. We call this scheme Zcompr+. Zcompr+ captures most of the benefit of PDATs-style compression on unimplemented instructions, without requiring a complicated and costly memory interface.

ZCompr+ introduces an interesting decision problem in the compressor: given a limited number of slots for unimplemented instructions, we must choose the best set to retain. The compressor tags these retained unimplemented instructions with a special flag, which causes the decompresser to update its table. Our current compression scheme uses a simple LRU policy in selecting which instructions to retain. It is likely that a better instruction selection algorithm in software could yeild substantially better compression.

## V. Results

### A. Experimental Setup

Our primary FPGA implementation platform in this study is the Nallatech ACP [17], a pair of Virtex 5 LX330T FPGAs that can be placed in a motherboard CPU socket along with other general purpose processors. Because the FPGAs can communicate over the front-side bus (FSB), interfacing to software has relatively low latency and high bandwidth. We consider this style of system to be a reasonable prototype of future tightly coupled FPGA-CPU systems, though we note that die-level integration would substantially improve communication latency and bandwidth.

The environment that we consider in evaluating ZIP-IO is a portion of an industrial batching system. The entire system is quite large, with thousands of machines, reflecting the need for enormous computational resources in the development and verification of modern processors. However, we have isolated our test machines within a single network and given them a dedicated, relatively local NFS store. Although this system may be slightly noisy, it permits us to set up much larger experiments that are more representative of an industrial use case. The system consists of several large, Ivy Bridge-class servers and some older Xeon machines with ACPs inside of them. All communication between programs is achieved through file system pipes.

The simulator we have chosen to evaluate in this paper is the Dinero IV uniprocessor cache simulator [9]. Dinero IV simulates a memory hierarchy consisting of multiple levels of cache and a main system memory, extracting useful statistics, such as hit and miss rate, for each cache. Because Dinero IV is a cache simulator, it requires only the memory access information from the decompressed trace information. Dinero can operate in excess of 5 MIPS on the Ivy Bridge machines under optimal conditions. Because Dinero is a relatively simple simulator, it serves as a useful limit study of the performance of our trace-based simulation systems, since it can consume even very fast trace streams before it becomes the system bottleneck. In our simulations, we drive Dinero with a mix of SPEC benchmark traces stored remotely on the network share, or locally in DRAM, depending on the system configuration.

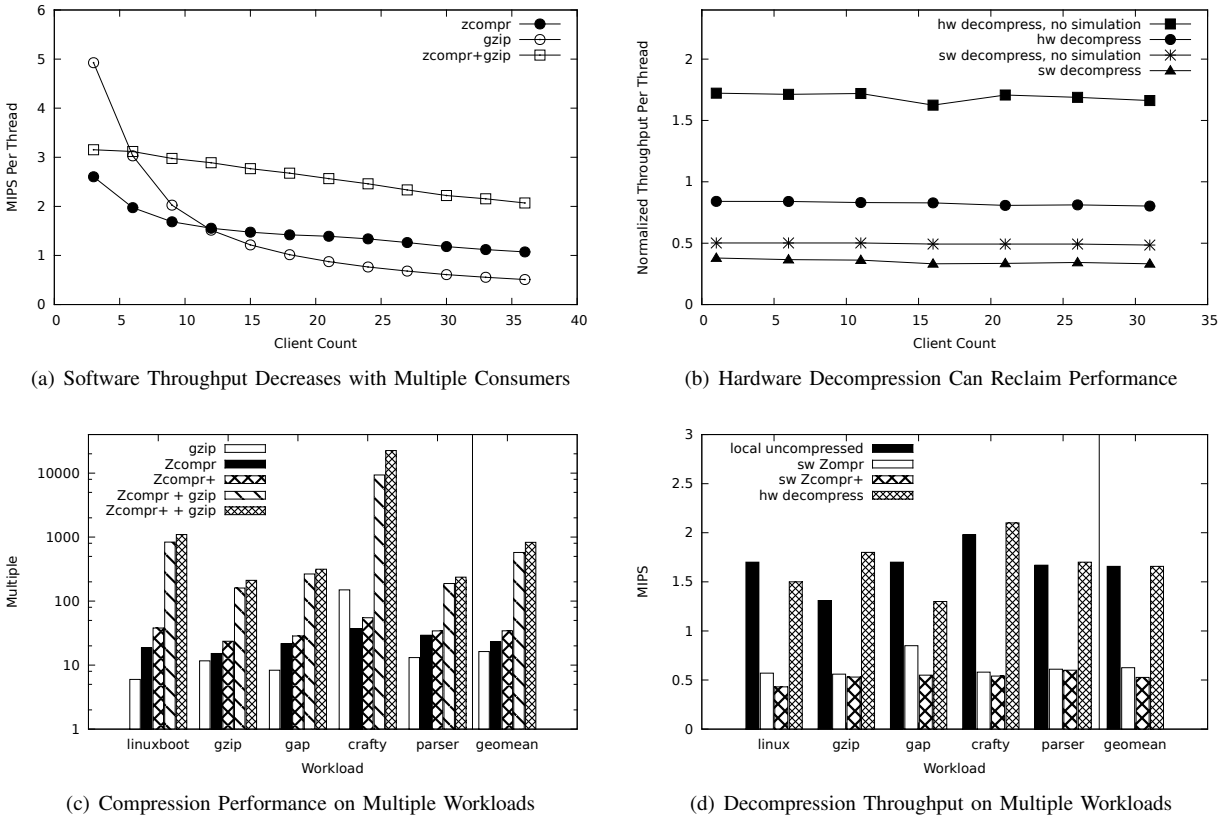
### B. Benefit of Compression

Figure 5(a) shows Dinero's average throughput as the number the number of simultaneous simulations scales for different compression scenarios. In general, performance degrades as the network store and network bandwidth are progressively overwhelmed by trace traffic. Uncompressed and gzipped traces require less computation to decompress and, as a result, have high throughput if there are relatively few consumers. However, because these schemes require more network bandwidth, they experience a steep decline in performance as the number of simulations increases. On the other hand, the combination of gzip and Zcompr+ has low single processor performance, due to the overhead of running the Zcompr+ algorithm. However, this scheme scales well beacuse it requires very little network traffic. Since industrial simulation workloads typically consist of hundreds or thousands of parallel simulators, ZCompr+ based compression is clearly a good choice because improved network bandwidth usage quickly outweighs the additional complexity of ZCompr+.

We also attempted to drive Dinero with remote, uncompressed traces. However, even a handful of simulators required hours to complete: uncompressed traces are simply infeasible in a production setting.

### C. Improvement to Zcompr

Figure 5(c) shows the performance of Zcompr and our augmentation, Zcompr+. Though relatively simple, Zcompr+




(a) Software Throughput Decreases with Multiple Consumers

(b) Hardware Decompression Can Reclaim Performance

(c) Compression Performance on Multiple Workloads

(d) Decompression Throughput on Multiple Workloads

Fig. 5. Performance Results

improves compression by 20% over the baseline Zcompr implementation. This relative advantage is maintained even after applying gzip to the compressed result, confirming that gzip is not able to extract the relative instruction behavior that Zcompr+ targets.

Figure 5(d) compares, among others, the throughput of ZCompr and ZCompr+. Zcompr+ achieves better compression, but at a cost of around 20% software throughput. This is another argument for the FPGA-based implementation: algorithmic complexity reduces overall throughput in software. An FPGA-based implementation of ZCompr+ suffer no such degradation.

### *D. Compression Performance*

Figure 5(c) shows the compression performance of the various compression schemes across different SPEC workloads. Stand-alone Zcompr has a compression performance comparable to gzip. However, Zcompr addresses compression scenarios, such as randomly mutating register values, that are not well-handled by gzip, and therefore composes with gzip to produce even better compression. Indeed, by composing the two compression algorithms, we achieve total compression of more than 1000x in some cases.

### *E. Decompression Throughput*

Figure 5(d) shows the throughput of the Dinero simulations across workload traces and with different compression schemes on the ACP platform. In an effort to normalize performance across silicon generations, we present results only from Xeon processor coupled with the ACP platform. As a limit study, we loaded a small trace in the RAM of the ACP machine.

On average, Dinero using ZIP-IO Zcompr has equivalent performance to Dinero running on uncompressed traces preloaded into DRAM, demonstrating that ZIP-IO is able to satisfy the bandwidth requirements of Dinero running on a Xeon server. Indeed, the hardware Zcompr implementation can sustain trace decompression rates of up to 15 MIPS, which is sufficient to accelerate Dinero even on the high-end Ivy Bridge servers.

Figure 5(b) shows the throughput of hardware-accelerated Dinero relative to a pure software implementation. Because the ACP must operate inside of a relatively old Xeon server as opposed to the newer Ivy Bridge servers, we normalize the performances in the graph against the speed of decompressing a small trace loaded on RAM, a performance upper bound. It is evident that using hardware decompression is effective in reclaiming most of the performance lost by using general purpose processor for decompression. We do not completely reclaim the performance of decompressing from DRAM due to non-idealities in the network.

The performance of the hardware-accelerated system is actually limited by the throughput of Dinero. If Dinero is removed, the hardware achieves nearly a 5x performance gain

| Module | *f*Max(MHz) | LUTs | Registers | BRAM |
|---|---|---|---|---|
| **Instruction Cache** | 100 | 771 | 342 | 3 |
| **Decompresser Control** | 100 | 1780 | 1105 | 58 |
| **Predictor CPU** | 115 | 2346 | 802 | 9 |
| **Total** | 100 | 10100 | 6969 | 99 |

TABLE I
RESOURCE USAGE FOR FPGA ZCOMPR IMPLEMENTATION, TARGETING XILINX VIRTEX-5.

over the software decompression. Although Dinero cannot make use of this bandwidth, other, less computationally intensive simulators, for example simulators doing statistical sampling [23], might be able to saturate the hardware bandwidth.

### *F. Hardware Implementation Requirements*

For ZIP-IO to be successful, interesting compression algorithms must be implementable in a reasonable area. Although we are currently using a V5LX330T for prototyping, this large FPGA is not necessary. Many interesting compression algorithms, including Zcompr can fit into a very small area. Table I gives a modular break down of the reconfigurable logic required for Zcompr. This resource usage is approximately 4% of the LX330T logic area, and 30% of the total memory, most of which is used for somewhat over-provisioned caches. This suggests that a future ZIP-IO would do well to have some kind of hardened memory interface.

In the throughput experiments discussed in Section V-B, we used modern Ivy Bridge processors, which have several cores per chip. Thus, a real deployment of ZCompr would need to support multiple simultaneous trace decompressions in the same fabric. There are two ways to extend our Zcompr hardware to handle this scenario. Zcompr itself could be re-architected to support GPU-style simultaneous multi-threading (SMT) for multiple on-going decompressions, while sharing a common memory subsystem among the threads. Alternatively, since Zcompr is so small, multiple instances could be laid out on a single fabric, again sharing a common memory subsystem. Our standard I/O implementation would make this sort of extension relatively straightforward – software could simply pass file handles to each of the Zcompr instances at runtime.

## VI. CONCLUSION

"Big Data" is a serious problem which must be addressed in future computer architectures. Application-specific compression is one means of dealing with the big data problem, provided that application-specific compression schemes can be well implemented in future platforms. In this paper, we presented ZIP-IO, a system architecture and libraries intended to accelerate these algorithms. We examined ZIP-IO in the context of trace-based processor simulation. However, the methods we employed implementing ZIP-IO can extend to many other workloads, including video, computational biology, and internet search, by way of our general Standard I/O interface.

As general purpose computers become increasingly heterogeneous, it is worth considering the integration of a fine grained reconfigurable substrate on-die. Application-specific compression, one means of mitigating the "Big Data" problem, is one workload that could benefit from such an integration. With tighter processor-fabric integration, the performance gains that we observe in this paper would be amplified, perhaps by as much as an order of magnitude. This acceleration is especially valuable in power-sensitive deployments, in which software capabilities may be severely constrained.

## REFERENCES


[1] [Online]. Available: http://asim.csail.mit.edu/redmine/projects/show/leap

[2] "Facebook newsroom." [Online]. Available: http://newsroom.fb.com

[3] "MIPS IV Instruction Set." [Online]. Available: http://techpubs.sgi.com/library/manuals/2000/007-2597-001/pdf/007-2597-001.pdf

[4] "Standard Performance Evaluation Corporation," *http://www.spec.org*.

[5] "The Secret Sauce: 0.5x Write Amplification," 2009. [Online]. Available: http://www.anandtech.com/show/2899/3

[6] K. C. Barr and K. Asanovi, "Branch trace compression for snapshot-based simulation," in *In International Symposium on Performance Analysis of Systems and Software*, 2006.

[7] G. Beskales, M. Fontoura, M. Gurevich, S. Vassilvitskii, and V. Josifovski, "Factorization-based Lossless Compression of Inverted Indices," in *Proceedings of the 20th ACM international conference on Information and knowledge management*, ser. CIKM '11. New York, NY, USA: ACM, 2011, pp. 327–332. [Online]. Available: http://doi.acm.org/10.1145/2063576.2063628

[8] I.-T. V. C. E. Group, "Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification," May, 2003.

[9] J. Edler and M. D. Hill, "Dinero iv trace-driven uniprocessor cache simulator." [Online]. Available: http://pages.cs.wisc.edu/ markhill/DineroIV/

[10] A. Jaleel, R. S. Cohn, C. keung Luk, and B. Jacob, "Cmp$im: A Binary Instrumentation Approach to Modeling Memory Behavior of Workloads on CMPs," Tech. Rep., 2006.

[11] Johnson, Eric E. and Ha, Jiheng and Zaidi, M. Baqar, "Lossless Trace Compression," *IEEE Trans. Comput.*, pp. 158–173, Feb. 2001.

[12] S. Kanev and R. Cohn, "Portable Trace Compression Through Instruction Interpretation," in *ISPASS*, 2011, pp. 107–116.

[13] C. Kozanitis, C. Saunders, S. Kruglyak, V. Bafna, and G. Varghese, "Compressing Genomic Sequence Fragments Using SlimGene," in *International Conference on Research in Molecular Biology*, 2010.

[14] S. Lee, J. Park, K. Fleming, Arvind, and J. Kim, "Improving Performance and Lifetime of Solid-state Drives Using Hardware-accelerated Compression," *Trans. on Consumer Electronics*, vol. 57, no. 4, pp. 1732 –1739, november 2011.

[15] Microsoft Corporation, "NTFS Technical Reference," 2011. [Online]. Available: http://technet.microsoft.com/en-us/library/cc758691%28WS.10%29.aspx

[16] Milenković, Aleksandar and Milenković, Milena, "An efficient single-pass trace compression technique utilizing instruction streams," *ACM Trans. Model. Comput. Simul.*, vol. 17, no. 1, Jan. 2007.

[17] Nallatech, "Intel xeon fsb fpga socket fillers," *http://www.nallatech.com/intel-xeon-fsb-fpga-socket-fillers.html*.

[18] A. Parashar, M. Adler, M. Pellauer, and J. Emer, "Hybrid cpu/fpga performance models," in *3rd Workshop on Architectural Research Prototyping (WARP 2008)*, June 2008.

[19] Red Hat Corporation, "JFFS2: The Journalling Flash File System," 2001. [Online]. Available: http://sources.redhat.com/jffs2/jffs2.pdf

[20] H. K.-H. So and R. W. Brodersen, "File system access from reconfigurable fpga hardware processes in borph," in *FPL*, 2008, pp. 567–570.

[21] H. K.-H. So and R. Brodersen, "Improving Usability of FPGA-Based Reconfigurable Computers Through Operating System Support," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, 2006, pp. 1 –6.

[22] R. Tremaine, P. Franaszek, J. Robinson, C.Schulz, T. Smith, M. Wazowski, and P. Bland, "IBM Memory Expansion Technology," in *IBM Journal of Research and Developement*, 2001.

[23] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," in *ISCA*, 2003.

[24] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE TRANSACTIONS ON INFORMATION THEORY*, vol. 23, no. 3, pp. 337–343, 1977.