Memory State Compressors for Giga-Scale Checkpoint/Restore

Andreas Moshovos
Electrical and Computer Engineering
University of Toronto
moshovos@eecg.toronto.edu

Alexandros Kostopoulos
Department of Informatics and Telecommunications
University of Athens

Abstract

We propose a checkpoint store compression method for coarse-grain giga-scale checkpoint/restore. This mechanism can be useful for debugging, post-mortem analysis and error recovery. Our compression method exploits value locality in the memory data and address streams. Our compressors require few resources, can be easily pipelined and can process a full cache block per processor cycle. We study two applications of our compressors for post-mortem analysis: (1) Using them alone, and (2) using them in-series with a dictionary-based compressor. When used alone they offer competitive compression rates in most cases. When combined with dictionary compressors, they significantly reducing on-chip buffer requirements.

1. Introduction

A *checkpoint/restore* or *CR* mechanism allows us to roll-back execution. In principle, CR allows us to take a complete machine state snapshot at any point during execution, continue to execute instructions, and then if necessary roll-back the machine's state to what it was when the checkpoint was taken. While originally proposed for supporting precise exception handling [19], similar mechanisms are now used to facilitate performance improvements via speculative execution [10,20].

Current CR mechanisms are fine-grain and relatively small-scale. A *fine-grain CR* mechanism facilitates restoring at very fine execution intervals, possibly at every instruction. The reorder buffer [19] and the alias table checkpoint FIFO used in MIPS R10000 [27] are examples of fine-grain CR mechanisms. The first allows recovery at every instruction while the second on speculated branches. Existing CR mechanisms are also *small-scale* since checkpoints are held only for instructions within the narrow execution windows of today's processors (e.g., around 100 or so instructions). CR mechanisms for larger windows in the range of a few thousand instructions were recently proposed [7, 11].

Recently, there have been several proposals for exploiting CR for other purposes such as online software invariant checking to aid debugging and runtime checking [17], to recover from hardware errors [18, 21] and *post-mortem* analysis [25] where checkpoints taken during runtime are used after a crash to replay the program's actions and determine what caused the crash. Common amongst these new CR applications is the need for checkpoints over large execution intervals. CR over a few seconds of real execution time is desired for post-mortem analysis, error recovery and debugging. With today's processing speeds this translates to several hundred millions or even billions of instructions. We will use the term *giga-scale* for CR mechanisms of this scope. While fine-grain, giga-scale CR mechanisms are in-principle possible, *coarse-grain* CR mechanisms appear more practical as they require a lot less storage (see Section 2).

We propose high-performance giga-scale, coarse-grain CR mechanisms that require significantly less resources than existing proposals. We focus on post-mortem analysis, however, the techniques we propose should be applicable to other CR uses. In giga-scale CR the key issues are: (1) the amount of storage required for checkpointing, and (2) the performance impact of taking checkpoints. We show that several megabytes are typically needed per checkpoint. Storing this amount of data on-chip is not possible today. Even if it was, it would not be the best allocation of on-chip resources. Saving checkpoints off chip impacts performance and cost in two ways. First, the checkpoint traffic can negatively impact performance because it increases pressure on the off-chip interconnect. Second, unless checkpoint data can be written immediately to off-chip storage, on-chip buffering is needed. The amount of on-chip buffering places an artificial limit on performance since execution must be stalled while the buffer is full and additional checkpoint data must be saved.

An obvious way of reducing off-chip bandwidth and storage demand is via on-line compression. Previous work focused on identifying what information should be checkpointed (program data and race outcomes) [18, 25]. Naturally, the question of how to best reduce the checkpoint size was a secondary issue and thus previous work relied on existing hardware dictionary-based compressors [29]. Such compressors require several millions of transistors and are comparatively slow [22]. As we show in Section 5, even under optimistic assumptions about their speed, large on-chip buffers are needed to avoid a significant performance hit.

In this work, we propose improved checkpoint mechanisms that require fewer on-chip resources while maintaining high-performance. We propose three *hardware, value-predictor-based compressors* that require very few resources and that can operate at the same frequency as the processor's core. We consider two ways of using these compressors: (1) Used alone as a low cost on-chip compressors, and (2) used in-series with previously proposed hardware dictionary-based compressors. We demonstrate that when used alone our compressors offer competitive compression ratios when compared to the much more expensive dictionary-based compressors. However, a key result of our work is that combining our compressors with a slower dictionary-based compressor is *always* (for the programs we studied) better performance-, compression- and resource-wise than using a stand-alone dictionary-based compressor.

Our contributions are: (1) We study the checkpoint resource and performance overheads of typical programs for various checkpoint intervals. (2) We propose cost-effective, high-performance hardware compressors that exploit value predictability. (3) We analyze the performance and resource overheads of giga-scale checkpointing for dictionary- and value-predictor-based compressors. (4) We compare with the frequent value compressor, a state-of-the-art complexity-effective memory compressor [2,3]. To the best of our knowledge this is the first work (an initial investigation of the ideas presented in this work appears in [15]) that: (1) takes a detailed look at the hardware/performance overheads of checkpoint compression for giga-scale checkpointing, and (2) proposes hardware, value-predictor-based compressors for checkpoint storage reduction.

We show that with an on-chip buffer of just 1K bytes a combination of our compressor with a dictionary-based compressor results into an overall average performance slowdown of just 1.6%. Moreover, this combination reduces checkpoints to 34% of their original size. Even when used with a 64Kbyte on-chip buffer, the dictionary-based compressor alone incurs an overall performance slowdown of 3.7% and reduces checkpoints to 38% of their original size. The worst performance slowdown is 4.4% for our compressor and 11% for the dictionary-based compressor alone. When used alone, our compressor reduces checkpoint storage to 52% of its original size. While not as good as dictionary-compression this reduction is possible with very few resources (a dictionary-based compressor requires millions of transistors while our compressor a few thousand). All aforementioned results are for a checkpoint interval of 256 million instructions.

Burtscher and Jeeradit proposed *software* value-prediction-based compressors for programs traces [6]. In Section 4 we explain that vastly different cost, time and complexity trade-offs apply in our target application for two key reasons: (1) we are interested in *hardware* compressors, and (2) the input data in our case is only a subset of the whole program trace hence there is no guarantee to observe high levels of value locality.

The rest of this paper is organized as follows: In Section 2, we explain the resource and performance trade-offs that apply to giga-scale CR. In Section 3, we explain the principle of operation of three high-performance, value-prediction-based compressors. In Section 4 we review related work. In Section 5, we present our experimental analysis. We first look at the resource requirements of giga-scale CR. We then analyze the performance and resource requirements of dictionary- and value-predictor-based compressors. We summarize our work in Section 6.

2. Giga-Scale Checkpoint/Restore

Figure 1 shows an example of CR at work. At some point during execution the checkpoint mechanism is invoked and a new checkpoint is created. Execution then continues. At some later point, we decide that execution should be rolled back to the checkpoint. Using the checkpoint the machine state is restored to the values it had at the time the checkpoint was taken. The machine state comprises the register file and memory values, plus any internal state for the other system devices. In this work, we focus only on registers and memory because no standard semantics exist for device reads and writes. For some devices it may not be possible to do CR as some side-effects may be irreversible. We believe that the most meaningful approach to providing CR for devices is to define a clear set of APIs that convey information about the action taken and if and how it can be reversed when need. This investigation is beyond the scope of this paper.

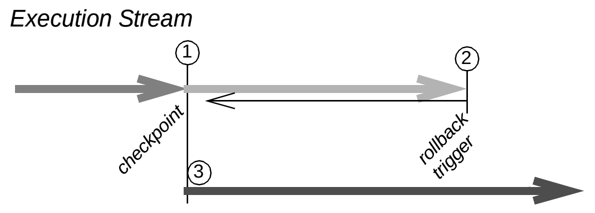


Figure 1: *What Checkpoint/Restore does (see text).*

Being able to restore the machine state at every instruction while possible in-principle is not very practical for giga-scale CR. Doing so would require storage proportional to the number of instructions executed. Accordingly, we restrict our attention to coarse-grain CR mechanisms. In giga-scale CR, the space required for saving the complete register file contents is negligible compared to that required for storing memory state. For this reason, we save a complete image of the register file with every checkpoint.

Lets take a closer look at what is required to checkpoint memory state. We can observe that at any given point the complete memory state is available. Hence, a possible solution to rolling-back memory state would be to keep track of the *changes* done to memory (this is equivalent to the history file method for fine-grain, small-scale CR [19]). Only the first write to each memory location needs to be checkpointed. This is because we do not care to rollback to any intermediate execution point. Another important consideration is the storage granularity at which checkpoints are taken. Memory writes can be of varying granularity. In most modern processors, a memory write can update a byte or up to eight bytes. If we wanted to keep a precise record of all updates we would then need to keep records at the smallest possible granularity. This would impose a high storage overhead. Instead, it is practical to keep records at a much larger granularity.

Using a cache block is convenient, since the contents of cache blocks are readily available on-chip and could be read or written simultaneously (we assume one sense amp per bitline pair). Because of spatial locality, chances are that most data saved this way will indeed be updated later on. In this work, we assume a cache block of 64 bytes without loss of generality. We also assume four byte *words*. Restoring machine state is fairly straightforward. We use the register file image to restore register file contents and the records of memory updates to restore memory values. For post-mortem analysis restoring can also be done in software. Accordingly, we do not consider decompression latency further. There are applications, however, such as debugging where decompression latency is important. Our compressors offer a simple and fast decompression path.

2.1. A Checkpoint/Restore Architecture

Because the checkpoints are large (several megabytes is typical) on-chip storage is presently either not an option or not a good one. An obvious choice is to use main memory to hold the checkpoints. Compressing the checkpoint stream on-chip reduces off-chip bandwidth and memory footprint overheads. Figure 2(a) shows a complete checkpointing architecture. A checkpointed cache block is first placed into an internal buffer (in-buffer). This buffer feeds an on-chip compression engine. Compressed records are placed on another buffer (out-buffer) where they compete with regular program accesses for main memory bandwidth and storage.

2.2. Performance, Resources and Complexity

Several performance and resource considerations exist with giga-scale CR mechanisms: (1) The on-chip compression engine and the associated buffers represent a resource overhead. (2) Whenever the in-buffer is full and a new block has to be checkpointed we have to stall the processor. Thus the on-chip buffers and the speed with which the compression engine can process checkpoint data can artificially reduce performance. (3) The compression ratio achieved also impacts performance since it is directly related to the demand placed on the main memory interconnect.

Complexity considerations also exist and they can impact performance. The compressor's output stream is placed on the "out-buffer" for writing into memory. Using the storage of the "out-buffer" effectively requires *aligning* the compressed data into a continuous stream. Writing to memory requires forming reasonably sized blocks of data. It follows that between the compressors and the "out-buffer" there should be an alignment network whose purpose is to shift the compressed data into place just after the previously written block within the "out-buffer". Thus it is preferable to use compressors that reduce the number of possible alignments. For example, from this perspective, a compressor that always writes at least a word is preferable over a compressor that may produce fewer bits and hence higher compression.

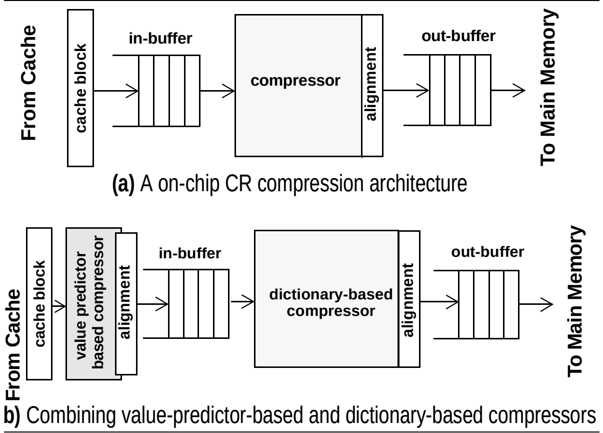

From Cache
cache block
in-buffer
compressor
alignment
out-buffer
To Main Memory
(a) A on-chip CR compression architecture
From Cache
cache block
value predictor based compressor
alignment
in-buffer
dictionary-based compressor
alignment
out-buffer
To Main Memory
b) Combining value-predictor-based and dictionary-based compressors

Figure 2: *(a) A checkpoint/restore architecture with an on-chip compressor. We show only the checkpoint path since restoring is trivial. Cache blocks are placed into a buffer before the first write occurs. The compressor processes blocks and places its output into another buffer. Whenever possible, the compressed blocks are written to main memory. (b) We study two different organizations that use a value-prediction-based compressor. The first uses the compressor alone as in part (a). In the second, the value-prediction-based compressor is placed in-series with a dictionary-based compressor. A buffer in front of the value-prediction-based compressor is not needed since we are capable of processing a full cache block per cycle.*

2.2.1. Hardware Dictionary-Based Compression. Previous work in giga-scale CR assumes an on-chip LZ77-like dictionary-based compression engine [25]. Such compression engines offer high compression and have been already built in hardware. However, they are relatively expensive and slow. To avoid stalling the processor it will be necessary to use larger buffers and hence there is an indirect increase in overall resource cost. There have been several hardware implementations of dictionary-based compressors. The most relevant for our purposes is IBM's MXT memory compressor [22] which has been implemented using a 0.25 micrometer process into a chipset operating with a 133Mhz clock [23]. It utilizes about one million gates and it is capable of processing four bytes per cycle. This implementation would require 17 cycles to process a checkpoint record for a 64 byte cache block assuming 37 bit physical addresses. As we show in Section 5.6, even with a 64K on-chip buffer a performance hit in excess of 10% is incurred even with a twice as fast compressor.

Two ways of improving dictionary-based compression speed would be to process more bytes in parallel or to use a higher frequency. Franaszek *et al.,* have shown that it is possible to process more bytes in parallel by splitting the dictionary [9]. To avoid an abrupt decrease in compression rates it is necessary to update all parts of the dictionary with all bytes that are processed simultaneously. In general, to process N bytes simultaneously they split the dictionary into N equal parts, where each part is implemented as a CAM with one lookup port and N write ports. In addition, processing more bytes in parallel requires a more complex shuffling network at the compressor's output in order to align the outputs of all sub-parts into a continuous record. The MXT implementation relies on four 256-entry CAMs operating in tandem [23]. Every cycle, each one sees a probe, plus four updates (one from itself and three from the other CAMs). It would be challenging to operate this compressor at several Ghz or to pipeline it.

In this work we ask the following questions: (1) Can we get most of the compression benefits with a much simpler compressor? (2) Can we have a compressor that is fast enough to avoid stalling the processor while requiring little on-chip buffering?

3. Simplifying Compression Hardware

It is well known that many programs exhibit value locality in their memory stream. While this property has been demonstrated for streams that consist of all memory accesses, As we demonstrate in Section 5.3 this property holds also for the first updates to each memory block after arbitrary chosen points during execution. Exploiting this property and rather than storing memory blocks verbatim we propose *hardware, value-prediction-based compression.* In this technique we opt for a two-level representation for checkpointed cache blocks. For each four byte *word* (could have been some other quantity) we first record whether it can be successfully predicted by a preselected predictor. Then, we store only those words that could not be predicted.

We have experimented with various prediction structures and present three alternatives. Figure 3 illustrates the simplest alternative where 16 single-entry value predictors are used for block data and a single-entry stride predictor for block addresses. This way all 16 words within a 64 byte block can be processed in parallel. Each cache word is processed by a different predictor and the mapping of words to predictors is static. As explained in Figure 4, the final representation (checkpoint record) of a block consists of a header, and possibly an address and up to 16 words. Whenever a new checkpoint starts, all predictor entries are zeroed out to avoiding having to checkpoint the predictor's state. Restoring the machine's state is straightforward as long as the checkpoint records are processed in the order they were saved. We simply pass them through the predictor and use the data they contain or the value provided by the predictor as instructed by the headers. Figure 5 shows an example of how compression works.

In order to save the checkpoint record into memory we need to align the header and any words that follow into a continuous block. This is straightforward via a barrel shifter. Since there are 16 words, four stages should be sufficient to place all words in order. The header and the single byte header for the address are always saved, hence we need to also make room for the address in case this is needed. It follows that we should be able to form the continuous checkpoint block in five stages. More

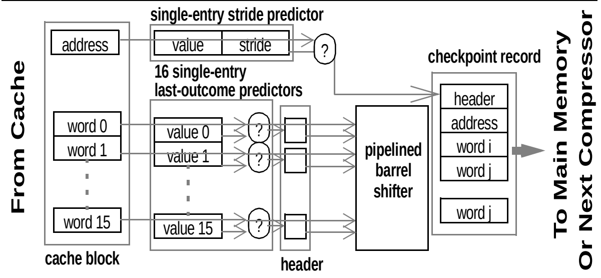


Figure 3: *Last-outcome based compressor. Each of the 16 words of the cache block are passed to a different single-entry last-outcome value predictor. A 16-bit header vector identifies the words that are predicted correctly. Incorrectly predicted words are stored explicitly. Similarly, a single-entry stride predictor is used for recording block addresses. Only if the address is not predicted correctly we need to save it. Instead of using an additional bit to record whether the address is predicted or not we use a full byte to reduce the possible alignments for the final checkpoint block. A pipelined shuffle network is used to place the word that were not predicted into place so that the header and the data words appear as a continuous block. Since there are 16 words in the block four stages should be sufficient for the shuffle network. An additional stage is needed to align the address into place.*

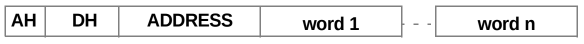
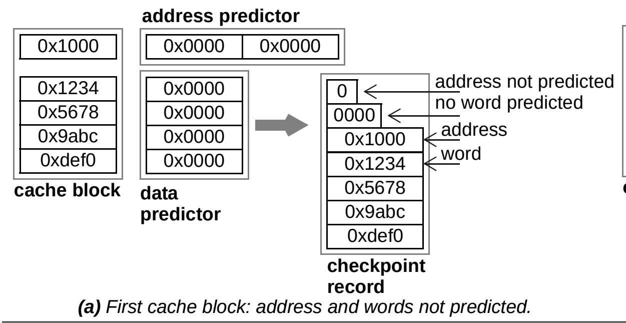


Figure 4: *Checkpoint record format. "AH" or address header is a single byte which is non zero if the address can be predicted. "DH" or data header is a 16-bit vector with one bit per word in the original cache block. A bit value of 1 indicates that the corresponding word can be predicted. The optional "ADDRESS" field exists only when the address cannot be predicted. Each bit indicates whether the corresponding word was predicted correctly. "WORD i" optional fields contain the values of those words that cannot be predicted. Having all header bits first allows easy decoding of the record during decompression.*

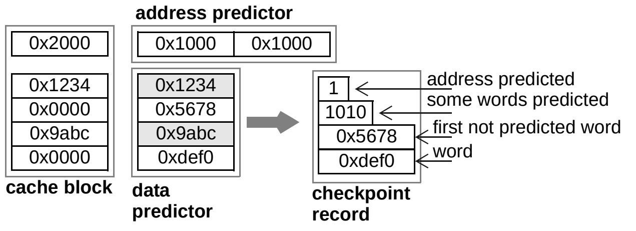
importantly, this process can be pipelined thus maintaining high performance. A simple alignment network will also be needed to align the continuous block into position for writing it into the on-chip buffers (this is required for any compressor). We expect this compressor to operate at the same frequency as the processor core since: (1) all cache words are processed simultaneously, (2) there are no intra-block dependences, and (3) prediction amounts to a few simple operations (e.g., comparison, selecting the output value and updating a single-entry predictor). Overall, our compressor requires few resources: For the value predictors we need 16 word entries and 16 comparators; For the address predictor we need two words, an adder, a subtractor and a comparator; We also need a five stage barrel shifter capable of handling about 18 words in total plus a control unit.

3.1. Combined Value-Predictor-Based Compressors

We have experimented with other predictors that offer a trade-off between cost and compression ratio. We discuss



(a) *First cache block: address and words not predicted.*



(b) *Second cache block: address and some words predicted and not saved*

Figure 5: *An example of value-predictor-based compression. In the interest of space and without a loss in generality we assume four word cache blocks where addresses and words are all 16 bits long. (a) Initially, the predictors are zeroed out. The first write to block 0x1000 occurs. Since no value is predicted correctly, the resulting checkpoint record contains all words plus the header. (b) The value and address predictors now have the values calculated while predicting the previous cache block. The next write is to block 0x2000. The address and some words are predicted correctly. The checkpoint block now contains the header and only those words that were not predicted correctly.*

two of these predictors that performed fairly well and that attempt to exploit common data patterns. We use two header bits per cache word so that we can encode four possibilities. The “00” combination is used for “not predicted”, the “01” for “predicted via last-outcome”, and the “10” for “predicted by a previous-to-last-outcome predictor” (i.e., we use a predictor with history depth of two). In the *combined neighbor* predictor a “11” header value indicates that the word value matches that of the immediately preceding word within the cache block. For the first word within a cache block we do not use this prediction mode to avoiding losing information[1]. The motivation for this compressor comes from data that represents physical quantities such as pixel values for images. It is common for adjacent pixels to hold the same value. There are other reasons why adjacent memory values are often equal. The neighbor predictor attempts to exploit locality *within* cache blocks in addition to locality *across* blocks that is captured by its last-outcome component. In the *combined stride* predictor the “11” combination indicates that the value can be predicted by a stride predictor. It is well known that many data values exhibit this behavior. The neighbor predictor requires three times as many comparators compared to the last outcome predictor and a bypass network for communicating values within the block. The latter can be easily pipelined. The combined stride predictor has considerable additional cost as it requires an additional subtractor, adder and comparator per predictor entry.

3.2. Support Mechanisms

A mechanism is needed for identifying the first update to each memory block. An impractical yet easy to understand solution would be to have a large bit vector with a bit per memory block to remember whether this block was checkpointed. When we want to take a new checkpoint we reset all bits. A practical implementation

[1] Think of a cache block where all words have the same value. In this case, all words match the one preceding them. However, we need to save the value in order to be able to recreate the block. By disabling this prediction mode for the first word we make sure that the word is either saved explicitly or that it can be predicted by the other predictor components.

Table 1: ***The three value predictor based compressors we studied.***

Header Value	Meaning
LAST OUTCOME (LO)	
0	not predicted/stored explicitly
1	predicted/not stored
COMBINED NEIGHBOR (CN)	
00	not predicted/stored explicitly
01	Last-outcome predicted/not stored
10	Predicted by previous to Last-Outcome/not stored
11	Same as preceding word in block/not stored
COMBINED STRIDE (CS)	
00	not predicted/stored explicitly
01	Last-outcome predicted/not stored
10	Predicted by previous to Last-Outcome/not stored
11	Predicted by Stride Predictor/not stored

splits the vector into chunks so that they can be saved in the page table. This way the appropriate vector chunk will be readily available any time a write occurs (the TLB miss would bring in the vector chunk too). Assuming 4K pages and 64 byte blocks, a 64 bit vector per page is sufficient. Even then, clearing all bit vectors every time a new checkpoint starts would be expensive. A simple solution introduces an additional 64 bit tag per page table entry and a global checkpoint counter. Every time we take a new checkpoint we increase the global checkpoint counter. Every time we update a page table vector chunk we set its 64 bit tag to the current value of the global checkpoint counter. A vector chunk is valid only when its tag matches the global checkpoint counter. Using 64-bit counters and even if we assume 1K checkpoints per second, approximately 571 million years will be needed for the counter to wrap around. Other alternatives exist, e.g., using smaller counters and clearing the vectors every couple of hours or using bloom filters but their investigation is beyond the scope of this paper. Second, since checkpoints are stored in main memory, a mechanism is needed to allocate the appropriate space. For this purpose we can leverage the existing virtual memory API. In this case, checkpoint store appears as part of the application’s memory space and we have to ensure that there are no conflicts with program data. Alternatively, via operating system support we could use another memory space for checkpoint storage. There several open questions in this area that should be addressed. However, further investigation of this topic is beyond the scope of this paper.

4. Related Work

Related work can be classified in mechanisms for supporting CR, techniques that utilize CR for a purpose other than speculative execution, hardware compression methods and work on value-prediction-based compression. Burtscher and Jeeradit proposed using value prediction for compressing program traces in software [6]. In this work we are concerned with hardware-based compressors sharing the goal of quick compression but having the additional constraint of using few, low complexity hardware resources. The cost and the complexity of data alignment and value prediction are vastly different in hardware than in software. Moreover, the value locality potential is different in our application as we consider only a subset of the memory stream at the cache block level.

Existing fine-grain, small-scale CR mechanisms are variations of designs that were proposed for supporting precise exceptions and out-of-order, speculative execution [10,19,20]. Recently there has been work in extending these mechanisms for scheduling windows of several hundreds [11] or several thousands of instructions [7]. There has also been work in reducing the number of global, fast checkpoints needed for frequent recovery actions such as branch mispredictions [1,14]. Besides supporting speculative execution recent proposals rely on CR mechanisms for other purposes. Specifically, Oplinger and Lam suggest using CR for on-the-fly software-based invariant checking [17]. Zhou *et al.* propose hardware-based monitors for debugging purposes [28]. Both aforementioned works rely on extensions of thread-level speculation mechanisms for checkpoint and recovery. Sorin *et al.* propose SafetyNet, a CR mechanism for long-latency fault detection and recovery. SafetyNet relies on relatively large on-chip buffers (512K) for holding checkpoints and is targeted at much shorter checkpoint intervals (i.e., less than a millisecond) than giga-scale CR. Prulovic and Torrellas proposed ReEnact a system for recovering from data races in multiprocessors [18]. ReEnact is also targeted at relatively short intervals. Xu, Bodik and Hill describe a CR mechanism for post-mortem analysis [25]. Their focus is on recording data races with low overhead and assume a hardware-based LZ77 compressor for reducing checkpoint storage. We build on previous work and propose a fast compressor that exploits value locality. In principle, all aforementioned techniques could benefit from our compressor as using it would reduce on-chip buffer requirements. Narayanasamy *et. al.* propose mechanisms for deterministic playback for the purposes of debugging [16].

Several hardware compressors for memory data have been developed. IBM's MXT technology [22] relies on a variation [9] of the LZ77 algorithm [29] to compress 1K cache blocks. We demonstrate that simpler mechanisms can offer most of the benefits with a lot less cost and at a much faster speed. Other memory compression techniques have been proposed [2,4,8,12,13] that exploit locality within a cache block or frequent bit patterns (such as sequences of zeroes or ones). To do so, most of these implementations process the block serially. A technique that exploits frequent values for compressing cache data has been proposed by Yang, Zhang and Gupta [26]. It too processes the block serially. Our compressor is much simpler than these methods. Alameldeen and Wood proposed Frequent Pattern Compression (FPC), a relatively fast compressor for on-chip caches [3]. We consider FPC in Section 5.4.

5. Evaluation

In Section 5.1 we present our methodology. In Section 5.2 we study the checkpoint storage requirements for various checkpoint intervals and show that often several megabytes of storage are required. In Section 5.3 we show that our compressors can reduce checkpoint storage requirements significantly. In Section 5.4 we compare our compressors to the Frequent Pattern Compressor [2,3]. In Section 5.5, we study dictionary-based compression and show that often our compressors offer similar compression ratios. We also show that higher overall compression is possible when our compressors are used in-series with a dictionary-based compressor. In Section 5.6, we demonstrate that the combination of our compressors with a dictionary-based compressor significantly reduces on-chip buffer requirements.

5.1. Methodology

We extended the Simplescalar v3.0 simulators [5]. We used the functional simulator to study value locality, checkpoint requirements and compressibility. We used the timing simulator to determine the performance impact of various compressors under varying assumptions about on-chip buffers and compressor speed. Our base configuration is an aggressive wide-issue, 8-way dynamically-scheduled, deeply pipelined superscalar processor. We modified Simplescalar's main memory system to appropriately model bus contention. Table 2 depicts the base processor configuration and other key simulation parameters. We used the following integer or floating-point benchmarks from the SPECCPU 2000 suite: *164.gzip, 174.parser, 176.gcc, 177.mesa, 181.mcf, 183.equake, 188.ammp, 197.parser, 254.gap, 255.vortex, 256.bzip2 and 300.twolf.* We used a reference input dataset for all benchmarks except mcf and parser. For the latter two we used a training input dataset since it was not possible to allocate sufficient memory in our systems to simulate them with a reference data set. The binaries were compiled for the PISA architecture using GNU's gcc and g77 v2.7 (options: "-O2 -funroll-loops -finline-functions"). For the functional simulations we simulated

Table 2: ***System parameters.***

Branch Predictor	16k GShare+16K bi-modal 16K selector 2 branches per cycle
Instruction Window	256 entries
Issue/Decode/Commit	any 8 instructions / cycle
DL1/IL1 Geometry	64Kbyte/64-byte blocks/4-way SA
L1/UL2 Latencies	3/16 cycles
Fetch Unit	Up to 8 instr. per cycle 64-entry Fetch Buffer 2 branches per cycle Non-blocking I-Cache
Load/Store Queue	64 entries, 4 loads or stores per cycle Perfect disambiguation
FU Latencies	same as MIPS R10000
UL2 Geometry	1Mbyte/64byte blocks/8-way SA
Stage Latencies	8 cycles from branch predict to decode stage 6 cycles for decode and renaming 6 cycles from writeback to commit 10 cycles branch misprediction penalty.
Instructions Simulated	80 Billion max. functional Five Billion timing
Input Data Set	Training for mcf and parser Reference for all others
Checkpoint Intervals	64M, 256M, 1B and 4B instructions
On Chip Buffers	1K, 4K, 16K and 64K bytes.
Main Memory	Infinite capacity, 100 cycles latency 16 Banks, one port/bank 64-byte interleaved
LZW dictionary size	1K, 4K, 16K or 64K
LZW consumption rates	from 32 bytes/cycle to 4 bytes per 16 cycles

up to 80 billion instructions or to completion (whichever came first). In order to obtain tractable simulation times under timing simulation we simulated five billion committed instructions after skipping five or ten billion instructions in order to skip the initialization. Instead of using the less effective LZ77 [29] compressor we opt for an LZW implementation [24].

In all our simulations we ignore the first checkpoint to avoid artificially skewing our results towards higher compression ratios. This is necessary since most memory blocks initially are zeroed out in Simplescalar. We verified that the compression rates did not decrease had we skipped more checkpoints. We do not report statistics for the 16 billion checkpoint interval for gcc, mcf and parser since they execute less than 32 billion instructions. For the latter two it was also not possible to collect statistics for the four billion instruction checkpoint interval.

5.2. Checkpoint Storage Requirements

We study the maximum checkpoint storage requirements for a single checkpoint as a function of the checkpoint interval to show that on-chip storage is not sufficient for storing even a few checkpoints. Figure 6 reports the checkpoint storage requirements for checkpoint intervals of 1K through 16 billions of instructions. We report the base two logarithm of the checkpoint size in bytes. Note that a whole cache block is checkpointed prior to the first update. This explains why for example about 1K or more are needed for the 1K interval. Generally, checkpoint requirements increase almost linearly with the checkpoint interval. For some programs (e.g., amp, gcc and twolf), the

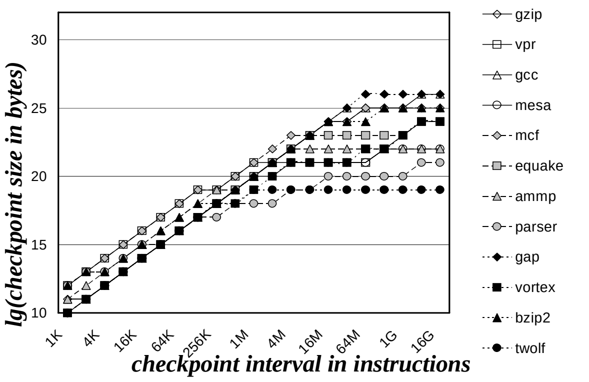
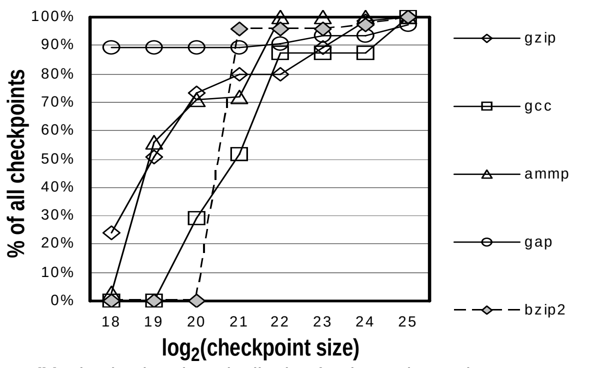


Figure 6: *Checkpoint storage requirements as a function of the checkpointing interval: 1K through 16Billion instructions.*

checkpoint requirements level off after a few million instructions. Still, several megabytes are needed for all programs except twolf and for the larger checkpoint intervals. In the rest of this work we will focus on checkpoint intervals of 64 million instructions or more.

Figure 7 shows the cumulative distribution of checkpoint sizes for few representative applications. These results show that while the maximum checkpoint may require several megabytes of storage, there can be great variation in checkpoint sizes depending on the application. For example, the maximum checkpoint in gcc requires about 32Mbytes, however, 50% of all checkpoints require 2Mbytes or less. From these results it follows that the checkpoint requirements of most programs are relatively large and hence techniques for compressing checkpoints are worth investigating. Furthermore, on-chip storage of even a single checkpoint would require several tens of megabytes for some applications.



(b) Checkpoint Size Distribution for the 1B interval

Figure 7: *Cumulative checkpoint size distribution per benchmark and for the 1B instruction checkpoint interval. For clarity we show a few representative applications.*

5.3. Value-Prediction-Based Compressors

We study the compression possible with the last-outcome compressor, the simplest amongst those we proposed. Figure 8(a) reports the overall *compression ratio* observed with this compressor for various checkpoint intervals. We consider the total storage required by *all* checkpoints combined and report the ratio of the storage required with our compressor over the storage required without compression. Hence the lower the compression ratio the higher the compression. This measurement includes the compression headers in addition to data and addresses whenever needed. A compression ratio of 10% means that we need only 1/10th of the original space. On the average, compression ratios of 60% to 54% are possible. The best compression ratio of 14% is observed for gcc for the 1B checkpoint interval. The worst compression ratio of 88% is observed for vpr and for the 64M checkpoint interval. In general, the compression observed does not vary monotonically with the checkpoint interval. The amount of compression possible depends on how predictable the data words and the block addresses are. The prediction rates of the underlying value and address predictors are shown in figures 8(b) and 8(c) respectively. For most programs, data prediction rates either remain relatively unchanged or tend to increase for larger checkpoint intervals. However, prediction rates decrease with larger intervals for some applications (e.g., equake). The prediction rate depends on the amount of value locality that exists amongst the data that is updated during a checkpoint. As we have seen in Section 5.2, in some applications the amount of data per checkpoint increases with the checkpoint interval. Value locality may be lower or higher in this larger data set. Block address prediction rates tend to increase with the checkpoint interval for all applications except gap.

5.3.1. Combined-Predictor-Based Compressors. We report the compression ratios possible with the two combined-predictor-based compressors of Section 3.1. In the interest of space we restrict our attention to the 256M and 1B checkpoint intervals. Figure 9 reports the compression ratios for the last-outcome (LO), combined neighbor (CN) and combined stride (CS) compressors. Sometimes the differences amongst them are negligible. The CN is always better than the LO and in mesa, vortex, gap and twolf the improvement is relatively significant. The CS compressor is better than CN only for vpr and somewhat better than LO for most applications. Thus, compared to LO or CN, CS does not offer benefits that would justify its much higher cost (adders and subtractors). On the other hand, CN offers noticeable compression improvements over LO at a small additional cost.

5.4. Frequent Pattern Compression

Frequent pattern compression (FPC) exploits frequent occurring sub-word patterns and zero word runs to compress L2 blocks [2]. With appropriate modification, the same compressor could also be used for gigascale CR. Accordingly, we compare its compression performance to the LO and CN compressors. The results are shown in Figure 10 for the 256M and 1B checkpoint intervals. On the average, FPC performs slightly better (2%) than LO and slightly worse (4%) than CN. Compression performance varies significantly per program. This is not unexpected since FPC exploits different behavior than the other compressors. FPC is slightly better than CN for vpr (3%), gap (2%), parser (1%), vortex (7%) and twolf (5%). (Shown in parentheses are approximate differences for the 1B checkpoint interval.) CN is slightly better than FPC for gzip (9%), gcc (5%), mcf (15%), equake (13%), ampp (6%) and bzip2 (4%). Thus, focusing on compression ratio alone there is no compelling reason to chose one compressor over the other. Complexity-wise and ignoring the fixed-length headers, FPC requires a more powerful alignment network since a four byte word can be compressed to one, two, three or four bytes. The number of permutations is higher if we consider zero-runs (our results include zero run compression for FPC). While simplifications may be possible a straightforward way of accounting for zero runs is to assume that each word can be represented in five possible ways: no byte, one byte, two bytes, three bytes or four. Thus there are 5^{16} permutations per 16 word block for FPC[1]. CN represents a word using either zero or four bytes and thus there just two options per word and thus there are 2^{16} permutations per block. Accordingly, we conclude that: (1) there is no reason to dismiss the CN predictor, and (2) that given that the alignment network for CN is less demanding it is important to further consider the CN predictor as it may lead to simpler compression hardware. The results of this section serve also as motivation for studying a possible combination of FPC and CN. However, this study is beyond the scope of this work.

5.5. Dictionary-Based Compression

We study the LZW compressor which offers higher compression compared to the LZ77 variants implemented in hardware we reviewed in Section 4.. A key design parameter for dictionary-based predictors is the size of the dictionary. Using larger dictionaries typically results in higher compression. However, larger dictionaries imply higher cost and slower operation. IBM's MXT uses a 1K byte dictionary. In Figure 11(a) we report the compression ratios possible with a 64K byte dictionary for checkpoint intervals of 64M through 16B instructions. We use a larger dictionary since we are interested in demonstrating that our compressors offer competitive compression ratios. In most cases, compression is higher compared to our compressors. However, in equake LZW fails to reduce the amount of storage required. This suggests that while there is some repetition in the memory values rarely there are repeating patterns of more than one value. Alternative encoding can be used to avoid inflation [22, 23]. In Figure 11(b), we show that indeed in most cases, using larger dictionaries results in higher compression. Shown are the compression ratios for LZW compressors with 1K (10 bits), 4K (12 bits), 16K and 64K dictionaries for the 1B checkpoint interval.

[1] Note that this analysis is for our application of compressing whole program checkpoints. FPC was originally proposed for L2 block compression and there the minimum amount of space used per compressed block was 8 bytes. This significantly reduces the number of relevant alignments. In our case, we do care to reduce the input block to the minimum possible size hence we do have to consider all possible permutations.

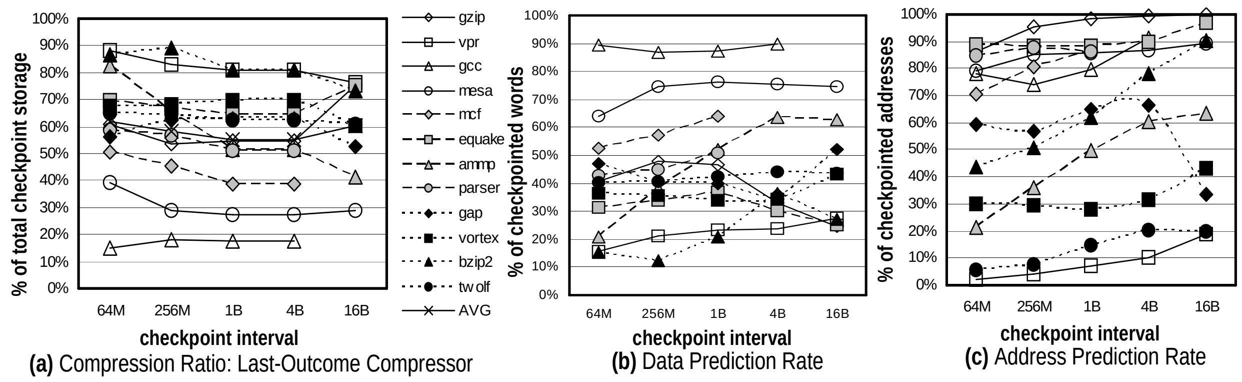


Figure 8: *Last-outcome-based compressor. (a) Overall compression ratio for various checkpoint intervals. Reported is the ratio:* $\text{Total\_Checkpoint\_Storage}_{\text{after-compression}}/\text{Total\_Checkpoint\_Storage}_{\text{without-compression}}$. *Lower is better. (b) Prediction rates for data words. Higher is better. (c) Prediction rates for address blocks. Higher is better.*

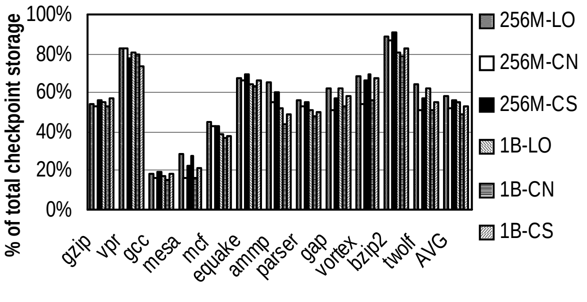


Figure 9: *Compression ratios with three value predictor based compressors and for the 256M and 1B instruction checkpoint intervals. Shown are the last-outcome (LO), combined neighbor (CN) and combined stride (CS) predictors. Bars are labeled as I-P where I is the checkpoint interval (256M or 1B) and I is the predictor. Lower is better.*

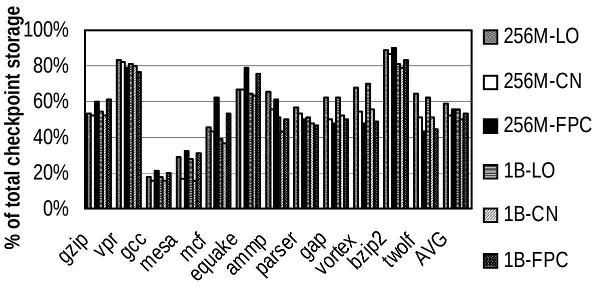


Figure 10: *Compression ratios with last-outcome (LO), combined neighbor (CN) and frequent pattern (FPC) compressors and for the 256M and 1B instruction intervals. Bars are labeled as I-P where I is the checkpoint interval (256M or 1B) and I is the predictor. Lower is better.*

Figure 12 shows a direct comparison of LZW and of our compressors. In the interest of space we restrict our attention to the 1B checkpoint interval. We report compression ratios for the 64K dictionary LZW (LZW-16bits), the LO and the CN compressors. We also consider combining the two value prediction based compressors

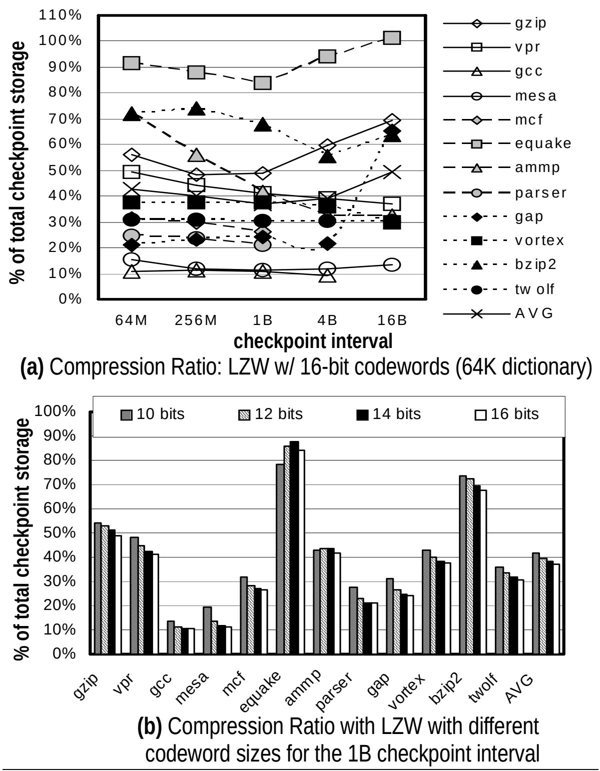


Figure 11: *LZW compression. (a) Compression ratio with an LZW compressor with a 16-bit codewords (64Kbyte dictionary) and for various checkpoint intervals. (b) Compression ratio for LZW compressors with 10 through 16 bit codewords (1K through 64K dictionaries) for the 1B instruction checkpoint interval. In both graphs lower is better.*

with LZW (LO+LZW and CN+LZW) as shown in Figure 2(b). Except for equake, LZW compression offers higher compression compared to LO and CN. Considering the high resource cost of LZW compression, the difference with CN is relatively small for gzip, gcc, mesa, mcf, ammp, and bzip2. However, for vpr, parser, gap and twolf compression with LZW is about twice as much as it is with

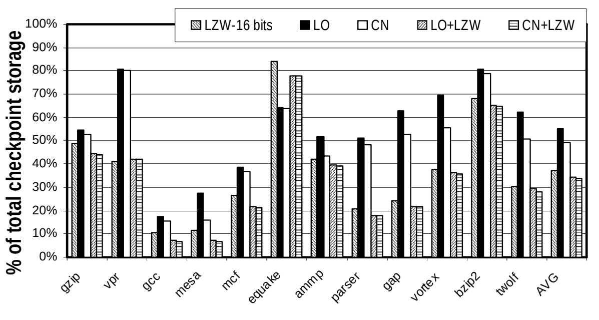
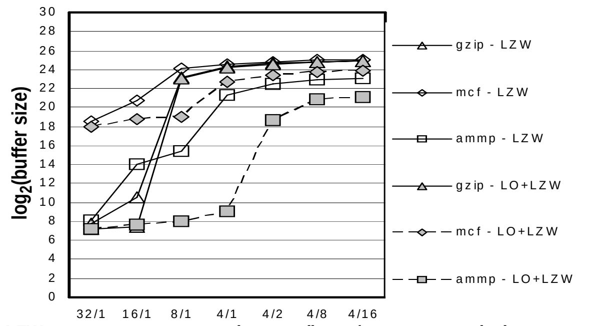


Figure 12: *Comparing dictionary-based compression with value prediction based compression and combining the two. Shown are the compression ratios for the following compressors: LZW with 64K dictionary (LZW-16bits), Last-outcome (LO), Combined Neighbor (CN), Last-outcome in-series with LZW (LO+LZW) and Combined Neighbor in-series with LZW (CN+LZW). Lower is better. Results are shown for the 1B checkpoint interval.*

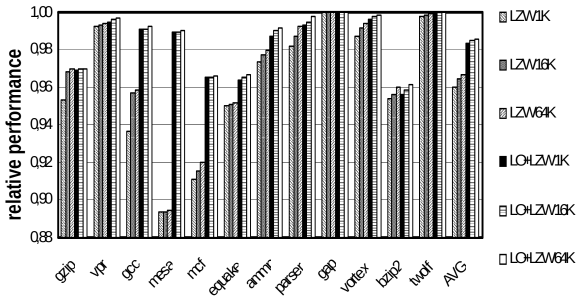
CN. When we combine our compressors with LZW we get a slight improvement in compression rate for most programs. While this improvement is small, as we will show in Section 5.6, this combination results in a significant reduction of on-chip buffer resources.

5.6. Performance and Resource Overheads

We first demonstrate that our compressors when combined with LZW can result in a high-performance, low-cost compressor. Here we consider two alternatives. The first uses just an LZW compressor (64K dictionary) and the second places a LO or a CN compressor in-series with the LZW (where our compressor appears first). In front of the LZW compressor there is a buffer ("in-buffer" of figure 2) so that we can avoid stalling the processor while the LZW processes checkpointed blocks. They key issue here is what size this buffer needs to be to avoid a significant impact on performance. We first report the maximum buffer size required so that we never have to stall the processor for various LZW processing rates. These results are shown in figure 13(a) for processing rates of 32 bytes every *processor* cycle through four bytes per 16 processor cycles. The latter processing rate is more realistic if we consider existing hardware implementations of dictionary-based compressors. Specifically, Pinnacle can process four bytes every cycle for a 133Mhz clock and a 0.25 micrometer implementation [23]. If we optimistically assume that the same hardware can be clocked twice as fast (e.g., 266Mhz) when implemented with a better technology then about 16 processor cycles will be needed assuming a 4Ghz processor cycle. In the interest of space we restrict our attention to three representative applications. It can be seen that as the LZW processing rate decreases a much smaller buffer is required when our compressor pre-processes the checkpoint (LO+LZW) compared to that required when LZW operates alone (LZW). While not clearly shown, in the worst case scenario of gzip our compressor reduces the on-chip buffer requirements by half (notice that the size scale is logarithmic). In the best case of ammp, a 2^{12} *times* smaller buffer is required when the LZW compressor processes four bytes every processor cycle (4/1).



(a) Maximum buffer size required for not stalling ever



(b) Performance degradation as a function of the on-chip "in-buffer" Consumption rate is 4 bytes per 16 processor cycles

Figure 13: *Combining a CN and an LZW compressor. (a) Maximum on-chip buffering required to avoid stalling the processor as a function of the LZW compressor processing rate. Processing rates are shown as B/C where B is the number of bytes processed simultaneously and C is the number of processor cycles required. Lower is better. In the interest of space, we report results for three representative applications. (b) Relative performance when the LZW compressor can process four bytes every 16 processor cycles for on-chip buffers of 1K, 16K and 64K. Shown are results for the LZW compressor alone and the LO in-series with the LZW (LO+LZW labels). All results are for the 256M checkpoint interval. Labels are of the form "C S" where C is the compressor and S is the "in-buffer" size. Higher is better.*

Figure 13(b) reports performance degradation over the base configuration that does no checkpointing for various checkpointing configurations that use on-chip "in-buffers" (see Figure 2) of 1K, 16K or 64K bytes. With LZW alone, an average performance slowdown of 3.7% is observed even with a 64K byte on-chip buffer. Furthermore, for all benchmarks increasing the on-chip buffer for 16K to 64K did not result in a noticeable improvement in performance. When our LO compressor is used in-series with LZW performance degradation is lower (three right-most bars). In the case of gcc, mesa and mcf and to a lesser extend gzip the performance benefit is significant. More importantly, even when we use just an 1K byte buffer (LO+LZW 1K), the LO+LZW combination offers performance that is better than that of LZW alone with a 64K byte buffer (LZW 64K). This is the case for all benchmarks except bzip2 and gzip. In these two programs the difference in performance is below 0.2%. For the same "in-buffer" the LO+LZW is always better than the LZW alone performance-wise. On the average, the LZW compressor with a 64Kbyte "in-buffer" results in a 3.7% performance slowdown while the combination of LZW and LO and with just a 1Kbyte buffer results in a 1.6% slowdown. The benefits of value-prediction-based compression are clearly seen when we consider worst case performance. With LZW compression alone and even if we use a 64K in-buffer, worst case performance degradation is at 11% for mesa. The LO+LZW compressor with just 1K in-buffer incurs a worst case performance degradation of 4.4% for bzip2. From these results it follows that when used in-series with a dictionary-based compressor our compressors offer better performance and reduce on-chip buffering requirements significantly. While modern high performance processors do have hundreds of millions of transistors and processors with billions of transistors will soon appear, arguing that the aforementioned reduction in on-chip buffers is insignificant is shortsighted. For one, even with a 64K on-chip buffer, dictionary compression still imposes a higher performance penalty. For another, there are numerous other uses for 64K of on-chip storage for improving performance and power or possibility for enhancing functionality.

Finally, we show that even if it was possible to build a faster dictionary-based compressor (this includes the option of making LZW more parallel) our compressors could still offer a performance and resource advantage. We focus only on three of the benchmarks where using a value-prediction-based compressor yields significant improvements. Differences for other benchmarks exist but they are small (we note that for the same size "in-buffer" the LO+LZW combination always offers better or same performance as the LZW alone). Figure 14 shows relative performance as a function of dictionary-based processing speed. We consider processing speeds of four bytes every four processor cycles (4/4), four bytes every eight processor cycles (4/8) and four bytes every 16 processor cycles (4/16). For each processing speed we report performance slowdowns for the dictionary-based compressor alone (LZW) and for when a last-outcome value-prediction-based compression is placed in-series with the LZW (LO+LZW). As the dictionary-based compressor becomes faster, overall performance improves for both systems. However, in all cases the LO+LZW compressor offers better performance. For gcc and mesa the LO+LZW results in significantly better performance for all rates. The same is true for mcf except when the processing rate rises to four bytes every four processor cycles. In the latter case, the difference between LZW and LO+LZW is negligible.

6. Summary

Recent work has showed that checkpoint/restore can have many useful applications in the areas of debugging,

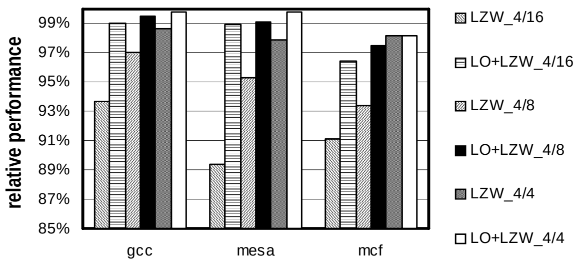


Figure 14: *Performance slowdown for LZW and the in-series combination with last-outcome (LO+LZW) for difference LZW processing rates. Processing rates are shown in the form "bytes/processor cycles". We assume a 1Kbyte "in-buffer" and a 256M checkpoint interval.*

crash analysis and reliability. For this type of applications checkpoint/restore mechanisms over several hundred million or few billions of instructions are desirable. A key consideration with such giga-scale CR mechanisms is the amount of storage required for checkpoints and the performance impact of saving checkpoints. In this work and motivated by the value locality found in the memory stream of typical programs we proposed hardware value-prediction-based compression of checkpoint data. Our compressors are inexpensive since they rely on small, direct-mapped structures they operate at high frequencies matching the processor's clock. We have shown that when used alone, they offer in most cases competitive compression rates when compared with much more expensive and slower dictionary-based compressors. We have also demonstrated that our compressors when combined with dictionary-based compression offer slightly higher compression rates, while significantly reducing on-chip buffering requirements. For this reason they offer an alternative, viable solution to the frequency and concurrency scalability issues that exist with dictionary-based compression. Value-prediction-based compressors may be useful in other applications where low-cost, high-speed compression is necessary.

Acknowledgments

We would like to thank the reviewers of this and previous conferences. Andreas Moshovos' research is supported by an NSERC Discovery Grant and a Canada Foundation for Innovation equipment grant. Part of this research was performed while the first author was on leave at the University of Athens, Greece.

References

[1] H. Akkary, R. Rajwar and S. T. Srinivasan, *Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors*, In Proc. 30th Annual International Symposium on Microarchitecture, December 2003.

[2] A. R. Alameldeen and D A. Wood. *Adaptive Cache Compression for High-Performance Processors*, In Proc.

of the 31st Annual International Symposium on Computer Architecture, June 2004.

[3] A. R. Alameldeen and D. A. Wood. *Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches*, Technical Report 1500, Computer Sciences Department, University of Wisconsin-Madison, April 2004.

[4] L. Benini, D. Bruni, B. Ricco, A. Macii, and E. Macii. *An Adaptive Data Compression Scheme for Memory Traffic Minimization in Processor-Based Systems.* In Proc. of the IEEE International Conference on Circuits and Systems, May 2002.

[5] D. Burger and T. Austin. The Simplescalar Tool Set v2.0, Technical Report UW-CS-97-1342. Computer Sciences Dept., Univ. of Wisconsin-Madison, June 1997.

[6] M. Burtscher and M. Jeeradit, *Compressing Extended Program Traces Using Value Predictors,* In Proc. of International Conference on Parallel Architectures and Compilation Techniques, September 2003.

[7] A. Cristal, D. Ortega, J. Llosa and M. Valero, *Kilo-instruction Processors*. In Proc. 5th International Symposium on High Performance Computing, Lecture Notes in Computer Science 2858, Springer, October 2003.

[8] M. Ekman and P. Stenström, *A Robust Main Memory Compression Scheme.* In Proc. of the 32nd Annual International Symposium on Computer Architecture, June 2005.

[9] P. Franaszek, J. Robinson, and J. Thomas. *Parallel Compression with Cooperative Dictionary Construction.* In Proc. of the Data Compression Conference, March 1996.

[10] W.-M. W. Hwu and Y. N. Patt, *Checkpoint Repair for High-Performance Out-of-Order Execution Machines*. IEEE Transactions on Computers 36(12), 1987.

[11] A. R. Lebeck, T. Li, E. Rotenberg, J. Koppanalil, and J. Patwardhan. *A Large, Fast Instruction Window for Tolerating Cache Misses.* In Proc. of the 29th Annual International Symposium on Computer Architecture, June 2002.

[12] M. Kjelso, M. Gooch, and S. Jones. *Design and Performance of a Main Memory Hardware Data Compressor.* In Proc. of the 22nd EUROMICRO Conference, 1996.

[13] J.-S. Lee, W.-K. Hong, and S.-D. Kim. *Adaptive Methods to Minimize Decompression Overhead for Compressed On-chip Cache.* International Journal of Computers and Application, 25(2), January 2003.

[14] A. Moshovos. *Checkpointing alternatives for high performance, power-aware processors.* In Proc. International Symposium on Low Power Electronics and Design, August 2003.

[15] A. Moshovos and A. Kostopoulos, *Gigascale Checkpoint/Restore,* Computer Group Technical Report, Univ. of Toronto, November 2004.

[16] S. Narayanasamy, G. Pokam and B. Calder, *BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging*. In Proc. of the 32nd Annual International Symposium on Computer Architecture, June 2005.

[17] J. T. Oplinger, M. S. Lam. *Enhancing software reliability with speculative threads.* In Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2003.

[18] M. Prvulovic and J. Torrellas, *ReEnact: Using Thread Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes,* In Proc. of the 30th Annual International Symposium on Computer Architecture, June 2003.

[19] J. E. Smith and A. R. Pleszkun. *Implementing Precise Interrupts in Pipelined Processors.* IEEE Transactions on Computers 37(5), pages 562-573, 1988.

[20] G. S. Sohi. *Instruction Issue Logic for High-Performance Interruptible, Multiple Functional Unit, Pipelined Computers.* IEEE Transactions on Computers 39(3), pages 349-359, 1990.

[21] D. J. Sorin, M. M. K. Martin, M. D. Hill and D. A. Wood, *SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery*. In Proc. of the 29th Annual International Symposium on Computer Architecture, June 2002.

[22] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, *IBM Memory Expansion Technology (MXT),* IBM Journal of Research and Development, Vol. 45, No. 2, 2001.

[23] R. B. Tremaine, T. B. Smith, M. E. Wazlowski, D. Har K.-K. Mak, and S. Arramreddy, *Pinnacle: IBM MXT in a memory controller chip,* IEEE MICRO, March-April 2001.

[24] T. A. Welch, *A Technique for High Performance Data Compression*, IEEE Computer Vol 17, No 6, pages 8-19, June 1984.

[25] M. Xu, R. Bodík and M. D. Hill, *A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay*. In Proc. of the 30th Annual International Symposium on Computer Architecture, June 2003.

[26] J. Yang, Y. Zhang, and R. Gupta. *Frequent Value Compression in Data Caches.* In Proc. of the 33rd Annual International Symposium on Microarchitecture, December 2000.

[27] K. C. Yeager, *The MIPS R10000 Superscalar Microprocessor,* IEEE MICRO, April 1996.

[28] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas, *iWatcher: Efficient Architectural Support for Software Debugging,* In Proc. of the 31st Annual International Symposium on Computer Architecture, June 2004.

[29] J. Ziv and A. Lempel, *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory, May 1977.