

Cost-Effective, High-Performance Giga-Scale Checkpoint/Restore

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

Alexandros Kostopoulos
Physics Department and
Department of Informatics and Telecommunications
University of Athens
Greece

Computer Engineering Group Technical Report
Electrical and Computer Engineering Department
University of Toronto
November 18, 2004

Abstract

This work proposes a novel checkpoint store compression method for giga-scale, coarse-grain checkpoint/restore. This mechanism can be useful for debugging, post-mortem analysis and error-recovery. The effectiveness of our compression method lies in exploiting value locality in the memory data and address streams. Previously proposed dictionary-based hardware compressors exploit the same properties however they are expensive and relatively slow. We observe that because program behavior is typically not very random much simpler mechanisms can offer most of the compression benefits. We propose three compressors that exploit value locality using very small direct mapped structures. Our compressors require few resources, can be easily pipelined and can process one full block per processor cycle. We study two uses 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 relatively competitive compression rates in most cases. We demonstrate that when combined with previously proposed hardware-based compression methods, our compressors improve overall compression rates while significantly reducing on-chip buffer requirements. Specifically, with an on-chip buffer of just 1K bytes a combination of our compressor with a dictionary-based compressor results into an overall performance slowdown of just 1.6% on the average for storing checkpoints to main memory. Moreover, this combination reduces checkpoints to 34% of their original size. The dictionary-based compressor alone even when used with a 64Kbyte on-chip buffer 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. Thus with a lot less resources (1Kbyte vs. 64Kbyte on-chip buffering) our technique offers better performance and compression. 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 few thousand). All aforementioned results are for a checkpoint interval of 256 million instructions.

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 some 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 [17], similar mechanisms are now also used to facilitate performance improvements via speculative execution [18, 10]. CR mechanisms have thus facilitated a "try out" approach to improving performance: Optimistically execute some instructions in hope of improving performance and if this does not work out (e.g., we violated sequential execution semantics or we should

have never executed these instructions) undo any changes done so that it looks as if we never executed any of these instructions. A commonly used technique of this type is control flow speculative execution where we predict the target of a control flow instruction and speculatively execute instructions down that path.

To date, CR mechanisms in use are fine-grain and relatively small-scale. A *fine-grain CR* mechanism allows us to restore the machine state at very fine execution intervals, possibly at every instruction. The reorder buffer [17] and the alias table checkpoint FIFO used in MIPS R10000 [25] are examples of fine-grain CR mechanisms. The first allows recovery at every instruction while the second only on speculated branches. Existing CR mechanisms are also *small-scale* since they are of narrow scope. Because CR mechanisms are used to facilitate speculative execution, checkpoints need to be held only for instructions within the relatively narrow execution windows of today's processors (e.g., around 100 or so instructions). Recent work has looked at CR mechanisms for larger windows in the range of a few thousand instructions [8, 11].

Recently, there have been several proposals for exploiting CR for other purposes. One such proposal uses CR for online software invariant checking to aid debugging and runtime checking [15]. Others utilize CR for reliability by using it to recover from hardware errors [16, 19]. Another application is *post-mortem* analysis [23] 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. Depending on the type of runtime checking performed, we may need CR over several thousand instructions for the first application. CR over several millions of instructions or more may be needed for the second application. Finally, CR over a few seconds of real execution time are desired for post-mortem analysis and possibly for error-recovery. With today's processing speeds the latter requirement translates to several hundred millions or even billions of instructions. We'll 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. Being able to restore to any instruction requires storage proportional to the number of executed instructions. Coarse-grain checkpointing has much lower storage requirements as we explain in section 2. In coarse-grain CR we take checkpoints every several hundred millions of instructions.

In this work, we focus on post-mortem analysis, however, the techniques we propose should be applicable to other CR uses. We propose giga-scale, coarse-grain CR mechanisms that are high-performance and that require significantly less resources than previously proposed techniques. In giga-scale CR the key issues are: (1) the amount of storage required for storing each checkpoint, and (2) the performance impact of taking checkpoints. As we show in section 5.2, 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 checkpoint data 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 while competing with regular program accesses. 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 program execution has to be stalled while buffer space is unavailable and additional checkpoint data has to be saved.

An obvious way of reducing off-chip bandwidth demand is to use on-line compression to reduce the footprint of checkpoint data. Previous work focused on identifying how checkpoints should be taken and what information they should include (program data and race outcomes) [16, 23]. Naturally, the question of how to best reduce the checkpoint data footprint in main memory was a secondary issue and thus previous work suggested using existing hardware dictionary-based compression [27]. Dictionary-based implementations require several millions of transistors and are comparatively slow (e.g., compress four bytes every cycle for a 133 Mhz clock) [20]. As we show in section 5, even under optimistic assumptions about its speed, large on-chip buffers are needed to avoid a significant performance hit with such compressors. Improving their speed through parallelism (while possible in-principle) may be difficult in practice as a result of increased interconnection and port requirements (see section 2.2).

In this work we take a closer look at optimizing the checkpoint mechanism so that it requires fewer on-chip resources while maintaining high-performance and avoiding increasing off-chip bandwidth demand. Specifically, motivated by the observation that typical programs exhibit value locality in the memory stream we propose a family of hardware value predictor based compressors that require very few resources and that can operate at the same frequency as the processor’s core as they can be easily pipelined (software value prediction based compressors for trace compaction have been proposed in [7]). We propose two uses of our compressors: (1) Used alone as a low cost on-chip compressor, and (2) used in-series with previously proposed hardware dictionary-based compressors. We demonstrate that when used alone and in most cases our compressors offer competitive compression ratios when compared to the much more expensive dictionary-based compressors. However, the key result of our work is that when used in-series with dictionary-based compression, our compressors offer three advantages: (1) slightly better compression ratios, (2) significantly reduced on-chip buffering requirements, and (3) lower performance degradation to perform checkpointing. Thus, combining our compressors with a slower dictionary-based compressor results in a cost-effective, high-performance on-chip compressor that is *always* (for the programs we studied) better performance-, compression- and resource-wise than a stand-alone dictionary-based compressor.

Our contributions are: (1) We study the checkpoint resource and performance overheads of typical programs and 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 and demonstrate that it is possible to reduce the on-chip resource requirements and the performance slowdown incurred for checkpointing. To the best of our knowledge this is the first work that: (1) takes a detailed look at the hardware/performance overheads of checkpoint compression for giga-scale checkpointing (previous work only focused on the off-chip storage requirements for one case of compressor and checkpoint interval), and (2) proposes a value-predictor based compressor for checkpoint storage reduction.

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 what CR is supposed to do. 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 unrolled back to the checkpoint. Using the information stored in 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 what it means to read and write from and to devices. For some devices it may not be possible to do CR as some side-effects may be irreversible (e.g., when a device communicates with another system and it holds state relating to the communication). 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. While this discussion is interesting and necessary, it is presently beyond the scope of this paper. The mechanisms we propose are a necessary step towards complete giga-scale CR solution.

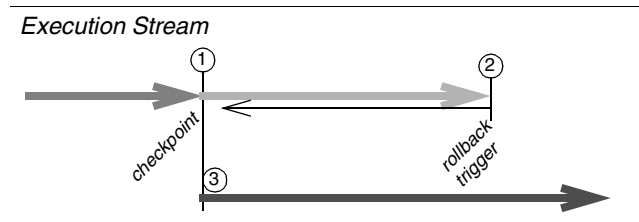


Figure 1: *What Checkpoint/Restore does. (1) First a new checkpoint is taken. (2) Later on a rollback is triggered. The effects of all instructions executed since the checkpoint is undone. (3) Execution resumes from the checkpoint.*

For our purposes a checkpoint can be used to reconstruct the register and memory values. 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, in the rest of this work we assume that every time a checkpoint is taken, a complete image of the register file is saved.

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 so that they can be reversed when needed (this is equivalent to the history file method for fine-grain, small-scale CR [17]). Specifically, initially the checkpoint is empty. When a memory write occurs, we record its address and value before the write takes place. Only one record per memory address is needed. Subsequent writes to a previously checkpointed memory location do not need to save anything in the checkpoint store. This is because we do not care to rollback to any intermediate point in the execution. 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 since an address will have to be saved per single byte of memory data. Instead, it is correct and 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. 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*. Finally, 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.

2.1 A Checkpoint/Restore Architecture

Because the checkpoints are relatively large (several megabytes is typical) on-chip storage is presently not an option. Moreover, even if this was technically possible, it may not be the best allocation of on-chip resources (larger caches or branch predictors or additional processors seem more appropriate). An obvious choice is to use main memory to hold the checkpoints. In this case, checkpoint writes will compete for main memory bandwidth. For this reason, it makes sense to try to reduce the bandwidth demand by first compressing the checkpoint records. Figure 2(a) shows a complete checkpointing architecture. Whenever a

cache block is checkpointed, it is 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.

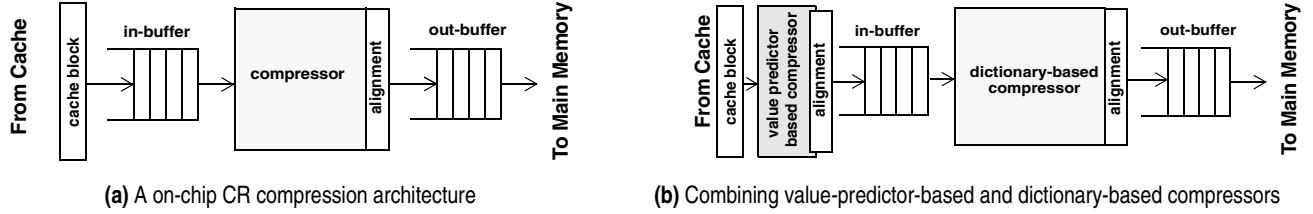


Figure 2: (a) A checkpoint/restore architecture with an on-chip compressor. We show only the checkpoint path since restoring is trivial. Cache blocks are placed into a buffer before the first write occurs. The compressor processes blocks and places its output into another buffer. Whenever possible, the 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. There is no need for an additional buffer in front of the value-prediction-based compressor since it is capable of processing a full cache block per cycle.

2.2 Performance, Resource and Complexity Trade-offs

From the preceding discussion, it follows that several performance and resource considerations exist with giga-scale CR mechanisms. Specifically, the on-chip compression engine and the associated buffers represent a resource overhead. Also, whenever any of the buffers fills-up it is not possible to immediately checkpoint any additional cache block. In this case, we have to stall the processor. It follows that the size of on-chip buffers and the speed with which the compression engine can process checkpoint data can impact overall performance. Finally, the compression ratio achieved also impacts performance since the smaller the checkpoint storage required the less demand will be placed on the main memory bus. All aforementioned considerations place additional artificial limits on processor performance.

Complexity considerations also exist and they can impact performance. The compressor generates a compressed stream of data which is placed on the “out-buffer” for writing into memory. Effectively using the storage of the “out-buffer” requires *aligning* the compressed data. 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”. To reduce the size and the complexity of this network it is thus preferable to use compressors that reduce the number of possible alignments necessary. 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. Assuming 64 byte “out-buffer” entries and blocks the former compressor would result in just eight possible alignments while a compressor that operates at a bit granularity would result in 512 possible alignments.

Previous work in giga-scale CR assumes an on-chip LZ77-like dictionary-based compression engine [23]. The advantages of such compression engines are that they offer high compression and that they have been already built in hardware. The disadvantages are that they are relatively expensive and slow. To avoid stalling the processor when using a relatively slow compression engine, 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 [20]. It has been built using 0.25 micron process into a chipset operating with a 133Mhz clock [21]. 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 5.5, even with a 64K on-chip buffer a performance hit in excess of 10% is incurred in some cases even if we assume a compressor that it is twice as fast.

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 their solution amounts to splitting 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 [21]. Every cycle, each one sees a probe, plus four updates (one from itself and three from the other CAMs). It is clear that with existing technology it is not possible to operate these CAMs with the clock of a multi-GhZ processor core. Furthermore, as described it is not possible to pipeline the compression hardware.

Based on the preceding discussion, 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 works fast enough to avoid stalling the processor while requiring little on-chip buffering? By exploiting the well known property of value locality we show that in many cases we can get very similar compression with much lower cost. More importantly, we show it is possible to significantly reduce the amount of on-chip buffers and to avoid the complexities and cost of further parallelization of an LZ77-like compressor while getting slightly better compression rates and better overall performance. In the latter case, a new simple compressor is placed in-series with a slower dictionary-based predictor as shown in figure 2(b).

3 Exploiting Value Predictability to Simplify 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, intuitively one would hope that it would also hold for a subset of the references such as the first updates to each memory block after arbitrary chosen points during execution. This is indeed the case as we show in section 5.3. 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 shows the simplest alternative where 16 single-entry value predictors are used for block data and a single-entry stride predictor for block addresses. We chose 16 single entry predictors so that 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. Specifically, the first word is processed by the first predictor, the second word by the second predictor and so on. As shown in figure 4, the final representation (checkpoint record) of a block consists of a header, and possibly an address and up to 16 words. The header is a bit vector with one bit per word in the original cache block. A two byte header is sufficient for 64-byte blocks. A bit value of 1 is used when the value can be predicted. In this case, the actual value is not stored in the checkpoint record. If the value cannot be predicted the corresponding header bit is set to 0 and the value is stored in the checkpoint record. One more bit¹ is used for indicating whether the block address can be predicted by the address predictor. Only if prediction is not possible the block address is included in the checkpoint record. Whenever a new checkpoint starts, all predictor entries are zeroed out (otherwise we would have to save the predictor state in the checkpoint store also). 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 barrel shifter stages. More importantly, this process can be pipelined thus maintaining high performance. A simple alignment network will also be

¹ We store a full byte to simplify the alignment network at the compressor's output.

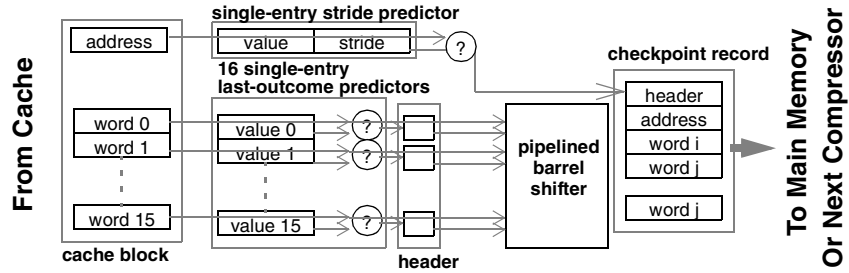


Figure 3: Last-outcome based compressor. Each of the 16 words of the cache block are passed to a different single-entry last-outcome value predictor. This way we need to store only those words that are not predicted correctly. A 16-bit header vector identifies the words that are predicted correctly. 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 also align the address into place (storing the address requires a byte header and if necessary another four bytes for the actual address).

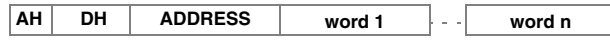


Figure 4: Checkpoint record format. “AH” or address header is a single byte which is non zero if the address can be predicted. “DH” or data header is a 16-bit vector with one bit per word in the original cache block. A bit value of 1 indicates that the corresponding word can be predicted. The optional “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.

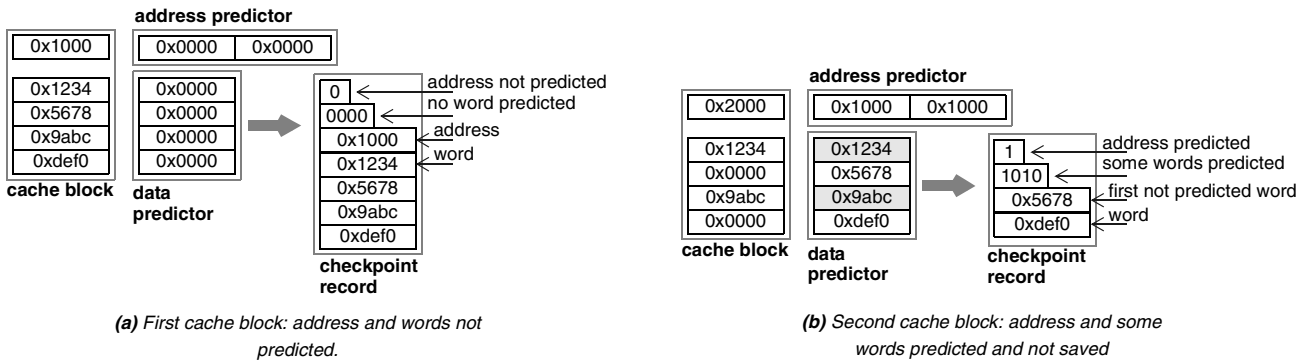


Figure 5: An example of value-predictor-based compression. In the interest of space and without a loss in generality we assume four word cache blocks where addresses and words are all 16 bits long. (a) Initially, the predictors are zeroed out. The first write to block 0x1000 occurs. Since no value is 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.

needed to align the continuous block into position for writing it into the on-chip buffers (this, however, would be required for any compressor). The compressor should be able to operate at the same frequency as the processor core since: (1) all cache words are processed simultaneously, (2) there is not intra-block dependences for the actions performed, and (3) prediction amounts to few simple operations (e.g., comparison, selecting the output value and updating a single-entry predictor). Overall, our compressor requires few resources. Specifically, 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. This is nowhere close to the one million gates required by a hardware dictionary-based compressor [21].

3.1 Combined Value Predictor Based Compressors

While we have described a simple last-outcome based compressor, the concept can be generalized by using other predictors. We have experimented with other predictors that offer a trade-off between cost and compression ratio. We discuss two of these predictors. 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 avoid losing information¹. In the *combined stride* predictor the “11” combination indicates that the value can be predicted by a stride predictor. 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. It may appear that the design of these two predictors is completely ad-hoc. The neighbor predictor attempts to exploit locality *within* cache blocks in addition to locality *across* blocks that is captured by its last-outcome component. The stride predictor is tailored to sequences that lookup based predictors cannot capture (i.e., it can predict a value that it has not yet seen). Intuitively this is important as even LZ77 compressors are lookup based. Finally, the previous-to-last-outcome predictor is a natural extension of the last-outcome predictor.

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

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.

3.2 Support Mechanisms

In this section, we discuss two support mechanisms that are required by all giga-scale CR methods. First, 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. When we want to take a new checkpoint we reset all bits. Any time a memory write is about to happen we use the block address to check the corresponding bit. If it is zero then this is the first write to the block and we need to checkpoint its state. After doing so we set the bit so that subsequent writes to the same block would not trigger a checkpoint store. A practical implementation 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. An invalid vector is equivalent to all zeroes (no update has occurred for this checkpoint). Using 64-bit counters and even if we assume 1K checkpoints per second (which is very frequent for the application we are interested in), 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 on-chip bloom filters [5]), however, 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. Further investigation of this topic is beyond the scope of this work.

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 especially those for on-line memory value compression and work on value prediction based compression.

Burtscher and Jeeradit first proposed value prediction based compressors [7]. They propose software-based compressors for quickly compressing programs traces. In this work we rely on the same concept as we exploit value predictability for compression. However, there are several key differences. Specifically, in this work we are concerned with hardware based compressors sharing the goal of quick compression but having the additional constraints of using simple hardware and few hardware resources. The tradeoffs in software

based compression are much different than those in hardware based compression. Certain operations are much more expensive in software and vice versa. Moreover, in this work we are interested in compressing checkpoints as opposed to whole program traces. A program trace contains a fine-grain checkpoint (full instruction history) and thus it is likely to contain much higher value and address locality. Moreover, a trace may contain additional information that would not be required for recovery.

Existing fine-grain, small-scale CR mechanisms are variations of designs that were proposed in the late 80's for supporting precise exceptions and out-of-order, speculative execution [10, 17, 18]. Recently there has been work in extending these mechanisms for scheduling windows of several hundreds [11] or several thousands [8] of instructions. 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 [15]. Zhou *et al.* propose hardware-based monitors for debugging purposes. 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 gigascale CR. Prulovic and Torrellas proposed ReEnact a system for recovering from data races in multiprocessors [16]. ReEnact is also targeted at relatively short intervals. Xu, Bodik and Hill describe a CR mechanism for post-mortem analysis [23]. 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. In this work, we demonstrate that this is the case for post-mortem analysis CR over several hundred millions or a few billions of instructions.

Several hardware compressors for memory data have been developed. IBM's MXT technology [20] relies on a variation [9] of the LZ77 algorithm [27] to compress 1K cache blocks. We demonstrate that while dictionary-based algorithms such as LZ77 can capture value locality, simpler mechanisms such as the ones we propose 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, 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 [24]. It too processes the block serially. Our compressor is much simpler than these methods. Alameldeen and Wood proposed Frequent Pattern Compression, a relatively fast compressor that

is capable of compressing a 64 byte block in about five cycles assuming a 12 FO4 gate delays per cycle [3]. Their compressor exploit the fact that often data values need a lot fewer bits for their representation. Our technique can be pipelined and can process a full 64 byte block per cycle under similar assumptions. Comparing with Frequent Pattern Compression while interesting is beyond the scope of this paper. Our key contribution here is in identifying the tradeoffs that exist with gigascale CR and in showing how simple hardware compressors can be used to reduce on-chip resources requirements. The actual choice of predictor while important is secondary.

We should clarify that value predictability and compressibility while related they are two very different concepts. In many cases a predictable stream is also compressible and vice versa but this is not necessarily always the case. Furthermore in prediction we are trying to guess the next set of values having seen the history of previous values. In compression we are trying to decide which representation requires less storage relying also on the history of previous values. These two actions are different.

5 Evaluation

In section 5.1 we present our experimentation 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 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 5.5, we demonstrate that the combination of our compressors with a dictionary-based compressor significantly reduces on-chip buffer requirements while offering better performance.

5.1 Methodology

We extended the SimpleScalar v3.0 simulators [6]. 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 superscalar processor. Our processor is deeply pipelined to appropriately reflect modern processor designs. We modified SimpleScalar’s main memory system to appropriately model bus contention. This is necessary since checkpointing increases main memory bandwidth demand. Table 2 depicts the base processor configuration and other key simulation parameters. We used the following 12 integer and 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 data input for all benchmarks except *mcf* and *parser*. For the latter two we used a training input 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 MIPS-like PISA architecture using GNU’s gcc v2.9 (options: “-O2 -funroll-loops -finline-functions”). For the functional simulations we simulated up to 80 billions instructions or to completion (whichever came first). In order to obtain reasonable 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 [27] compressor we opt for an LZW implementation [22].

Table 2: Base processor configuration and other key simulation parameters.

Branch Predictor	16k GShare+16K bi-modal 16K selector 2 branches per cycle	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.
Instruction Window	256 entries	Instructions Simulated	80 Billion max. functional Five Billion timing
Issue/Decode/Commit	any 8 instructions / cycle	Input Data Set	Training for mcf and parser Reference for all others
DL1/IL1 Geometry	64Kbyte/64-byte blocks/4-way SA	Checkpoint Intervals	65M, 256M, 1B and 4B instructions
L1/UL2 Latencies	3/16 cycles	On Chip Buffers	1K, 4K, 16K and 64K bytes.
Fetch Unit	Up to 8 instr. per cycle 64-entry Fetch Buffer 2 branches per cycle Non-blocking I-Cache	Main Memory	Infinite capacity, 100 cycles latency 16 Banks, one port/bank 64-byte interleaved
Load/Store Queue	64 entries, 4 loads or stores per cycle Perfect disambiguation	LZW dictionary size	1K, 4K, 16K or 64K
FU Latencies	same as MIPS R10000	LZW consumption rates	from 32 bytes/cycle to 4 bytes per 16 cycles
UL2 Geometry	1Mbyte/64byte blocks/8-way SA		

In all our simulations *we ignore the first checkpoint to avoid artificially skewing our results towards higher compression ratios*. Because most memory blocks initially are zeroed out in SimpleScalar, the vast majority of values stored for the first checkpoint would be zeroes (recall that a checkpoint holds the values of memory locations prior to the first write). As a result, the first checkpoint is always highly predictable and compressible. We verified that the compression rates did not decrease had we skipped more checkpoints.

As a result of the number of instructions they execute and because we do not include the first checkpoint in our measurements we did not report statistics for the 16 billion checkpoint interval for gcc, mcf and parser (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: From Thousands to Billions

In this section we study the checkpoint storage requirements and how they vary as a function of the checkpoint interval. Our primary goal here is to show that on-chip storage even when one considers applications such as SPEC2000 is not sufficient for storing 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. Generally, checkpoint requirements increase almost linearly with the checkpoint interval. For some programs (e.g., amp, gcc and twolf), the 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. Only for relatively short checkpoint intervals (i.e., 256K instructions or less) would few megabytes be sufficient to store a few checkpoints on-chip. In the rest of this work we will focus on checkpoint intervals of 64 million instructions or more since our target application requires large checkpoint intervals.

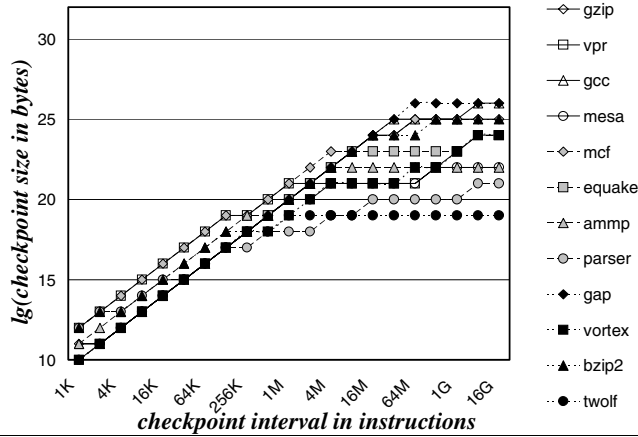


Figure 6: Checkpoint Storage Requirements as a function of the checkpointing interval: 1K through 16Billion instructions.

Figure 7(a) shows the size of the largest checkpoint per application for checkpoint intervals of 64 million (64M) through 16 billion (16B) instructions. We report the checkpoint size as a fraction of the application’s overall data memory footprint. The checkpoint size includes both data and addresses. The memory footprint of each application is listed next to the graph legend (in megabytes). In most applications checkpoint requirements increase with the checkpoint interval and in some cases abruptly (e.g., for equake and for the four billion checkpoint interval). In some applications, the maximum checkpoint size remains virtually unchanged. In general, the checkpoint size is directly analogous to the amount of memory data that is updated during the interval. Figure 7(b) shows the cumulative distribution of all checkpoint sizes for few representative cases. For clarity, we report the base two logarithm of the checkpoint size. 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 (anywhere from 5% to 43% of the overall data memory footprint) and hence techniques for compressing checkpoints are worth investigating¹. Furthermore, on-chip storage of even a single checkpoint would require several tens of megabytes for some applications.

¹ Thus far we have discussed storage requirements only. Another key consideration is the amount of traffic generated by checkpoints and how it impacts performance. Later on, we will show that the performance impact can be significant.

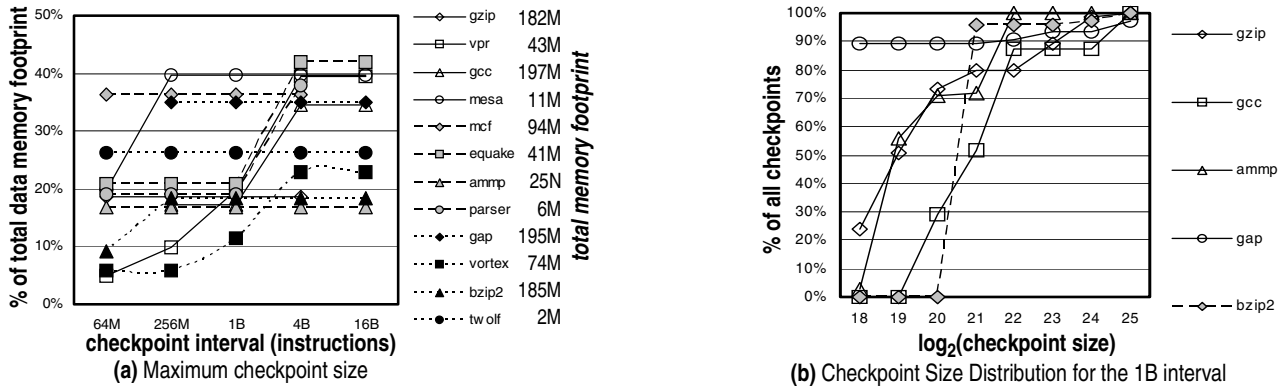


Figure 7: (a) Maximum checkpoint storage footprint as a fraction of the application’s overall data memory footprint for various checkpoint intervals. The overall data memory footprint per application and in megabytes is shown next to the legend. (b) Cumulative checkpoint size distribution per benchmark and for the 1G instruction checkpoint interval. For clarity we show few representative applications.

5.3 Value Prediction Based Compressors

Having shown that the applications have considerable checkpoint storage requirements we next study the compression possible with the last-outcome compressor which is 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. This measurement includes the compression headers in addition to data and addresses whenever needed. A compression ratio of 50% means that we need half the space to store the checkpoint, while 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. In some cases, higher compression is possible for larger intervals (e.g., equake) while for most applications compression is lower for the largest checkpoint interval. The amount of compression possible depends on how predictable the data words and the block addresses are. The prediction rate 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. While no clear trend exists for values, block address prediction rates tend to increase with the checkpoint interval for all applications except gap.

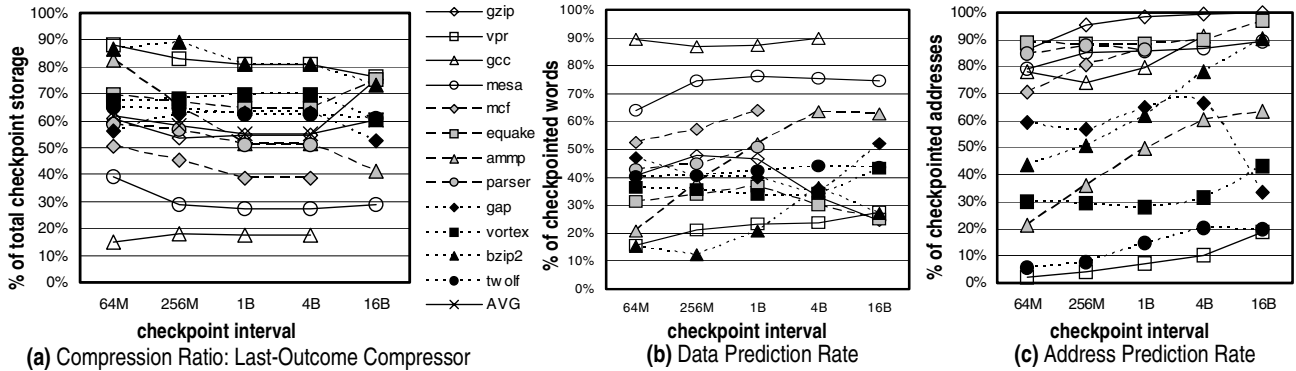


Figure 8: Last-outcome-based compressor. (a) Overall compression ratio for various checkpoint intervals. Reported is the ratio: $\text{Total_Checkpoint_Storage}_{\text{after-compression}} / \text{Total_Checkpoint_Storage}_{\text{without-compression}}$. Lower is better. (b) Prediction rates for data words.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

5.3.1 Combined Predictor Based Compressors

We next 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. In many cases, the differences amongst the three compressors 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.

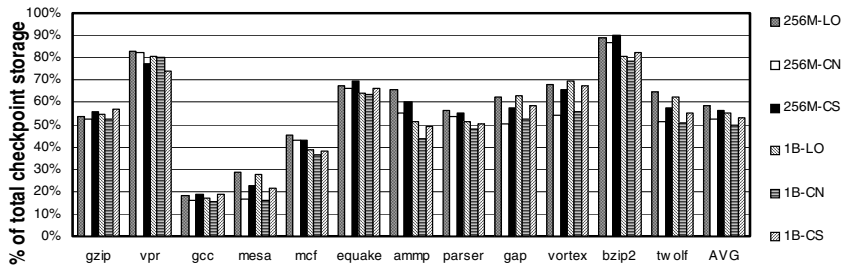


Figure 9: Compression ratios with three value predictor based compressors and for the 256M and 1B instruction checkpoint intervals. Shown are the last-outcome (LO), combined neighbor (CN) and combined stride (CS) predictors. Bars are labeled as I-P where I is the checkpoint interval (256M or 1B) and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

5.4 Interaction with Dictionary-Based Compression

We next look at dictionary-based compressors. For this purpose we use 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 10(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 quake 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 [20, 21]. In figure 10(b), we show that indeed in most cases, using larger dictionaries results in lower compression ratio (hence in higher compression). Shown are the compression ratios for LZW compressors with 1K (10 bits), 4K (12 bits), 16K (14 bits) and 64K (16 bits) dictionaries for the 1B checkpoint interval.

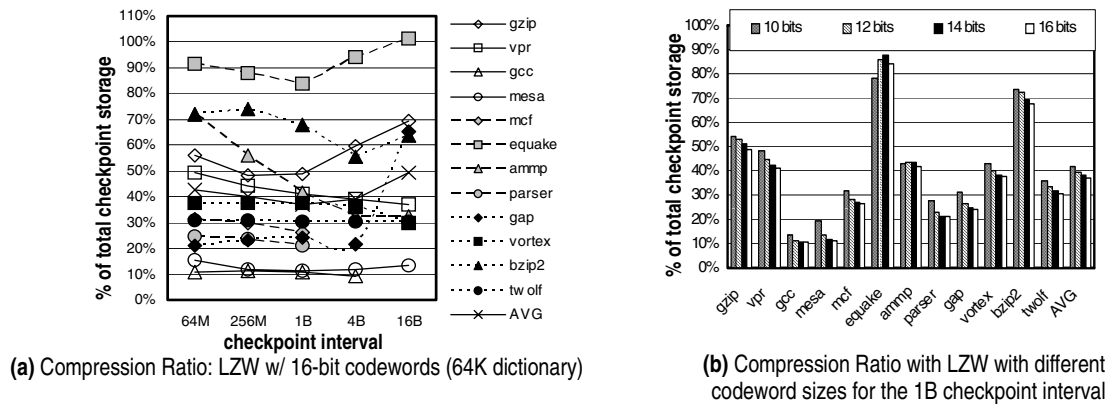


Figure 10: LZW compression. (a) Compression ratio with an LZW compressor with a 16-bit codewords (64Kbyte dictionary) and for various checkpoint intervals. (b) Compression ratio for LZW compressors with 10 through 16 bit codewords (1K through 64K dictionaries) for the 1B instruction checkpoint interval.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

Figure 11 allows for a more direct comparison of LZW with 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 with LZW (LO+LZW and CN+LZW) as shown in figure 2(b). Except for quake, 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 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 5.5, this combination results in a significant reduction of on-chip resources (we need a much smaller on-chip “in-buffer”).

5.5 Performance and On-Chip Buffer Overhead

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. In front of the LZW

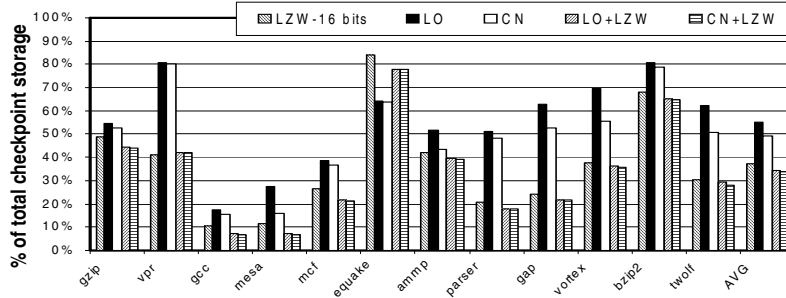


Figure 11: Comparing dictionary-based compression with value prediction based compression and combining the two. Shown are the compression ratios for the following compressors: LZW with 64K dictionary (LZW-16bits), Last-outcome (LO), Combined Neighbor (CN), Last-outcome in-series with LZW (LO+LZW) and Combined Neighbor in-series with LZW (CN+LZW). Lower is better. Results are shown for the 1B checkpoint interval.

compressor there is a buffer (“in-buffer” of figure 2) so that we can avoid stalling the processor while the LZW processes checkpointed blocks. The 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. This result is shown in figure 12(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, we have explained that Pinnacle can process four bytes every cycle for a 133Mhz clock and a 0.25 micron implementation [21]. 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 buffer that is 2^{12} times smaller is required when the LZW compressor processes four bytes every processor cycle (4/1).

Figure 12(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 3.4% is observed even with a 64K byte on-chip buffer. Furthermore, for all benchmarks increasing the on-chip buffer for 16K to 64K does 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 extent 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

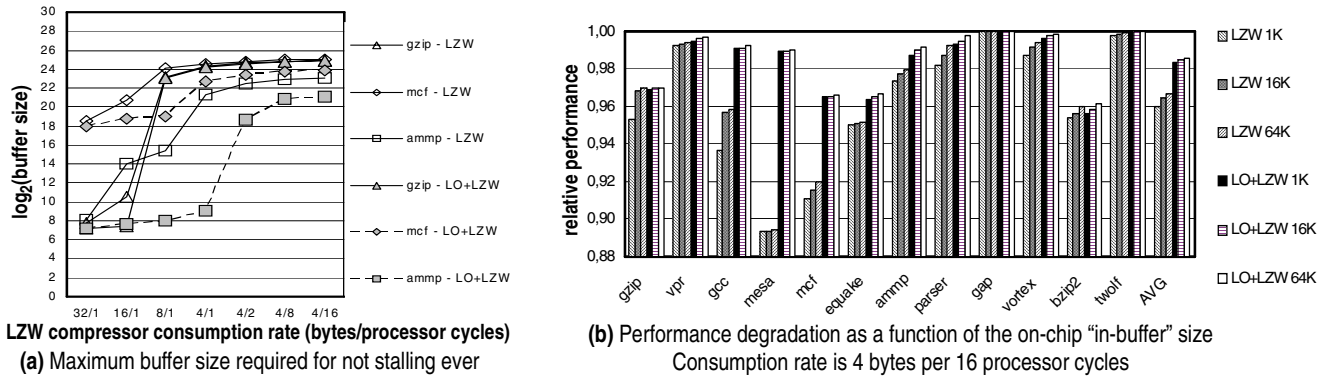


Figure 12: Combining a CN and an LZW compressor. (a) Maximum on-chip buffering required to avoid stalling the processor as a function of the LZW compressor processing rate. Processing rates are shown as B/C where B is the number of bytes processed simultaneously and C is the number of processor cycles 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 "CS" where C is the compressor and S is the "in-buffer" size. Higher is better.

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.

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 13 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 processing 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.

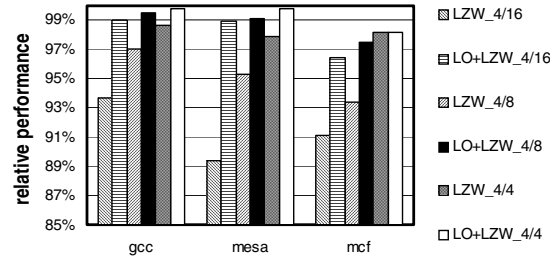


Figure 13: Performance slowdown for LZW and the in-series combination with last-outcome (LO+LZW) for difference LZW processing rates. Processing rates are shown in the form “bytes/processor cycles”. We assume a 1Kbyte “in-buffer” and a 256M checkpoint interval.

6 Summary

Recent work has showed that checkpoint/restore can have many useful applications in the areas of debugging, 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 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. However, in some cases, dictionary-based compression can reduce checkpoint store by twice as much. As we have demonstrated, 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.

References

- [1] Haitham Akkary, Ravi Rajwar and Srikanth T. Srinivasan, *Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors*, In Proceedings 30th Annual International Symposium on Microarchitecture, December 2003.
- [2] Alaa R. Alameldeen and David A. Wood, *Adaptive Cache Compression for High-Performance Processors*, In Proceedings of the 31st Annual International Symposium on Computer Architecture, June 2004.
- [3] Alaa R. Alameldeen and David 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] Luca Benini, Davide Bruni, Bruno Ricco, Alberto Macii, and Enrico Macii, *An Adaptive Data Compression Scheme for Memory Traffic Minimization in Processor-Based Systems*. In Proceedings of the IEEE International Conference on Circuits and Systems, May 2002.
- [5] B. Bloom. *Space/time trade-offs in hash coding with allowable errors*. Communications of ACM, 13(7), July 1970.
- [6] D. Burger and T. Austin. The SimpleScalar Tool Set v2.0, Technical Report UW-CS-97-1342. Computer Sciences Department, University of Wisconsin-Madison, June 1997.

- [7] M. Burtscher and M. Jeeradi, *Compressing Extended Program Traces Using Value Predictors*, In Proceedings of International Conference on Parallel Architectures and Compilation Techniques, September 2003.
- [8] Adrián Cristal, Daniel Ortega, Josep Llosa and Mateo Valero, *Kilo-instruction Processors*. in Proceedings 5th International Symposium on High Performance Computing, Lecture Notes in Computer Science 2858, Springer, October 2003.
- [9] Peter Franaszek, John Robinson, and Joy Thomas. *Parallel Compression with Cooperative Dictionary Construction*. In Proceedings of the Data Compression Conference, March 1996.
- [10] Wen-Mei W. Hwu and Yale N. Patt, *Checkpoint Repair for High-Performance Out-of-Order Execution Machines*. IEEE Transactions on Computers 36(12): 1496-1514, 1987.
- [11] Alvin R. Lebeck, Tong Li, Eric Rotenberg, Jinson Koppanalil, and Jaidev Patwardhan. *A Large, Fast Instruction Window for Tolerating Cache Misses*. In Proceedings of the 29th Annual International Symposium on Computer Architecture, June 2002.
- [12] Morten Kjelsø, Mark Gooch, and Simon Jones. *Design and Performance of a Main Memory Hardware Data Compressor*. In Proceedings of the 22nd EUROMICRO Conference, 1996.
- [13] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. *Adaptive Methods to Minimize Decompression Overhead for Compressed On-chip Cache*. International Journal of Computers and Application, 25(2), January 2003.
- [14] Andreas Moshovos. *Checkpointing alternatives for high performance, power-aware processors*. In Proceedings International Symposium on Low Power Electronics and Design, August 2003.
- [15] Jeffrey T. Oplinger, Monica S. Lam. *Enhancing software reliability with speculative threads*. In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, October 2003.
- [16] Milos Prvulovic and Josep Torrellas, *ReEnact: Using Thread Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes*, In Proceedings of the 30th Annual International Symposium on Computer Architecture, June 2003.
- [17] James E. Smith and Andrew R. Pleszkun. *Implementing Precise Interrupts in Pipelined Processors*. IEEE Transactions on Computers 37(5), pages 562-573, 1988.
- [18] Gurindar S. Sohi. *Instruction Issue Logic for High-Performance Interruptible, Multiple Functional Unit, Pipelined Computers*. IEEE Transactions on Computers 39(3), pages 349-359, 1990.
- [19] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill and David A. Wood, *SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery*. In Proceedings of the 29th Annual International Symposium on Computer Architecture, June 2002.
- [20] 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.
- [21] 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.
- [22] Terry A. Welch, *A Technique for High Performance Data Compression*, IEEE Computer Vol 17, No 6, pages 8-19, June 1984.
- [23] Min Xu, Rastislav Bodík and Mark D. Hill, *A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay*. In Proceedings of the 30th Annual International Symposium on Computer Architecture, June 2003.
- [24] Jun Yang, Youtao Zhang, and Rajiv Gupta. *Frequent Value Compression in Data Caches*. In Proceedings of the 33rd Annual International Symposium on Microarchitecture, December 2000.
- [25] K. C. Yeager, *The MIPS R10000 Superscalar Microprocessor*, IEEE MICRO, April 1996.
- [26] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas, *iWatcher: Efficient Architectural Support for Software Debugging*, In Proceedings of the 31st Annual International Symposium on Computer Architecture, June 2004.
- [27] J. Ziv and A. Lempel, *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory, May 1977.