

Lightweight Checkpointing for Faster SOAP Deserialization

Nayef Abu-Ghazaleh and Michael J. Lewis
State University of New York (SUNY)
Binghamton, NY 13902
{nayef, mlewis}@cs.binghamton.edu

Abstract

Differential Deserialization (DDS) is an optimization technique that exploits similarities between incoming SOAP messages to reduce deserialization time. DDS works by checkpointing the state of the SOAP deserializer at various points while deserializing a message, and using those checkpoints to avoid full deserialization of similar messages. DDS can improve performance in many cases, but its benefit is limited by the potentially significant memory and processing overhead associated with its checkpointing mechanism. Differential checkpointing (DCP) substantially reduces memory use, but still requires significant processing overhead. In this paper, we introduce lightweight checkpointing (LCP), a checkpointing approach that significantly reduces the cost of both DDS and DCP, in terms of both memory use and processing time. LCP statically determines locations in the incoming message where it would be most efficient to create checkpoints. LCP creates checkpoints much faster than both our original DDS checkpointing mechanism and our DCP approach. LCP also has significantly smaller memory requirements. For example, in some of our test cases, LCP requires only 10% of the memory that DCP requires, and only 3% of the memory that our original approach required. In terms of processing time, deserialization with LCP is approximately 50% to 60% faster than without differential deserialization, when approximately half the message is unchanged from the previous message.¹

1 Introduction

Differential Deserialization (DDS) [1] is a SOAP optimization technique that takes advantage of similarity between consecutive messages in a SOAP communication flow. The server-side optimization, in our bSOAP implementation, essentially checkpoints the state of the SOAP

parser at various points while deserializing a message. While the next message is arriving, the parser may be able to operate in “fast mode,” wherein expensive XML parsing and conversion is replaced with simpler and faster checksum comparisons that determine whether the new message is the same (in some regions) as the previous one. For portions of the message that are repeated in consecutive messages, DDS avoids the cost of deserializing again.

Because converting information out of (and into) ASCII-based XML to (and from) binary in-memory format can be the most expensive component of end-to-end SOAP processing time [2], and since DDS can eliminate this cost for some portions of a message, the technique has the potential to be extremely effective. This is especially true for the data types that are most computationally expensive to serialize and deserialize, namely scientific data stored in floats and doubles. This potential also depends on the degree of similarity among consecutive messages in the message stream, which is an application-specific property. We addressed this issue in our description of an analogous but different optimization technique on the client side, called differential serialization (DS) [3].

This paper addresses the primary challenge to implementing DDS efficiently, namely generating, storing, and using parser checkpoints. Our initial DDS implementation used a simple technique of storing a series of complete, self-confined, parser checkpoints. Each checkpoint corresponded to the state of the parser at a given point in the last incoming message. A checksum of the incoming message, up to that point, was compared with a checksum stored with the parser. The parser switched back and forth between normal SOAP parsing, and “fast mode” parsing (comparing equivalent checksums), as the opportunity and need to do so presented themselves. We showed that even this simple technique improved performance, but at the cost of significant memory use.

We then observed that consecutive checkpoints within this sequence of checkpoints associated with a message *themselves* showed similarity. This presented the opportunity to make only some of the checkpoints complete and

¹This research is supported by NSF Career Award ACI-0133838 and DOE Grant DE-FG02-02ER25526, and NSF Award CNS-0454298.

```

<address>
  ●<street>Main St.</street>
  ●<city>Johnson City</city>
  ●<state>New York</state>
  ●<zip>13790</zip>
  ●<phone>607-123-4567</phone>
</address>

```

Figure 1. A portion of a SOAP message. The bullets depict potential locations of lightweight checkpoints.

self-contained; other smaller checkpoints encode only the differences between a previous complete checkpoint. This technique, called *differential checkpointing* (DCP), was introduced in a previous paper [4], and is described and expanded here. DCP improves our original checkpointing approach, particularly in terms of memory usage.

The second contribution we describe in this paper is an even better checkpointing technique. In particular, *Lightweight checkpointing* (LCP) further cuts memory usage by one-third and requires less processing overhead. LCP identifies points within the structure of the message where state changes are small, and can be statically predicted. For example, the state of the deserializer just before it is about to deserialize a sub-type of a complex type is very similar to its state before deserializing any other sub-type of that complex type. Figure 1 illustrates this for an *address* complex type, which has a *street*, *city*, *state*, *zip*, and *phone*, respectively, as its sub-types.

LCP represents a significant performance improvement, particularly in terms of memory overhead, over both our original DDS checkpointing mechanism, and our improved DCP mechanism. For example, in some of our test cases, LCP requires only 10% of the memory that DCP requires, and only 3% of the memory that our original algorithm required. In terms of processing time, deserialization with LCP is about 36% better² than DCP and about 52% better than FCP, on average, when approximately half of the message is unchanged from the previous message.

This paper proceeds as follows. Section 2 summarizes the important aspects of our general differential deserialization approach, along with the original checkpointing algorithm (which we call *full checkpointing* (FCP), because the full parser state is stored with each checkpoint). Sections 3 and 4 then describe the design and implementation of differential checkpointing (DCP) and lightweight checkpointing (LCP), respectively. These parser state checkpointing implementations make DDS a viable option for significantly reducing the deserialization overhead associated with SOAP message processing (for streams of messages that exhibit similarity in consecutive messages). Section 5 characterizes the overhead reduction that the new checkpointing

²When comparing performance, we use the word *better* to refer to the improvement percentage, which we compute as $(speedup - 1) \times 100$.

implementations achieve. Section 6 describes related work, and Section 7 summarizes the contributions of this paper.

2 Background

This section describes at a high level, the differential deserialization optimization technique, which is introduced and described in considerably more detail elsewhere [1]. We provide only enough background information to appreciate the contributions of this paper.

2.1 Differential Deserialization

As a DDS-enabled deserializer processes the first SOAP message in a stream, it periodically computes checksums for contiguous portions of the message, and also saves the corresponding full state of the XML parser, at the time the checksum was calculated. When subsequent messages arrive, before parsing them completely, the deserializer first calculates message portion checksums and compares them with saved corresponding checksums from previous messages. If the checksums match, then with high probability the message contents also match, and the deserializer can avoid duplicating the work of parsing and converting the SOAP message contents in that region. To realize this optimization, the deserializer runs in either *regular* mode or *fast* mode, and switches between the two as appropriate, while processing the message. In regular mode, the deserializer reads and processes all SOAP tags and message contents, as a normal SOAP deserializer would, creating checkpoints and corresponding message portion checksums along the way. In fast mode, the deserializer considers the sequence of checksums (each corresponding to disjoint portions of the message), and compares them against the sequence of checksums associated with the most recently received message for the same service.

When all checksums of an incoming message match those stored for a previous message, then the deserializer spends its time computing and comparing checksums, rather than parsing and converting data. This only happens when the entire message exactly matches the previous one. In the more realistic case, the deserializer will discover checksum mismatches while processing a new message; this signals a difference in the incoming message from the corresponding portion of the previous message. When this happens, bSOAP switches from fast mode to regular parsing, and processes and converts that message portion's contents as it would otherwise have had to (without the DDS optimization). bSOAP can switch back to processing the message in fast mode if and when it recognizes that its parser state is the same as one that has been saved in a checkpoint. The worst case occurs when a message differs from the previous message in *all* message portions. When

this happens, a DDS-enabled deserializer runs slower than a normal deserializer, because it does the same work, in addition to calculating checksums and creating parser checkpoints.

Clearly, the effectiveness of the DDS optimization depends on the quantitative difference in the speed of fast and regular parsing modes, which depends on the contents of the incoming message. It also depends on how much time can be spent in fast mode, which in turn depends on the similarity between consecutive messages in an incoming message stream, and how fast the parser can recognize the need or opportunity to switch modes. Mode switching efficiency, in turn, depends in part on checkpoint creation frequency. DDS's component processing costs include checkpoint calculation and comparison, checkpoint creation, and parser state restoration [1].

2.2 Checkpointing and Memory Usage

Beyond the processing overhead described above, checkpointing also consumes memory. The amount of memory depends on characteristics of the incoming message (which is clearly not under the control of the SOAP implementation), and the frequency of checkpoints, which is a configurable parameter of bSOAP's DDS implementation. Reducing memory requirements can lead indirectly to better overall performance by freeing up the memory for other parts of the application (or other processes). It can also allow the SOAP server to create more frequent checkpoints while using the same amount of memory, which can lead to more efficient switching between parsing modes. Therefore, this subsection describes in more detail bSOAP's use of memory for checkpointing and DDS. We describe and characterize only the default DDS implementation (FCP), not DCP or LCP, saving those descriptions for Sections 3 and 4.

DDS requires memory to store the parser's state at different points during message processing. Due to the hierarchical nature of XML, the parser's state consists mainly of stacks. One stack keeps track of string references, and another holds the actual strings³. In addition, bSOAP implements namespace alias mappings in a hash table, and the hash table's nodes are stored on their own stack.

bSOAP uses a *matching stack* for describing the application objects being deserialized. This stack is mainly used to detect when the parser's state matches that of a checkpoint, which is required for switching to fast mode [1]. Also, a *memory blocks* stack keeps track of the dynamically allocated memory blocks during deserialization. Finally, bSOAP's deserializer runs on its own program stack, independent of the calling application's program stack. This

³This separation is needed because some strings may stay in the parsing buffer and never get copied onto the strings stack before they are no longer needed.

is also stored as part of the state.

The standard DDS implementation stores a complete copy of each stack with each checkpoint, despite the fact that some stacks may remain unchanged between two checkpoints, while others experience only a few changes.

3 Differential Checkpointing

As described in Section 2, the most significant portions of a parser's state are stored in various stacks, including the string reference stack, the strings stack, the matching stack, and the stack for namespace aliases. Therefore, performing differential checkpointing is a matter of storing only the differences in stacks at various times while processing an incoming message. To do so, bSOAP's DDS implementation must (i) track the changes that have been made to the stack since the last time it was stored in full, (ii) store the changes to the full stack separately in a partial stack copy, when appropriate, and (iii) be able to use the partial copy to restore the parser state when changes in the incoming message are detected by mismatched checksums. These three requirements are described separately below.

3.1 Tracking Changes

Each stack in bSOAP maintains a pointer to one *base copy* that holds a complete copy of an instance of that stack, which was previously saved. Each stack also maintains a *tracking pointer* that determines how much the current instance of the stack differs from its base copy. Whenever a new complete copy of a stack is created, its base stack copy pointer is set to point to that copy. In addition, the tracking pointer is set to point to the very top of the stack. Whenever data that is located before the tracking pointer is popped off, the tracking pointer is set to point to the new top of the stack. Any new data pushed on the stack does not result in a change to the tracking pointer. Thus, the tracking pointer essentially splits the stack into two portions: a bottom portion that is known to be the same in the base copy, and a top portion that may contain different data than the top portion of the base copy.

3.2 Creating Stack Copies

All created stack copies in a checkpoint are partial copies. A partial stack copy consists of any data that is located in the top portion of the stack. A partial stack copy also contains a pointer to the base copy, which contains its bottom portion and the value of the tracking pointer. Should the size of the bottom portion of the stack fall below a certain threshold, a new full copy is created before the partial copy.

3.3 Restoring a Partial Copy

Restoring a partial copy works by first restoring the bottom portion from the base copy, and then restoring the top portion. In addition, the tracking pointer is set appropriately and the base copy pointer of the stack is set to point to that of the partial copy. When the current instance of the stack and the partial copy have the same base copy, not all data is restored. In particular, the data in the smaller bottom portion—either the partial copy’s or the current stack instance’s—is not restored.

3.4 Deleting Base Copies

Each base copy contains a reference count, which is incremented whenever a partial copy referring to a base copy is created. Whenever a checkpoint is destroyed, the reference count of each base copy of the partial stack copies it contains, is decremented. The base copy is deleted when its reference count reaches zero.

3.5 Comparing State

As part of *progressive matching* [1], namespace-alias mappings and strings stacks must be compared with those of a checkpoint. For any of the stacks that get compared, when both the stack in the checkpoint and the stack in the deserializer share the same base copy⁴, comparison is optimized by not comparing any data that is known to be the same in both stacks. In particular, the data in the smaller of the two bottom portions that the stack in the deserializer and the stack in the checkpoint, respectively, share with the base copy, are not compared.

4 Lightweight Checkpointing

Lightweight checkpoints improve our DCP approach by creating checkpoints at locations that ensure that they are smaller than they would be if they were created arbitrarily. These locations can be statically predicted.

4.1 Checkpoints in bSOAP

Our initial DDS implementation [1] creates *context-independent* checkpoints. As their name implies, context-independent checkpoints can be created at any point in the message, making them especially useful in cases where large, non-structural content occurs within the message (e.g., large base64 or string values).

⁴The base copy of a stack in the deserializer is that of the last restored partial copy of the stack or the last created full copy, whichever occurs last.

The flexibility offered by context-independent checkpoints comes at the price of storing a lot of state information. This not only results in increasing memory requirements, but also induces processing overhead while creating, restoring, and comparing checkpoint states due to the large amount of state information being manipulated.

Lightweight checkpoints, on the other hand, contain very little state information but can only be created at predefined points within the structure of the message. Each lightweight checkpoint has a reference to a *base checkpoint* that contains state information it shares with other lightweight checkpoints. That is, the full state of the deserializer at a point where a lightweight checkpoint is created is defined by the combined state of the lightweight checkpoint and its base checkpoint. Lightweight checkpoints that have the same base checkpoint are said to belong to the same *group*.

4.2 Creating lightweight checkpoints

bSOAP maintains information necessary for creating lightweight checkpoints in two stacks: a *base checkpoints stack* that holds references to base checkpoints for which lightweight checkpoints can still be created and a *lightweight checkpoint location information stack* (or location information stack, for short) that contains last encountered potential locations of lightweight checkpoints in the parsing buffer as well as any lightweight checkpoint state information associated with those locations.

Each deserialization routine for which lightweight checkpointing can be applied (currently, arrays and complex types deserialization routines), allocates entries on both stacks if it decides to create a base checkpoint⁵.

A deserialization routine initially allocates a null entry on the location information stack. Whenever it encounters a potential location of a lightweight checkpoint, it updates its entry on that stack with the location of the checkpoint in the parsing buffer, and with state information associated with the lightweight checkpoint. For example, an arrays deserializer allows lightweight checkpoints to be located just before start tags of XML elements corresponding to the array elements, and stores the array index with them.

Whenever a processing interrupt⁶ occurs, the location information stack is inspected to determine if it is possible to create a lightweight checkpoint (indicated by a non-null entry). The current policy creates an LCP when possible, and skips creating a checkpoint otherwise. However, it

⁵Currently, bSOAP always creates base checkpoints in the very first possible location. We’re planning on incorporating a policy for deciding when and where to create base checkpoints, as an inappropriate allocation of base checkpoints can offset the benefits of lightweight checkpointing.

⁶A processing interrupt occurs when the parser is interrupted to potentially create a checkpoint. We implement this by faking the real number of bytes in the buffer, thus forcing the parser to request more bytes to be read after a specific number of bytes are processed.

never skips creating a checkpoint twice in row—it creates a context-independent checkpoint if no LCP can be created after two consecutive interrupts.

4.3 Restoring an LCP state

Restoring the state in a lightweight checkpoint involves restoring the state in its base checkpoint. This process can be optimized for the case where the base checkpoint existed on the base checkpoints stack before the last switch to fast mode. In this case, almost no state need to be restored. However, data on various stacks needs to be removed. A special case occurs when the last time bSOAP switched to fast mode was at a lightweight checkpoint that belongs to the same group as a lightweight checkpoint it is about to restore. In this case, no state manipulations take place on a switch to regular mode.

To allow creation of lightweight checkpoints after switching to regular mode, both base and context independent checkpoints store the base checkpoints stack and restore it when their state is restored. The entries on the location information stack are nullified on state restoration. This is because they wouldn't contain valid information—they would point to locations that used to be before the location of the restored checkpoint.

4.4 Lightweight checkpoints and progressive matching

Progressive matching is a process by which bSOAP identifies checkpoints at which it may switch to fast mode [1]. The process involves matching the entries of the deserializer's *matching stack* against those stored in a checkpoint. bSOAP completely avoids this process for lightweight checkpoints belonging to the same group.

4.5 DCP vs. LCP Summary

Thus, one difference between LCP and DCP is in how the deserializer decides where to create the checkpoints. Both mechanisms include checkpoint frequency as a configurable parameter. With DCP, this frequency will deterministically create a checkpoint at almost equally sized intervals within the message, once the appropriate number of bytes have been processed. Some of these checkpoints may be in convenient locations, and the checkpoint difference between it and the previous full checkpoint may be small, leading to a differential checkpoint that consumes little memory. However, some of them may happen to fall at locations that make the difference between the last full checkpoint larger.

Even when checkpoints are created at convenient locations with DCP, they may be large when compared to LCP. This is due to DCP's change tracking algorithm favoring

speed over accuracy, as well as the fact that not all of state information changes are tracked with DCP.

With LCP, the interrupt frequency is treated as a hint to create a checkpoint the next time it is strategically convenient to do so. A processing interrupt tells bSOAP to check whether the deserializer has already passed the location of a lightweight checkpoint (which it can tell by inspecting the stacks as described in Section 4.2 above), in which case it may choose to create one. In this case, "strategically convenient" means that creating a checkpoint would be likely to lead to a small (lightweight) checkpoint. In particular, the parser waits until after it completely deserializes a subtype or an array element before creating a checkpoint, then a lightweight checkpoint at that location is likely to be more similar to a checkpoint that was created at the end of a previous subtype of the same type. Similar checkpoints consume less memory and require less processing time to write, thereby reducing DDS overhead.

So LCP works at a higher level and is based on the fact that any *parser* state changes that occur due to processing element $\langle x \rangle$ disappear after processing element $\langle /x \rangle$, so only *deserializer* state is saved in an LCP. So in the tree that models an XML message each set of children may correspond to a group of LCPs (with the exception of one child corresponding to a base checkpoint). DCP, on the other hand, works at a lower level; it tries to *track* changes to the state (rather than determining ahead of time where to create checkpoints) and only store those changes.

5 Performance Study

In this section, we compare the performance of lightweight checkpointing (labelled "LCP" in the figures) with differential checkpointing (labelled "DCP" in the figures) and full checkpointing (labelled "FCP" in the figures) of parser states. We describe our experimental testbed, then report the overhead of creating checkpoints in Section 5.2. Section 5.3 then reports deserialization times for messages that require the parser to switch back and forth between regular and fast parsing modes. Finally, Section 5.4 describes the memory usage for LCP vs. DCP, demonstrating the primary benefit of the LCP optimization.

5.1 Experimental Setup

We ran all performance tests on a single Pentium 4 2.4 GHz machine with 512 MB of RAM, and a 200GB 7200 RPM IDE Drive. The tests started a timer, read a sequence of "incoming" SOAP messages from a file, deserialized them, then stopped the timer just before calling the back-end method. We averaged 100 iterations of the full test to amortize the initial read from the file—the full messages for all but the first message were typically read directly from

cache, allowing us to isolate the deserialization time. We set the parsing buffer size to 32K and compiled with gcc 4.0.2 and optimization flags set to “-O3”. For all DDS tests, we used a software-based CRC32 algorithm for computing checksums. All arrays contain “difficult to deserialize” doubles—randomly generated values that contain an average of 23 characters to encode.

5.2 Checkpointing overhead

This section reports results for bSOAP with DDS checkpointing and checksum creation. We study messages of size 350K, 1.7 MB, and 3.5 MB, where the deserializer is interrupted every N bytes to potentially create a checkpoint. We tested N in $\{32, 64, 128, 256, 512, 1K\}$ ⁷.

Figure 2 shows that the overhead of creating checkpoints is highest with FCP, followed by DCP, and then by LCP. This is because LCP writes less data than DCP, which in turn writes less data than FCP. As the number of checkpoints increases (due to smaller interrupt frequencies and larger messages), the gaps between LCP, DCP and FCP overheads increases.

For this test, LCP is 36.3% and 69.1% better than DCP and FCP, respectively, on average. It is 13.1% worse than bSOAP when compiled without DDS support, on average. This represents the cost of using DDS for message streams that exhibit *no similarity* in consecutive messages; each message is 100% different from the previous one.

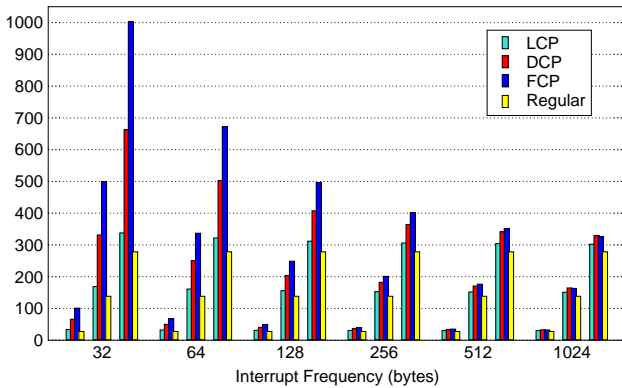


Figure 2. Checkpointing overhead, for various message sizes and interrupt frequencies—deserialization time measured in milliseconds (ms). Each set of bars demonstrates the performance for messages of size 350K, 1.7 MB, and 3.5 MB. The bars in each set are arranged by message size, with the bar corresponding to the smallest message on the far left.

When considering only interrupt frequencies of 32, 64, and 128, the improvement grows to 59.8% and 121.2%

⁷We refer to this value N as the “interrupt frequency.”

over DCP and FCP, respectively, and the overhead grows slightly to 16.6% when compared to bSOAP without DDS support. This improvement means that for the same deserialization overhead, using LCP potentially enables the creation of more checkpoints than both DCP and FCP.

5.3 LCP Performance

Differential deserialization can require that the parser move back and forth between fast processing mode (using and comparing checksums) and regular processing mode (full parsing, while saving checkpoints for future use).

The first part of this test fixes the interrupt frequency, N , at 32, 128 and 512 bytes, respectively, and changes P percent of the array values from message to message, for $P=25$, $P=50$, and $P=75$. For this test, the values that changed were in T separate partitions, scattered uniformly throughout the array.

As shown in Figures 3 and 4, LCP performs better, particularly as the number of created checkpoints increases (generally due to larger values of P , T , and message size, and smaller values of N).

Table 1 shows results for a more comprehensive set of similar tests. The table contains the percentage of performance improvement LCP obtains over bSOAP with no DDS support, for various values of N (interrupt frequencies), T (number of partitions, to influence the number of mode switches), P (percentage of values that change), and array sizes. The numbers in the table were obtained by subtracting 1 from the speedup (T_{LCP} / T_{NoDDS}) and then multiplying by 100. Thus, positive numbers reflect performance improvement of LCP over bSOAP without DDS support, while negative numbers reflect performance degradation.

Table 1 shows that checkpointing overhead is no longer a major bottleneck with LCP. In fact, in previous tests [1], we have shown that fully processing a 100K-element hard doubles array in fast mode using the CRC32 checksumming algorithm performed within 6% of bSOAP without DDS support deserializing the same array. The results for a 100K-element array and P of 100 demonstrate that most overhead lies in the checksumming algorithm, as bSOAP would have to compute checksums of portions twice—once to detect a checksum mismatch and once to compute the checksum of the new content of a message portion. As a result, LCP allows DDS to benefit from increasing the number of checkpoints created, which generally results in less time spent in regular mode. Also, smaller message portion sizes means that we can use faster checksumming algorithms.

5.4 Memory Usage

The primary benefit of differential checkpointing is its memory efficiency. Figure 5 demonstrates the difference

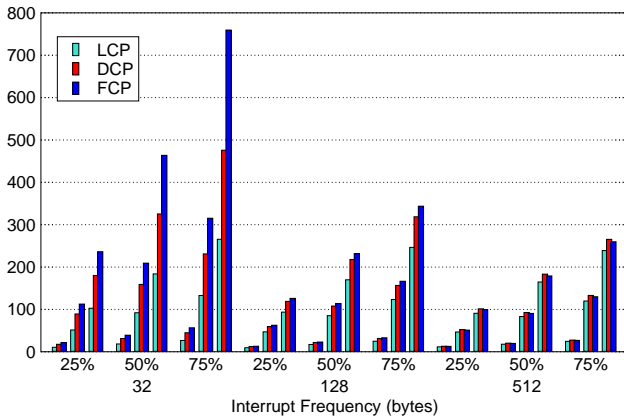


Figure 3. Deserialization time, in milliseconds, for messages divided into 50 partitions ($T=50$). Three different percentages of the values ($P=25$, $P=50$, and $P=75$) around the center of each partition are changed, and three interrupt frequencies ($N=32$, $N=128$, and $N=512$) are tested. Starting from the left in each set of bars, the bars correspond to arrays of 50K, 75K, and 100K elements, respectively.

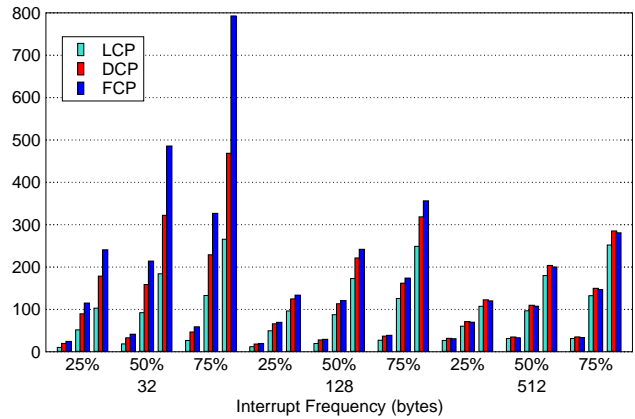


Figure 4. Deserialization time, in milliseconds, for messages divided into 500 partitions ($T=500$). Other test parameters identical to those for Figure 3.

in memory requirements for LCP vs. DCP⁸. Clearly, lightweight checkpointing reduces memory requirements considerably. LCP uses only about 3% of the memory that FCP uses, and 10% of the memory that DCP uses. This extra memory could be used to create more frequent checkpoints, thereby increasing the effectiveness of differential deserialization in general, or to allocate to other parts of the application or other processes.

6 Related Work

Deserialization converts XML streams in wire-format to objects in memory. This process consists of four steps; reading the XML stream from the socket layer into a buffer, parsing the XML stream into tokens, converting data types in ASCII to binary formats, and populating data structures with the received values. The widely used paradigms for parsing XML documents include Document Object Model (DOM) [5], Simple API for XML (SAX) [6], and XML Pull Parser (XPP) [7]. Other recent research on high performance parsers investigates the advantages of using schema specific parsers [8].

The DOM model maps the XML document into a tree representation in memory. This enable easy traversal and modification. However, for large documents, DOM parsing can be memory intensive. In contrast, SAX parsing never stores the entire XML document in memory. Instead, it emits events for all the document’s elements and tags. For large static documents, SAX is preferable to DOM. SAX is

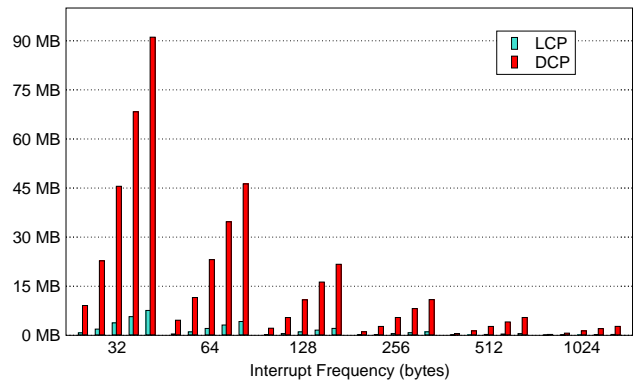


Figure 5. Memory requirements, for various message sizes and interrupt frequencies—memory used, measured in Megabytes (MB). Each set of bars demonstrates the performance for messages for arrays of doubles ranging in size from 10K to 100K. The pairs of bars in each set are arranged by array size, with the pair corresponding to the smallest array being on the far left.

also often used when only a few specific elements of an XML document need to be extracted and processed. *Pull parsing*, employed by the XPP [7] parser, is specialized for parsing payloads in which elements are processed in succession, and no element needs to be revisited. XPP provides the added feature of incrementally building a partial XML Infoset tree in memory. As a result, applications can process the document as it is being parsed, without waiting for the entire document to be parsed.

bSOAP contains a pull parser that incorporates many optimizations found in other parsers, including avoiding unnecessary string copies. One unique feature of bSOAP’s XML parser is that it contains parsing routines that are optimized for the case where the expected content of the message is known a priori (e.g., for SOAP messages). In this

⁸For clarity of presentation, we do not show FCP memory requirements in this graph.

		Number of partitions (T)															
		1				100				200				500			
$P\%$	Arr. Size	Interrupt frequency (F)				Interrupt frequency (F)				Interrupt frequency (F)				Interrupt frequency (F)			
		32	128	256	512	32	128	256	512	32	128	256	512	32	128	256	512
25%	25000	164	192	192	195	165	89.4	11.6	-11	213	56.2	-10	-11	163	26.0	-12	-11
	50000	168	194	203	205	166	178	169	154	165	165	115	63.3	165	130	58.5	2.70
	75000	170	197	205	207	170	190	189	170	171	185	170	135	169	168	121	67.6
	100000	169	197	205	207	168	192	196	185	169	190	188	169	167	179	163	129
50%	25000	50.1	62.3	65.8	65.0	50.6	26.2	-10	-10	49.9	3.39	-11	-10	48.8	-5.3	-11	-9.8
	50000	48.9	61.8	67.0	67.6	49.3	57.1	52.3	50.4	49.0	52.3	38.9	17.7	48.9	40.6	15.0	-12
	75000	50.9	63.6	68.1	68.6	50.7	61.8	62.2	56.5	50.1	59.4	57.8	44.6	50.6	54.2	41.4	19.2
	100000	50.8	63.5	67.7	68.2	50.1	62.2	65.0	64.9	50.3	61.1	61.9	56.5	50.0	58.1	53.9	43.1
75%	25000	5.03	12.9	15.1	17.0	4.89	-5.8	-11	-10	11.7	-8.9	-11	-10	4.13	-10	-11	-10
	50000	3.65	11.9	14.9	15.2	3.71	9.47	8.29	4.20	3.66	7.24	2.89	-12	3.26	1.17	-12	-12
	75000	4.69	12.9	15.9	16.2	4.46	11.8	13.4	11.3	4.73	11.2	11.2	5.41	4.60	8.36	4.16	-11
	100000	4.33	12.5	15.5	15.7	4.27	12.0	14.1	13.9	4.16	11.4	13.0	11.1	4.18	10.1	9.25	4.50
100%	25000	-19	-13	-11	-11	-19	-13	-11	-10	-19	-13	-11	-10	-19	-13	-11	-11
	50000	-21	-15	-12	-12	-21	-15	-12	-12	-21	-15	-12	-12	-21	-15	-12	-12
	75000	-20	-14	-12	-11	-20	-14	-11	-11	-20	-14	-11	-11	-20	-14	-11	-11
	100000	-20	-14	-12	-12	-20	-14	-12	-12	-21	-14	-12	-12	-20	-14	-12	-12

Table 1. Percent performance improvement in deserialization times of LCP over regular deserialization, for various array sizes, percentages of values that change from message to message (P), message partitions (T), and interrupt frequencies (F). Percent improvement is calculated as $100 \times (T_{LCP} / T_{NoDDS} - 1)$. Negative numbers indicate the few cases where LCP performs worse than regular deserialization.

case, the parser requires only a single pass over the document content.

7 Summary

We describe two SOAP parser state checkpointing algorithms, called differential and lightweight checkpointing (DCP and LCP respectively). Each supports our implementation of differential deserialization (DDS), which allows SOAP parsers to avoid repeating the expensive conversion of ASCII-based XML to binary in memory representations, when a SOAP server receives a stream of consecutive similar messages. The conversion is replaced by the relatively smaller cost of comparing bytes and calculating checksums. Our new checkpointing algorithms reduce the number of bytes that need to be stored. LCP, for example, requires only 3% of the memory required by our original checkpointing algorithm. This reduction comes as a result of often storing only the differences between checkpoints, rather than a sequence of complete, self-contained checkpoints. Writing fewer bytes also reduces processing overhead, making differential deserialization a viable mechanism for avoiding the SOAP deserialization bottleneck.

References

- [1] N. Abu-Ghazaleh and M. J. Lewis, "Differential Deserialization for Optimized SOAP Performance," *Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC 2005)*, p. 21, November 2005, Seattle, Washington.
- [2] K. Chiu, M. Govindaraju, and R. Bramley, "Investigating the Limits of SOAP Performance for Scientific Computing," in *IEEE HPDC-11*, Edinburgh, Scotland, July 23-26, 2002, pp. 246-254.
- [3] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju, "Differential Serialization for Optimized SOAP Performance," *IEEE HPDC-13*, pp. 55-64, June 2004, Honolulu, Hawaii.
- [4] N. Abu-Ghazaleh and M. J. Lewis, "Differential Checkpointing for Reducing Memory Requirements in Optimized SOAP Deserialization," *Grid 2005 Workshop (short paper)*, pp. 250-255, November 2005, Seattle, WA.
- [5] World Wide Web consortium, "Document object model," <http://www.w3c.org/DOM>.
- [6] David Megginson et al., "SAX 2.0.1: The Simple API for XML," <http://www.saxproject.org>.
- [7] Indiana University, Extreme! Computing Lab, "Grid Web Services," <http://www.extreme.indiana.edu/xgws/>.
- [8] K. Chiu and W. Lu, "A Compiler-Based Approach to Schema-Specific Parsing," in *First International Workshop on High Performance XML Processing*, 2004.