Designing a Trace Format for Heap Allocation Events

Trishul Chilimbi
Microsoft Research
One Microsoft Way
Redmond, WA, 98052 USA
trishulc@microsoft.com

Richard Jones
Computing Laboratory
University of Kent
Canterbury, Kent, CT2 7NF
R.E.Jones@ukc.ac.uk

Benjamin Zorn
Microsoft Research
One Microsoft Way
Redmond, WA, 98052 USA
zorn@microsoft.com

ABSTRACT

Dynamic storage allocation continues to play an important role in the performance and correctness of systems ranging from user productivity software to high-performance servers. While algorithms for dynamic storage allocation have been studied for decades, much of the literature is based on measuring the performance of benchmark programs unrepresentative of many important allocation-intensive workloads. Furthermore, to date no standard has emerged or been proposed for publishing and exchanging representative allocation workloads. In this paper, we describe a preliminary design of a trace format for such workloads and investigate its effectiveness at representing large allocation traces. Our proposal allows for a flexible encoding of information in the trace to achieve greater compression. We evaluate our preliminary design in two dimensions. First, we measure how effective these encodings are at reducing trace size. Second we consider how a meta-level specification language could be used to describe such formats and to generate trace readers and writers automatically.

1. INTRODUCTION

Dynamic storage allocation[1] continues to be an important part of many computer systems. Algorithms for DSA have been proposed and evaluated for decades, resulting in highly effective implementations. New research opportunities continue to emerge such as the scalability of allocator implementations in multithreaded systems [1, 14]. Unfortunately, much of the research in this field is based on the allocation behavior of relatively short-running benchmarks (for example, work by one of the authors [2, 23] and others [12]). These programs are becoming less appropriate for DSA research for several reasons. First, often DSA performance only becomes an issue after large amounts of allocation have been performed. Existing research is often based on programs in which at most several million objects are allocated, when actual performance problems may develop after tens or hundreds of millions of allocations. Second, single-threaded, single-heap benchmarks are not representative of a large and important class of multithreaded applications that often rely heavily on DSA [14]. Consequently, improving DSA performance remains a significant challenge for many such programs.

Large-scale multithreaded benchmarks are not typically used in research for several reasons. Publicly available versions of such long-running, allocation-intensive programs are difficult to find and/or are proprietary. In addition, long-running programs are inherently more difficult to work with experimentally. Server programs, an interesting subclass of such programs, are difficult to create workloads for, because they require configuring networks and clients, and often execute non-deterministically.

To overcome these difficulties, we propose that collecting and sharing large allocation traces is an attractive alternative to the ad hoc benchmarking that is currently taking place. This paper proposes and evaluates one possible format for such traces, and discusses a meta-language for trace format specification. Our goal is to encourage broader involvement in this effort. We view this paper as a starting point in a discussion about whether a trace-based approach to DSA research is effective and feasible, and if so, what the most appropriate format for such traces would be. Designing a trace format raises several issues. The following goals were used to guide the design presented here:

- Expressiveness—The trace format must be able to express enough information that the resulting traces can be used to make research contributions in the field.
- Compactness—Compact encodings allow larger, more representative traces to be created and shared. With current disk and network technology, we would like to be able to conveniently create and share traces that contain 100 million allocation events.
- Flexibility—Our initial design will not meet all the possible needs of researchers. Perhaps the most important goal is to design with the understanding that additional information (and potentially entirely new formats) will be needed.

[1]We use the terms “dynamic storage allocation,” “DSA,” and “heap allocation” as synonyms in the text.

Record Type	Description
alloc	Allocation record with fields: size, address, thread, heap, time, and attributes.
free	Deallocation record with fields: address, thread, heap, time, and attributes.
realloc/no alloc realloc/alloc free realloc/alloc realloc/free	Reallocation record with fields: size, old address, new address, thread, heap, time, and attributes. Realloc tags encode four possible outcomes: malloc with no free, free with no malloc, free then malloc, and no malloc or free. If malloc and/or free are called from realloc, records for these operations occur in the trace in addition to the realloc record.
createHeap/destroyHeap	Heap creation/destruction records with fields: heap, thread, time, and attributes.
createThread/destroyThread	Thread creation/destruction records with fields: thread, time, and attributes.
comment	Comment record with a varying length string field.

Table 1: HATF Record Types

Metadata operation	Description
set fieldSize *field width* where *width* indicates	0, 1, 2, 4, or 8 bytes. Attribute fields can be of varying size with a 1- or 2-byte size header.
set fieldInterp *field interp* ... where *interp* indicates	None => use field value directly Default *defaultValue* => use *defaultValue*, *width* is 0 bytes BaseOffset *base* => use field value plus *base* Delta *initialValue* => use field value plus previous field value Stride *initialValue stride* => use previous field value plus *stride*, *width* is 0 bytes

Table 2: HATF Metadata Operations

Keeping these goals in mind, this paper makes the following contributions:

- We have developed HATF (Heap Allocation Trace Format), version 1.0. HATF is a binary format that includes inline metadata that can dynamically modify the encoding of trace events. We see HATF 1.0 as a starting point in an open discussion about such formats. It has been implemented, and we intend to make the source code for a reader and writer publicly available.
- Using our preliminary implementation, we investigate the effectiveness of mechanisms to reduce trace size. Specifically, we consider how effective our approach is at compressing the traces over and above traditional techniques.
- Based on our experience with HATF, we are now developing Meta-TF, an easy-to-read meta-level specification of formats such as HATF that allows readers and writers of HATF-like formats to be generated automatically. By analogy, Meta-TF allows the HATF trace format to be specified and documented in the same way that DTDs in XML allow the HTML format to be specified. Our work with Meta-TF is an attempt to reduce the effort to design and implement trace formats while increasing their expressiveness and shareability.

The remainder of the paper considers these items in turn. Section 2 describes HATF 1.0. Section 3 presents results indicating the effectiveness of the HATF 1.0 design at representing traces. Section 4 motivates and describes our preliminary design of Meta-TF. Section 5 describes related work in the areas of trace formats and format specifications. Section 6 summarizes the paper.

2. HATF 1.0

2.1 Trace contents

A specification of the HATF 1.0 design is provided in Appendix A. This section describes the general characteristics of the format and attempts to explain the design.

We begin by stating our assumptions. We believe that long trace files (representing hours or even days of application execution) are neccessary to understand the performance implications of memory allocator design. The implication of this assumption is that trace compactness is an important consideration.

Our view is that the most important use of trace files will be to drive an allocator implementation for the purpose of understanding its performance characteristics. For this purpose, the majority of the trace needs to be read sequentially and incrementally[2]. We do not forsee the need to keep the entire trace in memory at any one time, nor do we see a need to support random access to parts of the trace. In typical use, a trace will be stored in a compressed form, decompressed and piped to a program that reads one trace record at a time and processes it.

[2]The trace might contain additional information, such as the set of types allocated or the set of allocation call stacks observed. Such information is likely to be read entirely and cached in memory. In general, however, we believe this information will occupy a small part of the total trace.

We believe that trace accessibility is also an important consideration. By accessibility, we mean the ease with which a trace can be used by a researcher. Our goal is to produce traces with a low "barrier-to-entry" for researchers so that they have more reasons to use them in experiments. A central decision in trace format design revolves around the choice of an ASCII or binary representation for the trace. In an ASCII form, a trace can be easily scrutined or generated using any text editor. Also, such traces are easily manipulated using text-oriented scripting languages such as Perl. A text-based structured representation such as XML might provide additional benefits, enabling traces to be manipulated using generic XML tools such as browsers, etc. Finally, ASCII provides a more portable format, as issues of byte-order are not relevant as they are in a binary format.

Having explained these benefits of an ASCII encoding, we have still chosen a binary encoding for our design. The specific advantage of a binary format is the compactness of the encoding. We reason that the advantages of an ASCII encoding mentioned above mostly do not apply to large DSA traces. In particular, we see no substantial benefit in allowing traces to be viewed in a text editor, as they will likely be stored in a compressed format in any case. Furthermore, we see little motivation to browse or edit traces directly, as a typical usable trace has millions of events, and making meaningful hand-edits to them would be very difficult. Although a binary format prevents text-oriented scripting languages, such as Perl, from being used to manipulate the traces, our solution is to provide a binary-to-ASCII translator that can be used on our traces. Using a pipe, a trace can be extracted and processed by Perl with relatively little overhead. In a later section, we discuss the performance cost of reading and writing ASCII traces using Perl.

Prior work in the design of allocation trace formats is typified by Johnstone's PhD research on fragmentation in allocators [11]. Johnstone used traces composed of malloc, realloc, and free operations. We call a trace event associated with these operations a trace record, or simply, a record. For malloc, Johnstone recorded the size requested and the address returned, for free the address of the object freed, and for realloc the address and size passed as arguments. We refer to these values as fields within each record. Johnstone's dissertation does not mention the specifics of how this information was represented in the trace.

To achieve greater expressiveness, HATF expands the number of record types and the fields within each record beyond those used by Johnstone. Specifically, we note that systems sometimes provide an extended storage allocation API beyond the traditional Unix-specific malloc/free/realloc operations. For example, the Win32 API allows individual heaps to be created, allocated from, and removed using the HeapCreate/HeapAlloc/HeapDestroy procedures. Other allocators, such as Vo's vmalloc implementation, also provide an interface to create multiple heaps [21]. HATF provides records to identify the creation of such heaps and the allocation of objects within a particular heap. In addition, in multithreaded applications, specific heaps may be associated with individual threads of execution. HATF provides records that identify the creation and deletion of threads and allows heap operations to be associated with specific threads. HATF 1.0 currently defines the following field types: size, address, heap, thread, time, and attributes. Each record has an associated variable-length attributes field that allows arbitrary additional information to be associated with each record. The remaining fields are interpreted as unsigned numeric data. Table 1 summarizes the record types supported by HATF.

2.2 Trace representation

The previous section described the contents of a HATF trace but not its representation. We have attempted to make HATF's representation of data as flexible as possible by allowing *metadata records* that describe the representation of data in the fields of subsequent data records. Specifically, the width and interpretation of each field type can be set dynamically with metadata records. Table 2 summarizes the metadata operations supported by HATF.

Before describing the specifics of the design of HATF metadata, we discuss the motivation that led to our decisions. Specifically:

- We wanted to allow the format to represent traces gathered from different machines, and as a result, wanted to allow 4- and 8-byte addresses to appear in a given trace. Thus the need to include metadata indicating how many bytes a field occupied.
- Often fields would be always empty (e.g., the thread field in the case where a trace is taken from a single-threaded application). In these cases, we wanted to be able to omit the field entirely from the record.
- Some of the fields, such as the size field, have very skewed distributions. For example, in most cases, a single byte is sufficient to encode the size of the object allocated. Likewise, the time field is generally monotonically increasing.
- Often there are regular patterns in the traces, such as the object address returned from malloc being a regular increasing sequence of addresses.

These thoughts led us to an approach in which the field width and interpretation could be defined and changed dynamically. A particular setting of field width and interpretation remains in effect until another metadata record changes the setting; thus there is no "nesting" of metadata settings.

The field interpretation indicates how the value of a field in a given record should be interpreted. If the value is stored directly in the field, the "none" interpretation is used. The "default" interpretation indicates that the field is entirely omitted from the record, and the value of the field is the value provided in the `set fieldIntpretation` record. The "BaseOffset" interpretation captures cases where there is substantial locality in the trace and many addresses can be encoded as small offsets from a base address (such as the base of the heap). The "Delta" interpretation encodes values as offsets from the previous value of a field. The "Stride" interpretation can encode a series of regularly increasing values without requiring a field in each record.

The HATF design allows traces to be compressed using metadata that changes field widths and interpretations. For

Program	Total Allocs ($\times 10^3$)	Total Reallocs ($\times 10^3$)	Total Frees ($\times 10^3$)	Total Bytes ($\times 10^3$)	Max. Objects ($\times 10^3$)	Max. Bytes ($\times 10^3$)	Avg. Obj. Size Bytes
espresso.largest	1675	16	1675	106813	4.45	268	63.8
espresso.test2	4483	48	4483	1114462	4.91	375	248.6
vortex	1489	0	1412	700967	157	47956	470.8
twolf	575	0	493	17920	105	2657	31.2

Table 3: Summary information about the memory allocation behavior for each of the test programs.

Program	Normalized Trace Size Bytes/Data Record (Uncompressed/Compressed)					
	Simple Binary	ASCII	HATF (naïve)	HATF (best)	Split Stream	HATF SplitAddr
espresso.largest	16 / 2.61	11.56 / 1.74	7.03 / 1.57	5.64 / 1.47	13 / 1.25	5.64 / 1.15
espresso.test2	16 / 2.82	11.57 / 1.61	7.03 / 1.47	5.68 / 1.39	13 / 1.27	5.68 / 1.16
vortex	16 / 3.38	13.64 / 3.11	7.05 / 2.86	5.98 / 2.82	13 / 2.52	5.98 / 2.47
twolf	16 / 1.67	12.56 / 2.37	7.15 / 2.31	5.30 / 1.98	13 / 1.31	5.30 / 1.39
average	16 / 2.62	12.33 / 2.21	7.06 / 2.05	5.65 / 1.92	13 / 1.59	5.65 / 1.54

Table 4: Compression Factor for Various Encodings

example, because sizes are so highly skewed, traces can be compressed by using only one byte to encode the size field for most records. In cases where a larger size field is necessary, metadata that expands and contracts the width of the size field can be inserted around the necessary records. Likewise, in a multithreaded application, if many consecutive records are related to the same thread, the default interpretation for the thread field can be set at the point in the trace when the thread is entered, and changed at the point when the thread is exited. Note that traditional compression techniques are possible above and beyond metadata encodings. One may wonder if the flexible encodings provided in HATF 1.0 provide any benefit at all over and above standard compression.

3. EVALUATION OF HATF DESIGN

This section examines the effectiveness of HATF in encoding traditional allocation traces. We have gathered a collection of traces summarized in Table 3. Although these programs do not use multiple heaps or threads, and as a result have allocation traces that are relatively simple compared to the kind for which HATF is intended to be used, we still believe they represent a good starting point for evaluating the effectiveness of HATF's encoding, both in terms of trace size and the cost to read and write a trace. All experiments were performed on a single processor of a dual-processor 500 MHz Pentium III Xeon system with 512 MB main memory running Windows 2000 Server. The times shown are the average of five executions, with a range of variation less than 1%.

In the tables that follow, we compare HATF with several alternative methods for representing traces. "Simple Binary" is an unsophisticated binary representation of the trace where all records have four 4-byte fields. "ASCII" is an ASCII encoding of the traces. "HATF (naïve)" uses HATF without any effort to compact traces by means of metadata records, while "HATF (best)" represents the most effective, albeit simple, combination of compaction techniques that we considered. We describe the methods used in HATF (best) in Section 3.3. "Split Stream" attempts to get better compression by separating the 4 data fields in the simple binary format into different streams that are compressed independently. The "HATF SplitAddr" encoding splits all unencoded addresses (i.e., addresses not encoded by BaseOffset or Stride field interpretations) from the rest of the HATF (best) trace, and compresses the two streams independently. We consider each approach both with and without Lempel-Ziv compression, as implemented in the gzip utility [22].

3.1 Trace size

Table 4 presents the effectiveness of the different approaches to compressing the trace. The table shows that without gzip compression, the HATF (best) encoding is the most efficient, requiring 5.6 bytes per record (bpr) on average. The metadata encodings in HATF (best) result in a 21% reduction in trace size over HATF (naïve). With compression, the HATF SplitAddr approach is most effective in all but one of the applications, with 1.5bpr on average. Compressing the HATF address stream independently reduces HATF traces approximately 25% overall as compared to compressed HATF (best). Without splitting out the address stream, the metadata encodings in HATF (best) result in a 7% post-compression reduction in trace size over using naïve HATF.

ASCII is a more compact encoding than simple binary because the tag and size field in most records are represented in less than the four bytes used in the binary format. In addition, the compressed ASCII encoding is smaller than the compressed binary encoding as well.

This data suggests that splitting allocation traces into independently compressed streams results in substantial com-

Program	Normalized Trace Processing Time μs/Data Record (Read/Write)						
	Simple Binary	ASCII	HATF (naïve)	HATF (best)	Split Stream	HATF SplitAddr	ASCII/ Perl
espresso.largest	5.29 / 0.20	8.07 / 6.29	6.94 / 1.05	6.91 / 1.18	6.1 / 1.08	7.04 / 1.34	16.83 / 10.01
espresso.test2	5.15 / 0.21	7.93 / 6.29	6.79 / 1.04	6.77 / 1.20	5.94 / 1.09	7.02 / 1.36	16.67 / 10.15
vortex	7.85 / 0.54	10.96 / 7.43	9.66 / 1.18	9.67 / 1.33	8.75 / 1.06	9.59 / 1.30	16.94 / 10.12
twolf	6.59 / 0.34	9.51 / 6.87	8.15 / 1.17	8.15 / 1.43	7.42 / 1.01	8.06 / 1.21	17.19 / 9.82
average	6.22 / 0.32	9.12 / 6.72	7.88 / 1.11	7.88 / 1.28	7.05 / 1.06	7.93 / 1.30	16.90 / 10.03

Table 5: Normalized Time to Read/Write Uncompressed Traces

Program	Normalized Compressed Trace Processing Time μs/Data Record (Read/Write)					
	Simple Binary	ASCII	HATF (naïve)	HATF (best)	Split Stream	HATF SplitAddr
espresso.largest	6.56 / 7.99	8.82 / 11.63	7.44 / 5.77	7.55 / 6.32	7.05 / 5.17	7.41 / 5.30
espresso.test2	6.0 / 7.97	8.67 / 11.42	7.19 / 5.58	7.19 / 6.09	6.84 / 5.48	7.42 / 5.55
vortex	9.06 / 8.00	11.93 / 14.59	10.42 / 9.23	10.41 / 9.68	10.12 / 5.98	10.28 / 5.09
twolf	7.91 / 4.58	10.45 / 12.58	9.11 / 8.52	9.0 / 9.11	8.88 / 4.32	9.11 / 12.69
average	7.38 / 7.14	9.97 / 12.56	8.54 / 7.28	8.54 / 7.80	8.22 / 5.24	8.56 / 7.16
average difference	1.16 / 6.81	0.85 / 5.84	0.66 / 6.17	0.66 / 6.52	1.17 / 4.18	0.63 / 5.86

Table 6: Normalized Time to Read/Write Gzip Compressed Traces. The final row, "average difference", indicates the difference between the average uncompressed time and the average compressed time.

pression benefits, and that the metadata encoding provided by HATF provides additional reductions.

3.2 The time to process the traces

Tables 5 and 6 present the overhead of reading and writing the uncompressed and compressed versions of the traces, respectively. In Table 5, we have added the "ASCII/Perl" encoding column, which indicates the processing time per record to read and write an ASCII trace using Perl. Several things are immediately clear from the table. First, the overhead to write the traces is usually substantially lower than the overhead to read. We believe this is because the application is stalled waiting for the I/O to complete during the read operation, while the application can proceed without waiting in the case of writing. In addition, the ASCII encoding has significantly higher time requirements than the other encodings, due to the need to translate the binary values to and from ASCII. With the ASCII/Perl combination, we see even higher overhead. Between the read and write numbers, the time to read a record is typically more important because our expectation is that trace files will be written once and read many times. As a result, due to space limitations, we limit our discussion to the read times in the remainder of this section.

As expected, Simple Binary is the fastest of the uncompressed traces to read (Table 5). Split Stream is somewhat slower because multiple files need to be opened and read from simultaneously. Other than ASCII, HATF is the slowest to read because it requires field interpretation (a lightweight compression method).

The final row of Table 6 indicates the difference between the average time to read an uncompressed and compressed record in each format — a measure of the additional overhead decompression requires. Simple Binary is still the fastest but decompressing it takes a large amount of time since its trace file is the largest. HATF adds the least additional overhead due to decompression because its files are quite compact. Finally, Split Stream benefits from having a small trace file to read. However, because it requires I/O operations from 4 independent streams per record, it adds additional overhead that HATF does not, raising the cost of decompression to 1.17μs/record, on average.

In summary, the overheads to read the compressed and uncompressed traces are on the order of 6.2-8.6μs/record for the non-ASCII encodings, an encouraging result given that our HATF implementation is currently untuned. While the encoding method used makes some difference, we believe that space costs are the more important factor in determining which encoding method to use.

3.3 Benefits of different interpretations

This section discusses the methods examined to help reduce the size of HATF traces using the metadata records. We call the most effective combination of these methods "HATF (best)" in the previous tables. The processing time is not considered, as it varies little from method to method. Table 7 presents the effectiveness of heuristic compression methods that encode the Size field: "Naïve" does no encoding; "Small Size" initially sets the size field to 1 byte width, but inserts metadata records to expand and contract the field when required; "Same Size" inserts metadata records at the beginning and end of runs of the same size to switch between a default value and "none", provided that the run is long enough to amortize the additional cost of metadata records; and "Combined Size" combines both size heuristics.

"Small Size" gains the greatest benefit, reducing the size field by a factor of four in many cases. "Same Size" provides less benefit, especially when combined with the "Small Size" heuristic, since smaller Size records require larger "Same Size" runs to amortize the cost of the meta-data records. More detailed data[3] indicates that the "Same Size" heuristic will benefit from smaller meta-data records (6-8 bytes rather than the current 12 bytes). With compression, the "Same Size" encoding actually provides no benefit, presumably because gzip compression is effective at identifying and compressing the same regularity in the traces.

Table 8 presents the effectiveness of different approaches to address field encoding. We had two intuitions. First, we felt that addresses were likely to have locality that a Base-Offset interpretation could exploit. Second, we suspected that sequences of allocations of the same size would result in address sequences that the Stride interpretation could capture.

Table 8 compares the following approaches. "BaseOffset 1-byte" encodes addresses with 1 byte width BaseOffset interpretation as appropriate, "BaseOffset 2-bytes" is similar but uses 2 bytes; "Best BaseOffset" combines the 1 and 2-byte approaches; "Stride" detects and encodes increasing sequences of addresses in a zero-width field; and "HATF (best)" combines the most effective size and address encodings.

Our attempts at compression via address encoding are mostly ineffective. In some specific cases, such as `twolf`, compression benefits are modest (8%) using the Stride and BaseOffset 2-bytes methods, but overall the benefits are quite small. Again, as in the case of Same Size, smaller meta-data records would permit more compact encodings as many "BaseOffset" and "Stride" address runs in the trace are fairly short. In combination with the size encodings, the address encodings reduce the compressed trace by 3% on average over that achieved by the size encodings alone. We conclude that HATF 1.0's relatively simple interpretations are insufficient to significantly compact addresses found in the allocation traces. This explains why "HATF SplitAddr", which relies on gzip compression for compacting unencoded addresses, produces the smallest trace.

3.4 Discussion

Based on the timing and compression results in this section, we reach several conclusions. First, allowing field interpretations and widths to change based on dynamic metadata is effective at compressing the size fields in the traces, but less so for address fields. Further, separation and compression of the address stream independently from the rest of the trace results in substantial additional reductions in trace size. HATF encodings require modest additional overhead to decode, but we believe the extra processing time is justified by the expressiveness and flexibility the format provides as well as the reduction in trace size.

The HATF 1.0 Specification defined in Appendix A does not include any mechanisms to support splitting out uninterpreted data addresses. Clearly, however, any number of domain-specific trace formats could benefit from mechanisms that would allow specific trace fields to be split and compressed independently. In Section 4, we propose Meta-TF, a meta-specification language that can be used to specify HATF 1.0. Meta-TF also provides facilities that allow arbitrary trace formats to be defined in such a way that independently compressing different streams of a trace is directly supported by the format.

4. META-TF SPECIFICATION

4.1 Limitations of HATF

Different researchers will undoubtedly have different requirements of traces. Program traces are used for a variety of purposes: event traces (of which heap events are just one example), state snapshots (e.g., capturing a part of the heap), debugging, profiling and so on. HATF is a domain-specific format, and hence is limited in a number of dimensions. Even for traces of heap events, it does not provide all the facilities that researchers may need. Its fixed set of record types do not capture such heap events as heap expansion, read and write barriers, promotion by a generational garbage collector or object loads and stores. Its records contain a fixed set of fields (although fields may be omitted from a record by assigning them a zero width), but there may be a need for new record fields (e.g., to capture the type of an object in an allocation event). Although the variable length 'attributes' field provides flexibility, it is a cumbersome mechanism, especially if it is known *a priori* precisely which fields a particular record type requires.

Despite the range of flexible field encodings and interpretations provided, there will inevitably be others that it does not support. For example, it would be wasteful to store full type names in each allocation record but HATF provides no clean way of enumerating these names more compactly. We return to this point later.

4.2 A higher level solution

Although HATF does not meet the requirements of a general trace format standard (even for heap events), it is a satisfactory *instantiation* of such a standard. One solution might be to allow extra records (and/or fields) to be defined, and to provide a structure for doing this. But, if the goal is to create a standard format for exchangeable trace files, who should define additional record types? How are these formats to be documented? How is the standard to be maintained as a standard?

Rather than attempt to propose a concrete, if extensible, format for all possible traces[4], we propose a solution inspired by the structured document community —for a trace is simply a structured document— that is more flexible yet still allows compact encoding of traces. We propose a higher-level,

[3]Not included due to space constraints.

[4]Or even all possible traces of heap events.

Program	Normalized Trace Size Bytes/Data Record (Uncompressed/Compressed)			
	Naïve	Small Size	Same Size	Combined Size
espresso.largest	7.03 / 1.57	5.67 / 1.47	6.96 / 1.6	5.67 / 1.48
espresso.test2	7.03 / 1.47	5.7 / 1.38	7.00 / 1.47	5.70 / 1.38
vortex	7.05 / 2.86	5.98 / 2.85	6.66 / 2.86	5.93 / 2.86
twolf	7.15 / 2.31	5.56 / 2.15	6.94 / 2.30	5.51 / 2.15
average	7.06 / 2.05	5.73 / 1.96	6.89 / 2.06	5.70 / 1.97

Table 7: Effectiveness of Size Encodings

Program	Normalized Trace Size Bytes/Data Record (Uncompressed/Compressed)				
	BaseOffset 1-byte	BaseOffset 2-bytes	Best BaseOffset	Stride	HATF (best)
espresso.largest	7.00 / 1.57	7.01 / 1.57	7.00 / 1.57	7.03 / 1.56	5.64 / 1.47
espresso.test2	7.01 / 1.47	7.02 / 1.47	7.01 / 1.47	7.03 / 1.47	5.68 / 1.39
vortex	7.05 / 2.86	7.05 / 2.85	7.05 / 2.84	7.05 / 2.86	5.98 / 2.82
twolf	7.15 / 2.26	7.02 / 2.14	7.02 / 2.17	6.89 / 2.14	5.30 / 1.98
average	7.05 / 2.04	7.02 / 2.01	7.02 / 2.01	7.00 / 2.01	5.65 / 1.91

Table 8: Effectiveness of Address Encodings

meta-language for trace format specification, *Meta-TF*, by distinguishing (i) the *trace* from (ii) a *Document Type Definition* (DTD) that specifies the format of that trace.

4.3 A trace meta-specification language

We believe that it is sufficient for trace data simply to comprise a list of records and for each record type to be composed of a fixed number of fields. The format of these records is described by a DTD. A meta-specification language must therefore define how the DTD should

- enumerate the record types that may appear in a trace, and the fields that each record type contains.
- define the format and properties (including, in some cases, the value required) of each field.

In addition, some automated support for the definition of semantic relationships between the fields of a record is useful. Clearly, it is not possible to define within the DTD all semantic relationships between fields that a particular application may require, but some are certainly useful (e.g., to encode HATF's variable length 'attributes' field).

A general solution to the specific problem of encoding type names discussed in the previous section is to write these details into a table and interpret per record fields as an index into that table. To this end, Meta-TF provides the notion of sections. Each section in the DTD specifies those records that may appear in the corresponding section of the trace, and a trace may contain records that announce the start of a new section. It turns out that sections provide powerful mechanisms for compacting traces, of which this is just one example. Another is support for independent compression of field streams.

We reject existing abstract syntax notations such as SGML [6] or ASN.1 [7] because of their verbosity. Although tags are necessary to distinguish different record types, no further tags are desirable in a trace: as the DTD defines precisely the structure of each record, neither field tags nor closing record tags are necessary. Thus our DTDs impose no cost on the size of the trace. As with HATF 1.0, Meta-TF provides binary metadata records that modify the format of fields of subsequent records. However, because the DTD *documents* the format of the trace, it should be human readable, and allow meaningful names for records, fields and attributes. A further advantage of this approach is that it simplifies the automatic generation of trace reader/writer implementations from the DTD[5]. A full EBNF definition for trace files and Meta-TF DTDs is given in Appendix B; an example, encoding HATF 1.0 in Meta-TF, is shown in Appendix C.

4.3.1 The trace file

A trace file consists of a *header*, followed by a list of data, metadata, section and comment *records*, freely interspersed; its format is binary. The header specifies the DTD for the trace. The remainder of the trace file consists of a list of records, identified by their *tag*. A record comprises one or more *fields*, each of which has an *interpretation*. A record may contain *application data* (such as an allocation event), identify the start of a new *section* of the trace, be *metadata* that redefines the format of subsequent fields in that section or be a *comment*. *Section* records divide the trace into contiguous sections. Any global data and definitions are placed before the first section.

[5]It would also be possible to construct a universal reader/writer that used a DTD to guide its interpretation of traces associated with that DTD.

4.3.2 The DTD

The DTD specifies the sections and record types that may appear in a valid trace. The scope of a record type definition may be global or local to the section within which the record is defined. In the latter case, this definition applies only to records in the corresponding section of the trace.

The DTD defines, for each record, the fields that that record contains, the interpretations of those fields and, possibly, their value. Records may be nested. A record may define a *tag* value which must be unique within the section defining that record, and must be set in the record's definition. Meta-TF supports a richer syntax than HATF 1.0. For example, as well as composite fields using the `(field,field, ... ,field)` syntax, it provides repeating sequences of fields, using the `field*field` syntax where the value found in the first field indicates the number of occurrences of the second field that follows.

Properties may be defined for each field of a record. Definition of a field places an obligation to define, for that field, a *width* property and an *interpretation* property. Other properties of field may also be defined, e.g., a value (this is mandatory for tag fields). In addition to the interpretations of HATF, Meta-TF provides `sectionOffset` and `sectionStride` interpretations. The value of a field may also be used in setting properties. For example, a record type may define the value of its tag, such as `tag.value=4;` Properties may also refer to other fields in the record. In this way, certain semantics of a record type can be defined (see, for example, the pre-defined variable length field, `vfield`, in Appendix B.)

We noted in Section 3 that Split Stream gave better compression because the compressor is able to exploit the uniformity of certain field streams (such as addresses). Meta-TF offers the opportunity to achieve the compression of HATF SplitAddr by placing independently compressed streams in separate sections. For this purpose, Meta-TF provides two further interpretations. "`sectionOffset` *sectionNumber*" indicates that the per record value is an offset into the given section. "`sectionStride` *sectionNumber stride*" indicates that the per-record value should be taken from successive elements of a section. On each occasion a field of this type is encountered, the offset into that section is incremented by the stride; thus if the stride is 1, the first item is taken from offset 0, the next from offset 1 and so on. In order to ease interleaved access to different sections, the trace record header provides an index of the sections of the trace (*cf.* Unix ELF format [15]).

Four record types are built-in: `vfield`, `comment`, `metadata` and `section`. `Section` records allow the trace to be divided into sections. `Metadata` specifies the record in the current section to be changed (identified by its tag), the field that is to have its width or interpretation changed (identified by its position within the record), the property interpretation to be set and the value of the new setting. For example, the width of the size field of an `alloc` record may be changed by the metadata record[6] `(metadata, alloc, size, width, 8)`.

[6]For clarity, we use symbolic names rather than the octet values that would be used in practice.

4.4 Discussion of Meta-TF

We are interested in exploring the design decisions of Meta-TF further. Sections were motivated by the desire to provide string tables, but also support split stream compression. Sections of constants in the DTD may guide higher performance specialized readers. If we allow sections to be interleaved rather than contiguous, switching sections may provide shorter and more convenient compaction than metadata records. Other questions arise. Should metadata be used to define new records on-the-fly? Should metadata change values of fields? One use might be to vary tag widths and hence values. Would it be useful to allow field properties to refer to other records (i.e., `record.field.property`)? What this would mean? Presumably it would refer to the last such record encountered, but this would allow invalid yet type-correct traces if no such record had been encountered. Unfortunately, metadata records are, in some sense, magic: the interpretations of its fields are special. While this lack of uniformity appears to be inevitable, it is nevertheless somewhat unsatisfactory.

5. RELATED WORK

This paper is about defining a format for allocation traces. There has been substantial work on defining formats for other, related, domains, and we mention the most closely related here. As mentioned, Johnstone collected allocation traces for his thesis work [11]. Seidl also mentions collecting trace information about object allocations, but does not provide details of the format [20]. Humphries *et al.* define the POSSE Trace Format (PTF), which they use to capture the allocation and reference behavior of persistent object systems [5]. While there are similarities in some of the events defined, the focus of PTF was not on providing a compact encoding. Scheuerl *et al.* developed the MaStA I/O trace format for studying the I/O costs of various database implementations [19] but their application domain differs significantly from ours and they make no effort to focus on trace compactness.

We touch on some of the large body of work in the general problem of compression. Previous work in the area of address trace compression includes mache [18] and PDATS [10, 9]. While this work achieves admirable compression of address traces, the authors do not consider more complex structured traces such as allocation traces. The use of mobile code on the Internet has prompted recent interest in reducing code size through compression, e.g. Ernst *et al.* [3], Pugh [17], and Fraser [4]. Code and trace data are very different. No one has yet considered the effectiveness of code compression techniques on trace data.

Abstract syntax notations to describe data are commonplace, and a number of techniques and standards have evolved. The best known are ASN.1 [7] and members of the SGML family [6]. ASN.1 is a formal language for abstractly describing messages to be exchanged between distributed computer systems. Although very powerful, its syntax is verbose and complex. Its Basic Encoding Rules leads to obese output as every field carries type and length information as well as a value: for example, a sequence of 64 boolean values encodes to 196 octets [13]! Packed Encoding Rules (PER) [8] is more compact as it omits type and length information wherever possible and can pack octets using its `UNALIGNED` encoding.

For the same example, PER requires only 9 octets. Even so, PER does not support metadata, interpretations or sections.

SGML is a markup language for structured document and data representation. Its notion of DTDs captures precisely the separation of document structure from the instantiation of a particular document we require. Furthermore, its notion of embedded 'processing instructions' may be useful, (e.g., to extract compressed data). However, it is verbose (although there are ways to compact SGML: single character tags can be defined with 'short-refs' and closing tags can be omitted, for example) and it provides no way of expressing semantics, such as "the value of this field gives the size of the next field" (*cf.* `vfield` above). Indeed, such an embedding of semantics into a DTD is foreign to the whole philosophy of SGML.

An elegant alternative is to consider a document as an *expression*, replacing tags by function calls (say, in a functional language), and non-tag content by arguments, which may in turn be sub-expressions, (see, for example, [16]). DTDs are replaced by a *preludes* defining each tag-function. By evaluating an expression using different preludes, a document can be displayed or analysed in different ways. Although this technique would permit semantic relationships between fields (arguments) of records (sub-expressions) to be declared, its verbosity is even worse than that of SGML.

6. SUMMARY

This paper is about the practical problem of defining the contents and representation of domain-specific trace data. Our interest in this area is motivated by a desire to collect, analyse and distribute substantial allocation traces that can be used by us and others to do research. In the context of this concrete task, this paper suggests answers to the following questions. What data should allocation traces contain? How is this data most effectively represented? How should the content and representation of traces be specified?

We have defined HATF 1.0 and implemented a prototype reader/writer to investigate its compression and performance. Our format allows traces to associate allocation operations with specific heaps and threads, allowing traces from multithreaded server programs to be collected and analysed. HATF 1.0 allows, but does not require, metadata records that change the field encodings of data to be intermingled with the data records. Our initial experiments with placing metadata records indicate that such encodings result in better trace compression even after a general-purpose compressor such as gzip is applied. We also find that splitting out the address stream in the trace, and compressing it independently, results in the highest compression, requiring on average 1.54 bytes per allocation data record.

Because we understand the inherent limitations of any fixed trace format, we also propose Meta-TF, a language for specifying a family of trace formats, including HATF. Meta-TF provides a simple yet powerful method for describing trace formats. These formats are as compact as those of ad hoc methods, and readers/writers can be generated automatically from the Meta-TF DTD. Its notion of sections and metadata suggest further opportunities for compaction which we hope others will explore.

We see this paper as the starting point in a discussion about how to specify, represent, and use large domain-specific traces. In the future, we will continue to use and gain experience with our initial HATF implementation, at the same time working toward a Meta-TF generated implementation of HATF. Ideally, our goal is a format in which issues of representation and content are entirely dissociated. A trace consumer should be able to extract and manipulate the contents of a trace without caring whether the format was ASCII or binary, compressed or uncompressed, etc. Another important goal is to avoid a representation that has too many external dependencies. For example, a delivery format that does not require a user to have a specialized compression program installed is preferrable to one that does. One attractive aspect of an ASCII format is that Perl, which is nearly ubiquitous, can be used to programmatically manipulate traces easily. Our current efforts thus far have not focused on issues of external dependencies, but we plan to consider these more carefully in the future.

7. REFERENCES

[1] Emery D. Berger and Robert D. Blumofe. Hoard: A fast, scalable, and memory-efficient allocator for shared-memory multiprocessors. Technical Report CS-TR-99-22, University of Texas, Austin, September 1, 1999.

[2] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. *Software—Practice and Experience*, 24(6):527–542, June 1994.

[3] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. *ACM SIGPLAN Notices*, 32(5):358–365, May 1997.

[4] Christopher W. Fraser. Automatic inference of models for statistical code compression. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 242–246, 1999.

[5] Thorna O. Humphries, Artur W. Klauser, Alexander L. Wolf, and Benjamin G. Zorn. POSSE trace format version 1.0. Technical Report CU-CS-897-00, Department of Computer Science, University of Colorado, Boulder, CO, January 2000.

[6] International Standards Organization. *ISO 8879: Standard Generalized Markup Language*, 1986.

[7] International Standards Organization. *ISO 8824-4: Abstract Syntax Notation One (ASN.1)*, 1998.

[8] International Standards Organization. *ISO 8825-2: ASN.1 Encoding Rules: Specification of Packed Encoding Rules (PER)*, 1998.

[9] Eric E. Johnson. PDATS II: Improved compression of address traces. In *1999 IEEE International Performance, Computing, and Communications Conference*, February 1999.

[10] Eric E. Johnson and Jiheng Ha. PDATS: Lossless address trace compression for reducing file size and access time. In *Proceedings of the 1994 IEEE International Phoenix Conference on Computers and Communication*, April 1994.

[11] Mark S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, The University of Texas at Austin, Austin, Texas, 1997.

[12] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *Proceedings of the International Symposium on Memory Management (ISMM-98)*, volume 34, 3 of *ACM SIGPLAN Notices*, pages 26–36, New York, October 17–19 1999. ACM Press.

[13] John Larmouth. *Understanding OSI*. International Thompson Computer Press, 1996.

[14] Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In *Proceedings of the International Symposium on Memory Management (ISMM-98)*, volume 34, 3 of *ACM SIGPLAN Notices*, pages 176–185, New York, October 17–19 1999. ACM Press.

[15] John R. Levine. *Linkers and Loaders*. Morgan Kaufman, 2000.

[16] Kurt Nørmack. Programming World Wide Web pages in Scheme. *ACM SIGPLAN Notices*, 34(12):37–46, December 1999.

[17] William Pugh. Compressing Java class files. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 247–258, 1999.

[18] Alan Dain Samples. Mache: No-loss trace compaction. Technical Report CSD-88-446, University of California, Berkeley, September 15, 1988.

[19] S.J.G. Scheuerl, R.C.H. Connor, R. Morrison, J.E.B. Moss, and D.S. Munro. The MaStA I/O trace format. Technical Report CS/95/4, School of Mathematical and Computational Sciences, University of St. Andrews, North Haugh, St Andrews, Fife, Scotland, 1995.

[20] Matthew L. Seidl and Benjamin G. Zorn. Segragating heap objects by reference behaviour and lifetime. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 12–23, 1998.

[21] K. Phong Vo. Vmalloc: A general and efficient memory allocator. *Software Practice & Experience*, 1996.

[22] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.

[23] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs. *SIGPLAN Notices*, 27(12):71–80, December 1992.

APPENDIX

A. HATF 1.0 SPECIFICATION

- Data in a trace comes in two varieties: metadata and data.
- The metadata indicates the size and interpretation of each data field in the data that follows.
- Metadata and data can be freely interleaved.

EBNF

```
<trace> ::= {<metadata>* <data>*}*

<metadata> ::=
   set fieldSize           <field> <size>
   set fieldInterpretation <field> <interpretation>

<field>        ::= size | address | time | thread | heap | attributes
<size>         ::= 0 | 1 | 2 | 4 | 8 | v1 | v2

<interpretation> ::= none | default <default> |
                     baseOffset <base> | delta <initialValue> |
                     stride <initialValue> <stride>

<base>         ::= <hatf data value>
<default>      ::= <hatf data value>
<initialValue> ::= <hatf data value>
<stride>       ::= <hatf data value>

<data> ::= alloc | free | realloc |
           createHeap | destroyHeap |
           createThread | destroyThread | comment
```

The `<metadata>` and `<data>` binary formats are described below. A `<hatf data value>` is 8 bytes of unsigned integer data.

Notes

- Arbitrary annotations can be added to traces using two mechanisms. The comment data record can appear anywhere. In addition, all data records have a variable length field that can contain arbitrary additional data.
- fieldSize
  - *size* indicates the number of bytes that the field will occupy in the binary encoding.
  - The *v1* and *v2* choices for fieldSize are legal only for attribute fields. v1 and v2 indicate variable-sized attributes (v1 indicates 1 byte of length info, v2 indicates 2 bytes of length info). Examples of possible attribute data include immediate caller and the current callstack.
- fieldInterpretation
  In each case, *metadata record* refers to the metadata record in which the fieldInterpretation was set.
  - With the "none" interpretation, the value is the value stored directly in the record field.
  - With the "default" interpretation, the value is the *default* associated with the metadata record. The record field has zero width.
  - With the "baseOffset" interpretation, the value is the *base* associated with the metadata record plus the signed value stored in the record field.
  - With the "delta" interpretation, the value is the value of the same field in the most recent previous record plus the signed value stored in the record field. The first record that occurs after the metadata record uses the *initialValue* associated with the metadata record as the previous value of the field.
  - With the "stride" interpretation, the value is the value of the same field in the most recent previous record plus the signed *stride* value associated with the metadata record. The first record that occurs after the metadata record uses the *initialValue* associated with the metadata record as the previous value of the field.
  - Changing the fieldInterpretation automatically changes the fieldSize in the following way. The "stride" and "default" interpretations always change the fieldSize to zero. The "none", "baseOffset" and "delta" interpretations automatically change the fieldSize to the last non-zero width.

Defaults

Here is the standard default configuration. If no metadata is provided, these values are assumed.

```
set fieldSize size       4   set fieldInterpretation size       none
set fieldSize address    4   set fieldInterpretation address    none
set fieldSize time       0   set fieldInterpretation time       default 0
set fieldSize thread     0   set fieldInterpretation thread     default 0
set fieldSize heap       0   set fieldInterpretation heap       default 0
set fieldSize attributes 0   set fieldInterpretation attributes default 0
```

This interpretation provides allocation traces with characteristics very similar to those used in previous studies, such as Johnstone [11].

Metadata binary formats

byte 0	byte 1	byte 2	byte 3
1	1		
tag	op	a1	a2

where op = set fieldSize,
a1 = field , a2 = fieldWidth

byte 0	byte 1	byte 2	byte 3
1	2		
tag	op	a1	a2

where op = set fieldInterpretation,
a1 = field , a2 = interpretation “none”

byte 0	byte 1	byte 2	byte 3	bytes 4–11
1	2			
tag	op	a1	a2	a3

where op = set fieldInterpretation, a1 = field
a2 = delta, baseoffset, default, a3 = arg

byte 0	byte 1	byte 2	byte 3	bytes 4–11	bytes 12–19
1	2				
tag	op	a1	a2	a3	a4

where op = set fieldInterpretation
a1 = field, a2 = stride, a3,a4 = arg

Data binary formats

alloc

0						···
tag	size	address	thread	heap	time	attributes

free

1					···
tag	address	thread	heap	time	attributes

realloc

2-5							···
tag	size	old addr	new addr	thread	heap	time	attributes

createHeap / destroyHeap

6/7				···
tag	heap	thread	time	attributes

createThread / destroyThread

8/9			···
tag	thread	time	attributes

comment

byte 0	bytes 2-3	
10		···
tag	length	string

2-byte data

byte 0	byte 1
8-15	0-7
bits	bits

attributes (width = v1)

byte 0	...
v1	string

4-byte data

byte 0	byte 1	byte 2	byte 3
24-31	16-23	8-15	0-7
bits	bits	bits	bits

attributes (width = v2)

bytes 0-1	...
v2	string

8-byte data

byte 0		byte 7
56-63	...	0-7
bits		bits

Tag and field encodings

```
tag:                      metadata ops:                metadata <fieldWidth>
0 - alloc                 1 - set fieldSize            0 - 0     8 - 8
1 - free                  2 - set fieldInterpretation  1 - 1     9 - v1
2 - realloc no alloc                                   2 - 2     10 - v2
3 - realloc alloc free                                 4 - 4
4 - realloc alloc         metadata a2 = <field>
5 - realloc free          0 - size                     metadata <interpretation>
6 - createHeap            1 - address                  0 - none
7 - destroyHeap           2 - time                     1 - default
8 - createThread          3 - thread                   2 - baseOffset
9 - destroyThread         4 - heap                     3 - delta
10 - comment              5 - attribute                4 - stride
11 - metadata
```

B. META-TF 1.0 SYNTAX

The trace file

A trace file consists of a header, followed by a list of records defined by the DTD. The header must include information such as address size, byte order, the URL of the DTD as well as a section header that provides sufficient information for a reader to discover the location of each section in the trace.

```
typedef struct {          // section header
    int sh_number;        // section number
    int sh_start;         // start of section (byte)
    int sh_size;          // size of section (bytes)
} sectionhdr;

char magic[4] = "\177MTF";
char class;               // address size, 1 = 32-bit, 2 = 64-bit
char byteorder;           // 1 = little-endian, 2 = big-endian
char dtdlength;           // length of DTD name
char pad [9];
int nsections;            // number of sections
int start;                // start of data events (byte)
int nrecords;             // number of data events
char dtd[];               // URL of DTD
sectionhdr sections[];    // section headers
```

The DTD

The DTD defines the sections and record types that may appear within a valid trace data file.

```
    <docTypeDef> ::= <defs> { <section>* <comment>* }*
    <defs> ::= { <record>* <property>* <comment>* }*
```

Comments follow the C++/Java style: either multi-line comments delimited by `/*` and `*/` or single line comments introduced by `//`.

The scope of a record type definition may be global or limited to the named section within which the record is defined. In the latter case, this definition applies only to records with this tag in this section of the data file. Other than the implied global section, each section shall have a unique *name* and a unique *ID* (a small integer in the range 1–255). The section name is redundant but aids documentation of the DTD. `Section` is a reserved name. All names are statically scoped and are case-sensitive.

```
<section> ::= "section"  <sectionName>  <sectionNumber>
                  "{" <defs> "}"
```

The DTD defines, for each record, the fields that that record contains. Pronounce ':' as 'of type'.

```
<record> ::= <recordName> ":" <field>
                [ "{" <property>* "}" ]
```

The field name `tag` is reserved. The width of this tag field can only be set once, globally. Its interpretation (see below) is always `none`.

```
<field> ::= <simpleField> |
            <record> |
            <fieldName> "*" <simpleField>
<simpleField> :: = <fieldName> |
                  "(" <field> { "," <field> }* ")"
```

The `fieldName*simpleField` syntax is used to handle a sequence of fields. The value found in the `fieldName` field of each record indicates the number of occurrences of `simpleField` that follow. The `(field,field, ... ,field)` is used to denote composite fields (possibly in conjunction with the repeating field syntax above).

```
<property> ::= <fieldName> "." <propertyName> "=" <value> ";"
```

Properties may be defined for each field of a record. Property names are local to each record definition. Definition of a field places an obligation to define, for that field, a *width* property and an *interpretation* property. Other properties of field may also be defined, for example a value (this is mandatory for tag fields). Any non-negative integer is a valid width value. Standard property names (which are reserved) are: `width | interpretation | value`.

Standard field interpretations follow those of HATF 1.0 but also include `sectionStride <sectionNumber> <stride>` to support split stream encoding. Data for `sectionStride` fields are acquired by striding through the section specified. The widths of fields with interpretations `default`, `stride` and `sectionStride` are zero.

```
none | baseOffset <base> | delta <initialValue> | default <value> | sectionOffset <sectionNumber> |
stride <initialValue> <stride> | sectionStride <sectionNumber> <stride>
```

Four record types are predefined; their names are reserved. `vfield` is simply provided as an abbreviation for a common set of field and property definitions that implement a variable-width field.

```
vfield : (length,data) {
    length.width = 1;
    length.interpretation = none;
    data.width = length.value;
    data.interpretation = none;
}

comment : vfield {
    tag.value = 1;
}

section : (tag, number) {
    tag.value = 2;
}
```

```
    metadata : (tag, record, field, property, value) {
        tag.value = 3; // record.value = the tag of the record to be changed
        record.width = tag.width;
        record.interpretation = none; // field.value = field of the record to be changed (0..)
        field.width = 1;
        field.interpretation = none;
        /* property.value = the property of the field to be changed
           property.interpretation is special
           value.interpretation depends on the propertyName
           ---------------------------------------------------------------
           property.value meaning                      value
           ---------------------------------------------------------------
           0              width                        width
           1              interpretation=none          -
           2              interpretation=baseOffset    base
           3              interpretation=delta         initialValue
           4              interpretation=default       value
           5              interpretation=sectionOffset sectionNumber
           6              interpretation=stride        initialValue, stride
           7              interpretation=sectionStride sectionNumber, stride
           ---------------------------------------------------------------
         */
        property.width = 1;
        value.width = 4;
    }
```

The length of a metadata record is encoded in the `property` field. If the defined interpretation is `none`, then no value is given; if the interpretation is `stride`/`sectionStride`, then both an initial value/section number and a stride is given; otherwise, a single value is given.

C. AN EXAMPLE

HATF 1.0, introduced in Section 2, can now be defined with Meta-TF.

```
// HATF10.dtd: A DTD for HATF 1.0
// Meta-TF version 1.0

tag.width=1;
size.width = 4;       size.interpretation = none;
address.width = 4;    address.interpretation = none;
attributes.width = 0; attributes.interpretation = none;
time.interpretation   = default 0;  // hence, width=0
thread.interpretation = default 0;
heap.interpretation   = default 0;

section heap 1 {
    alloc : (tag, size, address, time, thread, heap, vfield) {
        tag.value = 4;
    }
    realloc : (tag, address, address, time, thread, heap, vfield) {
        tag.value = 5;
    }
    free : (tag, address, time, thread, heap, vfield) {
        tag.value = 6;
    }
    createHeap : (tag, thread, heap, vfield) {
        tag.value = 7;
    }
    destroyHeap : (tag, thread, heap, vfield) {
        tag.value = 8;
    }
    createThread : (tag, thread, vfield) {
        tag.value = 9;
    }
    destroyThread : (tag, thread, vfield) {
        tag.value = 10;
    }
}
```