CACHE-BASED TRACE RECORDING USING CACHE COHERENCE PROTOCOL DATA

CROSS-REFERENCE TO RELATED APPLICATION

This application claims priority to, and the benefit of U.S. Provisional Patent Application No. 62/559,780, filed Sep. 18, 2017, and titled "CACHE-BASED TRACE RECORDING USING CACHE COHERENCE PROTOCOL DATA", the entire contents of which are incorporated by reference herein in their entirety.

BACKGROUND

[0001] When writing code during the development of software applications, developers commonly spend a significant amount of time "debugging" the code to find runtime and other source code errors. In doing so, developers may take several approaches to reproduce and localize a source code bug, such as observing the behavior of a program based on different inputs, inserting debugging code (e.g., to print variable values, to track branches of execution, etc.), temporarily removing code portions, etc. Tracking down runtime errors to pinpoint code bugs can occupy a significant portion of application development time.

[0002] Many types of debugging applications ("debuggers") have been developed in order to assist developers with the code debugging process. These tools offer developers the ability to trace, visualize, and alter the execution of computer code. For example, debuggers may visualize the execution of code instructions, may present code variable values at various times during code execution, may enable developers to alter code execution paths, and/or may enable developers to set "breakpoints" and/or "watchpoints" on code elements of interest (which, when reached during execution, causes execution of the code to be suspended), among other things.

[0003] An emerging form of debugging applications enable "time travel," "reverse," or "historic" debugging. With "time travel" debugging, execution of a program (e.g., executable entities such as threads) is recorded/traced by a trace application into one or more trace files. These trace file(s) can then be used to replay execution of the program later, for both forward and backward analysis. For example, "time travel" debuggers can enable a developer to set forward breakpoints/watchpoints (like conventional debuggers) as well as reverse breakpoints/watchpoints.

BRIEF SUMMARY

[0004] Embodiments herein enhance "time travel" debugging recordings, by utilizing a processor's shared cache, along with its cache coherence protocol (CCP), in order to determine what data should be logged to a trace file. Doing so can reduce trace file size by several orders of magnitude when compared to prior approaches, thereby significantly reducing the overhead of trace recording.

[0005] Some embodiments, are implemented in computing environments that include (i) a plurality of processing units, and (ii) a cache memory comprising a plurality of cache

coherence per cache line, as well as an index (e.g., 0, 1, 2, 3 for a four-processing unit processor) to the processing unit that has the current cache line state.

[0055] Embodiments utilize the processor's shared cache 102b to efficiently record a bit-accurate trace of execution of an application 104c and/or the operating system kernel 104b. These embodiments are built upon an observation that the processor 102 (including the shared cache 102b) form a semi- or quasi-closed system. For example, once portions of data for a process (i.e., code data and runtime application data) is loaded into the shared cache 102b, the processor 102 can run by itself— without any input— as a semi- or quasi-closed system for bursts of time. In particular, one or more of the processing units 102a execute instructions from the code portion of the shared cache 102b, using runtime data stored in the data portions of the shared cache 102b and using the registers 102d.

[0056] When a processing unit 102a needs some influx of information (e.g., because an instruction it is executing, will execute, or may execute accesses code or runtime data not already in the shared cache 102b), a "cache miss" occurs and that information is brought into the shard cache 102b from the system memory 103. For example, if a data cache miss occurs when an executed instruction performs a memory operation at a memory address within the application runtime data 103b, data from that memory address is brought into one of the cache lines of the data portion of the shared cache 102b. Similarly, if a code cache miss occurs when an instruction performs a memory operation at a memory address application code 103a stored in system memory 103, code from that memory address is brought into one of the cache lines of the code portion of the shared cache 102b. The processing unit 102a then continues execution using the new information in the shared cache 102b until new information is again brought into the shared cache 102b (e.g., due to another cache miss or an un-cached read).

[0057] The inventor has observed that, in order to record a bit-accurate representation of execution of an application, the tracer 104a can record sufficient data to be able to reproduce the influx of information into the shared cache 102b during execution of that application's thread(s). A first approach to doing this is to record all of the data brought into the shared cache 102b by logging all cache misses and un-cached reads (i.e., reads from hardware components and un-cacheable memory), along with a time during execution at which each piece of data was brought into the shared cache 102b (e.g., using a count of instructions executed or some other counter). This approach is described in Applicant’s co-pending application U.S. Ser. No. 15/349,555, filed Nov. 11, 2016 and titled “CACHE-BASED TRACING FOR TIME TRAVEL DEBUGGING AND ANALYSIS”, the entire contents of which are incorporated by reference herein in their entirety.

[0058] A second approach— which results in significantly smaller trace files than the first approach— is to track and record the cache lines that were "consumed" by each

processing unit 102a. As used herein, a processing unit has "consumed" a cache line when it is aware of its present value. This could be because the processing unit is the one that wrote the present value of the cache line, or because the processing unit performed a read on the cache line. This second approach involves extensions to the shared cache 102b that enable the processor 102 to identify, for each cache line, one or more processing units 102a that consumed the cache line. This approach is described in Applicant's co-pending application U.S. Ser. No. 15/298,439, filed Oct. 20, 2016 and titled "FACILITATING RECORDING A TRACE FILE OF CODE EXECUTION USING A PROCESSOR CACHE" and U.S. Ser. No. 15/604,334, filed May 24, 2017 and titled "FACILITATING RECORDING A TRACE FILE OF CODE EXECUTION USING INDEX BITS IN A PROCESSOR CACHE", the entire contents of each of which are incorporated by reference herein in their entirety.

[0059] According to the embodiments herein, a third approach is to utilize the processor's CCP to determine a subset of "consumed" cache lines to record in the file(s) 104d, and which will still enable activity of the shared cache 102b to be reproduced. This third approach results in significantly smaller trace files— and thus significantly lower tracing overheads— than both of the first and second approaches.

[0060] Some embodiments herein record trace data streams that correspond to processing units / threads. For example, the trace file(s) 104 could include one or more separate trace data stream for each processing unit. In these embodiments, data packets in each trace data stream may lack identification of the processing unit the data packet applies to, since this information is inherent based on the trace data stream itself. In these embodiments, if the computer system 101 includes multiple processors 102 (i.e., within different processor sockets), the trace file(s) could have one or more different trace data streams for each processing unit 102a in the different processors 102. Plural data streams could even be used for a single thread. For example, some embodiments might associate one data stream with a processing unit used by a thread, and associate one or more additional data streams with each shared cache used by the thread.

[0061] In other embodiments, the trace file(s) 104 could include a single trace data stream for the processor 102, and could identify in each data packet which processing unit the data packet applies to. In these embodiments, if the computer system 101 includes multiple processors 102, the trace file(s) 104 could include a separate trace data stream for each of the multiple processors 102. Regardless of the layout of the trace file(s), data packets for each processing unit 102a are generally recorded independent of other processing units, enabling different threads that executed at the different processing units 102a to be replayed independently. The trace files can, however, include some information— whether it be express or inherent—that provides a partial ordering among the different threads.

[0062] Figure 3 illustrates a flowchart of a method 300 for performing a cache-based trace recording using CCP data. Method 300 may include acts that are performed by the processor 102 as the tracer 104a traces the application 104c and/or the operating system

300 of Figure 3, in the context of Figures 1, 2, 4A, 4B, and 5. Figure 6A illustrates a first table 600a that shows read and write activity by four processing units 102a (i.e., P0-P3) on a single line in the shared cache 102b. Figure 6B illustrates a second table 600b that indicates one embodiment of tracked cache coherence state (e.g., as tracked using the processor's CCP) based on these reads and writes. Figure 6C illustrates a third table 600c that shows what might be stored in accounting bits of the shared cache 102b (as described in connection with Figures 4 A and 4B), if accounting bits are used at all. While only one type of accounting bits would typically be used (i.e., unit bits per line, index bits per line, or a flag bit per line), for completeness in description table 600c shows each of unit bits 603, index bits 604, and flag bit 605. Finally, Figure 6D illustrates a fourth table 600d that shows example types of log data 606 that could potentially be written to the trace files(s) 104d in connection with each operation.

[0083] For simplicity in description, table 600a depicts operations by only a single processing unit 102a at a time, but it will be appreciated that the principles described herein apply to situations in which there is concurrent activity (e.g., concurrent reads by two or more processing units of the same cache line). Additionally, examples described in connection with Figures 6A-6D assume that tracking is enabled for processing units P0-P2, and is disabled for processing unit P3. For example, as discussed above, this could be controlled a bit corresponding to each processing unit, such a plurality of bits of a control register.

[0084] Initially, for ease in description, this example will use simplified cache lines states that are derived from the cache line states (i.e., Modified, Owned, Exclusive, Shared, and Invalid) used in the CCPs discussed above (i.e., MSI, MESI, and MOESI). In this simplification, these states map to either a "read" state (i.e., the cache line has been read from) or a "write" state (i.e., the cache line has been written to). Table 1 below shows one example of these mappings. Note that these mapping are used as an example only, and are non-limiting. For example, there could exist CCPs and states other than the ones discussed herein, and one of ordinary skill in the art will recognize, in view of the disclosure herein, that similar mappings can be made with many different CCPs.

TABLE 1

Protocol State	**Mapped State**
Modified	Write
Owned	Read
Exclusive	Write

Shared	Read
Invalid	No mapping – cache line considered empty

[0085] Notably, embodiments could log CCP data at varying levels, depending on what data is available from the processor 102 and/or based on implementation choices. For example, CCP data could be logged based on "mapped" CCP states (such the ones shown in Table 1), based on actual CCP states (e.g., Modified, Owned, Exclusive, Shared, and/or Invalid) made visible by the processor 102, and/or even based on lower-level "raw" CCP data that may not typically be made visible by the processor 102.

[0086] Turning to Figures 6A-6D, table 600a includes a first column 601 showing an identifier (ID), which is used to specify a global order among the operations. Table 600a also includes four additional columns 602a-602d that each corresponds to one of the processing units. While, for simplicity, this example uses a global ID, it will be appreciated that in practice each processing unit would normally order operations using its own independent sets of identifiers. These IDs could comprise an instruction count (IC), or any other appropriate mechanism for specifying an ordering among operations, such as a "jump count" plus program counter. Note that the example uses memory in a way that is consistent with the MSI, MESI, and MOESI CCPs, but for simplicity it uses only the "modified," "shared," and "invalid" states. It is noted, however, that some CCPs could provide their own unique and/or monotonically-incrementing IDs that could also be recorded in a trace (e.g., in every packet, or in occasional packets) to strongly order trace entries. Even if the CCP doesn't provide such an ID, the value of a socket timer (e.g., TSC) or another orderable ID could potentially be used.

[0087] As shown in table 600a, at identifier ID[0] processing unit P0 performs a read, which causes a cache miss bringing data DATA[1] into the cache line. Correspondingly, table 600b shows that the processor's CCP notes that the cache line is now "shared" by P0. Table 600c shows that if unit bits 603 are used they indicate that processing unit P0 has consumed (i.e., read) the cache line (and that processing units P1-P3 have not), that if index bits 604 are used they indicate that P0 has consumed the cache line, and that if a flag bit 605 is used it indicates that some processing unit has consumed the cache line. Given this status, in act 303 the processor 102 would determine that logging is enabled for P0, and in act 304 it would determine that the cache line participates in logging (i.e., using unit bits 603, index bits 604, flag bit 605, or way-locking). Thus, in act 306 the processor 102 would utilize the

CCP to log appropriate data to the trace file(s), if necessary. Here, since the cache line is going from an invalid (empty) state to a read (table 600a)/shared (table 600b) state, data should be logged. As shown in the log data 606 of table 600d, the processor 102 could note the processing unit (P0) if necessary (i.e., depending on whether the data packets are being logged to separate data steams per processing unit, or to a single data stream); the cache line address (@); the instruction count or some other count; and the data (DATA[1]) that was brought into the cache line. While, as discussed above, the instruction count will typically be a processing unit-specific value, for simplicity table 600d refers to instruction counts in reference to the corresponding global ID (i.e., IC[0], in this instance).

[0088] It is noted that the cache line address (@) and the data (e.g., DATA[1]) could, in some embodiments, be compressed within the trace file(s) 104d. For example, memory addresses can be compressed by refraining from recording the "high" bits of a memory address by referencing (either expressly or implicitly) the "high" bits in a prior recorded memory address. Data can be compressed by grouping bits of a data value in to a plurality of groups comprising a plurality of bits each, and associating each group with a corresponding "flag" bit. If a group equals a particular pattern (e.g., all 0's, all l 's, etc.), the flag bit can be set, and that group of bits need not be stored in the trace. A further description of these techniques is described in Applicant’s co-pending application U.S. Ser. No. 15/349,474, filed Nov. 11, 2016 and titled “REDUCING STORAGE REQUIREMENTS FOR STORING MEMORY ADDRESSES AND VALUES”, the entire contents of which are incorporated by reference herein in their entirety.

[0089] Next, table 600a shows that at ID[1] processing unit PI performs a read on the cache line, reading data DATA[1]. Table 600b shows that the processor's CCP notes that the cache line is now "shared" by P0 and PI . Table 600c shows that that processing units P0 and PI have consumed the cache line (unit bits 603), that PI has consumed the cache line (index bits 604), or that some processing unit has consumed the cache line (flag bit 605). Note that it would also be correct for the index bits 604 to still reference P0 instead of PI . Table 600d shows that, using the CCP, the processor 102 determines that a record of the operation is to be logged. As shown, the processor 102 could note the processing unit (PI); the cache line address (@); the instruction count (IC[1]); that the cache line has gone from a read (shared) state to a read (shared) state; and that P0 had prior access to prior cache line, but now P0 and PI have access.

[0090] Next, table 600a shows that at ID[2] processing unit P0 performs a write to the cache line, writing data DATA[2]. Table 600b shows that the processor's CCP notes that the cache line is now "modified" by P0 and "invalid" for PI . Table 600c shows that only processing unit P0 has consumed (i.e., updated the value of) the cache line (unit bits 603), that P0 has consumed the cache line (index bits 604), or that some processing unit has consumed the cache line (flag bit 605). Table 600d shows that, using the CCP, the processor

information that helps quickly reduce the search space when separating trace data streams; and

(6) A non-full log (i.e., one without all of the transitions logged) can still be used to replay the trace. This can bring about extra computational cost at replay time to calculate the missing pieces.

[00116] In a first example, the CCP tracks cache line status per processing unit (i.e., each core has its own read and write status). In this example, the cache behaves much like an inclusive cache, except that there may be data that moves cross-cache or cross-socket that is not available at logging time. For brevity, in these examples the processing units 802 are referred to as "cores" and processors 801a and 801b are referred to a processors A and B or sockets A and B. Additionally, a simplified logging notation of "ID:Core:From:Transition(i.e., from -> to)" is used to represent types of data that could be logged. This notation is explained more detail inline. For the first example, logging could include:

[00117] At ID[0], "0:A0:R[DATA] -> [1]"— i.e., at ID[0], logging that core AO read DATA[1], per principle 1 above.

[00118] At ID[1], "1 :B0:R[DATA] -> [1]"— i.e., at ID[1], logging that core B0 read DATA[1], also per principle 1 above. If the cache in processor B is unaware that AO has the data already logged, then processor B logs it itself. Alternatively, if the cache in processor B is aware that AO has DATA[1] logged, then the log entry could include "1 :B0:R[A0]->R".

[00119] At ID[2], "2: Al :R[A0]->R"— i.e., at ID[2], logging that core Al did a read -> read transition, and that AO had access. Since the cache line state is shared with processor B, the entry could be "2:A1 :R:[A0,B0]->R"— i.e., at ID[2], logging that Al did a read -> read transition, and that AO and B0 had access. Since crossing sockets is typically more expensive than logging within a socket, the first log entry may be preferred for read -> read transitions. When logging to/from writes that cross sockets, however, logging also crosses sockets.

[00120] At ID[3], some embodiments log nothing. Alternatively, since the A2 core hasn't logged anything yet, and the first thing it does is a write, this could be logged as read -> write. Either way, since a write occurred, the other cores have their cache line state invalidated. The cost (e.g., in terms of trace data) of logging the read -> write at ID[3] would typically be less than logging actual data at ID[4], so it may be beneficial to log here. In this case, the log entry could include "3 :A2:R[A0,B1,B0] ->W"— i.e., core A2 did a read ->

then AO is on the source list of cores (e.g., "11 :A2:R[A0,A1]->R); otherwise, AO is not in the list (e.g., "11 :A2:R[A1]->R").

[00129] At ID[12] if socket B can see socket A, this is read -> read packet (e.g., "12:B0:R[A0,A1,A2]->R"). If it can't then it is a full data log (e.g., "12:B0:R[DATA]->[4]").

[00130] At ID[13] the data comes from B0, plus socket A if it is visible (e.g., "13 :B 1 :R[A0,A1,A2,B0]->R"). The list may omit core AO if the write was not logged at ID[8].

[00131] At ID[14] nothing needs to be logged if a packet was logged at ID[8] did already log. Otherwise AO will get the data from Al & A2, plus potentially the socket B if it can be seen. As such, the packet could include "14:A0:R[A1,A2,B0,B1]->R".

[00132] Note that while this example logged the sockets together, it would be correct to log each socket in isolation, similar to the way threads are can be logged in isolation. This might result in larger traces, but it has the advantage of not having to change any cross-socket communication mechanism in the processor.

[00133] Also, at any moment in time the cache line may be evicted, which would mean that the data needs to be gathered from another core or re-logged. For example, if before ID[11], AO had its cache line evicted, then A2 would get the value from Al . If both Al and AO were evited, then processor A may need to log the cache line value into the trace for A2.

[00134] Finally, some processors may know that data comes from another socket, but not know which core in that socket. In those cases, the processor could log precedence (source) as a socket ID, log the data itself, or log the socket ID and a hash of the data (i.e., to help order cross socket accesses, but not have to log the entire data to the trace).

[00135] In a second example, the CCP uses indexes instead of tracking cache coherency of each core separately. In this environment, the index could be tracked cross socket or intra-socket. Due to the performance of cross socket versus intra-socket communications, the latter case (intra-socket) may be more practical. When the index is tracked intra-socket, the trace may need to log something when data moves cross socket. This could include logging the index from the other socket (but this may not necessarily be unique enough for a deterministic trace), logging a hash of one or more portions of the cache line value, or logging a packet on the sending socket's trace to indicate that data was sent.

[00136] When tracking core indexes when using a non-fully-inclusive cache, a complication arises when an LI cache may have data that is not in the L3 cache. So, for example assume the following sequence of events: (i) AO gets a line (thus the index bits

refer to AO) in its LI cache; (ii) Al gets the line (thus the index bits refer to Al) in its LI cache; (iii) L3 cache evicts the line; (iv) Al evicts the line from is LI cache; and (v) A2 gets the cache line from AO in its LI cache. Here, although A2 gets the cache line from AO, the index does not refer to AO. This complicates logging mappings into the trace. Some solutions could include adding extra information (as described above), such as a hash of one or more portions of the cache line data, periodically adding redundant information like a hash of the general-purpose registers, etc. Logging evictions could also help, but that may significantly grow the trace file size and complicate logging (e.g., logging LI cache evictions that are not in the L2 or L3 caches, but not logging LI cache evictions the that are in the L2 or L3 caches).

[00137] In some embodiments, when data moves from an L3 cache to a child L2 or LI cache, a log entry is only made if the index changes. For example, suppose that AO has the line in its LI cache (thus the index bits refer to AO), then Al gets the line in its LI cache (index at Al), then both evict the cache line but the common L2 (or L3) still has it. If the L2 cache serves Al, then there is nothing to log. If the L2 cache serves AO then no log entry needs to be made if it is known that AO already had the data; but if it is not known (or can't be determined) if AO already had the data, then the processor may need to log a read -> read.

[00138] Below is an example log of the operations of table 800b, assuming that sockets log independently, that tracking is performed by index, that there are no extra hidden evictions, and that all writes that impact the CCP and that happen when logging is turned on are logged (e.g., one write needs to be logged if there are consecutive writes by the same core and there is no access between the writes by another core or other external entity). For the second example, logging could include:

[00139] For ID[0], "0:A0:R[DATA]->[1]".

[00140] For ID[1], "1 :B0:R[DATA]->[1]"— i.e., recall that each socket is logged separately.

[00141] For ID[2], "2: Al :R[A0]->R".

[00142] For ID[3], "3 :A2:R[A1]->W".

[00143] For ID[4], nothing.

[00144] For ID[5], "5:B0:R[DATA]->[2]". This is because the write at ID[3] invalidated the line across all sockets, and sockets are being traced independently (as stated above).

[00145] For ID[6], "6:B1 :R[B0]->R".

[00146] For ID[7], if the cache line for B0 hasn't been evited there is nothing to log.

[00147] For ID[8]: "8:A0:R[A2]->W", since the logging bit is on (and despite this core

of trace file size to record data packets that reference data possessed by another processing unit (when possible), rather than recording cache line data later (e.g., ID[4] in each of the preceding examples). Other benefits can also flow from recording by reference. For example, at replay, when there are a series of log entries that are by reference, it can be inferred that no external intervention happened in the cache line data. This is because when a full cache line data is re-logged, it means that either the cache line was evicted or invalidated. Thus, including log entries by reference, even in situation when a log entry may strictly not be necessary, can provide implicit information about the absence of external interventions that may be useful information at replay or for debugging.

[00159] In some implementations, the addresses that are recorded in the trace entries (e.g., the "@" entries above) comprise physical memory addresses. In these implementations, the processor 102 may record one or more entries of the TLB 102f into the trace file(s) 104d. This may be as part of the trace data streams for the different processing units, or as part of one more additional trace data streams. This will enable replay software to map these physical addresses to virtual addresses later.

[00160] In addition, since physical addresses may at times be considered "secret" information (e.g., when recording at the level of user mode), some embodiments record some representation of the actual physical addresses, rather than the physical addresses themselves. This representation could be any representation that uniquely maps its identifiers to physical addresses, without revealing the physical address. One example could be a hash of each physical address. When these representations are used, and entries of the TLB 102f are recorded into the trace file(s) 104d, the processor 102 record a mapping between these representations and virtual addresses, rather than physical addresses to virtual addresses.

[00161] As mentioned, the processor 102 can include one or more buffers 102e. These buffers can be used as a temporary storage location for trace file entries, prior to actually writing those entries to the trace file(s) 102f. Thus, when act 305 causes data to be logged to the trace, act 305 could comprise writing the data to the buffer(s) 102e. In some embodiments, the processor 102 employs deferred logging techniques in order to reduce impact of writing trace data on the processor 102 and the memory bus. In these embodiments, the processor 102 may store trace data into to the buffer(s) 102e, and defer writing to the trace file(s) 102f until there is available bandwidth on the memory bus, or the buffer(s) 102e is/are full. Some of these techniques are described in Applicant’s co-pending application U.S. Ser. No. 15/688,828, filed Aug. 28, 2017 and titled “LOGGING TRACE DATA FOR PROGRAM CODE EXECUTION AT AN INSTRUCTION LEVEL,” the entire contents of which are incorporated by reference herein in their entirety.

[00162] As was also mentioned, some embodiments may log cache evictions. Figures

What is claimed

1. A computing device, comprising:

a plurality of processing units;

a cache memory comprising a plurality of cache lines that are used to cache data from one or more backing stores and that are shared by the plurality of processing units, wherein consistency between data in the plurality of cache lines and the one or more backing stores is managed according to a cache coherence protocol (CCP); and

stored control logic that configures the computing device to perform at least the following:

determine that at least the following conditions have been met:

(i) an operation has caused an interaction between a particular cache line of the plurality of cache lines and the one or more backing stores;

(ii) logging is enabled for a particular processing unit of the plurality of processing units that caused the operation;

(iii) the particular cache line is a participant in logging; and

(iv) the CCP indicates that there is data to be logged to a trace based on the operation; and

based at least on determining that the conditions having been met, cause the data to be logged to the trace, the data usable to replay the operation.

2. The computing device as recited in claim 1, wherein the stored control logic also configures the computing device to update one or more accounting bits associated with the particular cache line to indicate whether the particular cache line remains a participant in logging after the operation.

3. The computing device as recited in claim 2, wherein the one or more accounting bits associated with the particular cache line comprises one of (i) a single bit, (ii) a plurality of bits that each corresponds to one of the plurality of processing units, or (iii) a plurality of bits that store a processor index value.

4 The computing device as recited in claim 2, wherein the one or more accounting bits associated with the particular cache line are stored in one or more reserved cache lines that are separate from cache lines that are used to cache data from one or more backing stores.

5. The computing device as recited in claim 1, wherein causing the data to be logged to the trace comprises writing the data to a buffer, and wherein flushing data from the buffer to the trace file is deferred based on memory bus activity.

6. The computing device as recited in claim 1, wherein the stored control logic also configures the computing device to log at least one cache eviction by reference to a group and a way in an associative cache.

7. The computing device as recited in claim 1, wherein the data logged comprises transitions between different CCP states.

8. The computing device as recited in claim 1, wherein the data logged comprises at least one of: a transition from write state to a read state, a transition from a write state to a write state, or a transition from a read state to a write state.

9. The computing device as recited in claim 1, wherein using the CCP to identify that there is data to be logged to a trace comprises identifying that a transition from a read state to a read state need not be logged to the trace.

10. The computing device as recited in claim 1, wherein data for each processing unit is logged to at least one separate data stream.

11. The computing device as recited in claim 1, wherein data for two or more processing units is logged to the same data stream, but tagged with a processing unit identifier.

12. The computing device as recited in claim 1, wherein the data to be logged to the trace comprises ordering information.

13. The computing device as recited in claim 1, wherein the data to be logged comprises data written to the particular cache line by an enclave, and wherein causing the data to be logged to the trace comprises:

when the operation that caused the interaction between the particular cache line and the one or more backing stores corresponds to a thread interacting with the enclave, causing the data to be logged into a trace data stream corresponding to the thread; or

when the operation that caused the interaction between the particular cache line and the one or more backing stores corresponds to the enclave, causing the data to be logged to be separated from the trace data stream corresponding to the thread.

14. A method, implemented in a computing environment that includes a plurality of processing units and a cache memory comprising a plurality of cache lines that are used to cache data from one or more backing stores and that are shared by the plurality of processing units, wherein consistency between data in the plurality of cache lines and the

one or more backing stores is managed according to a cache coherence protocol, the method for performing a cache-based trace recording using cache coherence protocol (CCP) data, the method comprising:

determining that at least the following conditions have been met:

(i) an operation has caused an interaction between a particular cache line of the plurality of cache lines and the one or more backing stores;

(ii) logging is enabled for a particular processing unit of the plurality of processing units that caused the operation;

(iii) the particular cache line is a participant in logging; and

(iv) the CCP indicates that there is data to be logged to a trace based on the operation; and

based at least on determining that the conditions having been met, causing the data to be logged to the trace, the data usable to replay the operation.

15. The method of claim 14, further comprising updating one or more accounting bits associated with the particular cache line to indicate whether the particular cache line remains a participant in logging after the operation.

16. The method of claim 14, wherein causing the data to be logged to the trace comprises writing the data to a buffer, and wherein flushing data from the buffer to the trace file is deferred based on memory bus activity.

17. The method of claim 14, wherein the data logged comprises transitions between different CCP states.

18. The method of claim 14, wherein the data logged comprises at least one of a transition from write state to a read state, a transition from a write state to a write state, or a transition from a read state to a write state.

19. The method of claim 14, wherein using the CCP to identify whether data is to be logged to a trace comprises identifying that a transition from a read state to a read state need not be logged to the trace.

20. A computer program product for use at a computing device that comprises a plurality of processing units and a cache memory comprising a plurality of cache lines that are used to cache data from one or more backing stores and that are shared by the plurality of processing units, wherein consistency between data in the plurality of cache lines and the one or more backing stores is managed according to a cache coherence protocol (CCP), the computer program product comprising one or more hardware storage devices having stored thereon computer-executable instructions that are executable by one or more processing units to cause the computing device to perform at least the following:

determine that at least the following conditions have been met:

(i) an operation has caused an interaction between a particular cache line of the plurality of cache lines and the one or more backing stores;

(ii) logging is enabled for a particular processing unit of the plurality of processing units that caused the operation;

(iii) the particular cache line is a participant in logging; and

(iv) the CCP indicates that there is data to be logged to a trace based on the operation; and

based at least on determining that the conditions having been met, cause the data to be logged to the trace, the data usable to replay the operation.