2012 19th Asia-Pacific Software Engineering Conference

Software-based Online Monitoring of Cache Contents on Platforms without Coherence Fabric

Adriaan Schmidt Oliver Horst
Fraunhofer Institute for Communication Systems ESK
Munich, Germany
Email: {adriaan.schmidt, oliver.horst}@esk.fraunhofer.de

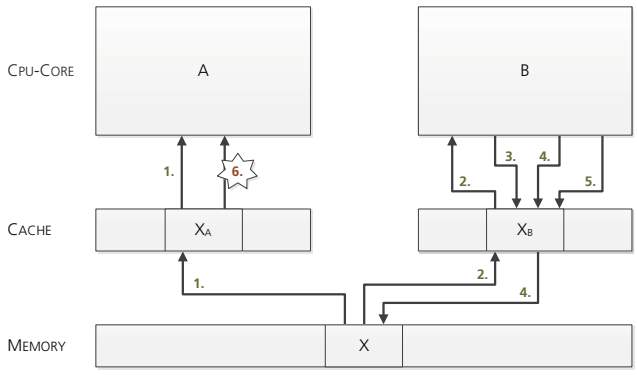
***Abstract*—In favor of smaller chip areas and associated fabrication costs, designers of embedded multi-core systems on occasion decide not to include cache coherence logic in the hardware design. However, handling all cache coherence exclusively in software is error-prone, and there are presently no tools supporting developers in this task. Thus, we propose a new software testing method, based on online inspection of the cache contents, to pinpoint programming mistakes related to cache handling. This concept helps localizing the causing data symbol even for complicated cache handling errors, e. g. where the causing and manifesting code-location of an error differ. Our solution is a pure software solution and does not require any specialized hardware. We evaluate our approach by using it in a large application, and show that we can detect typical cache-related errors.**

I. Introduction

The problem of maintaining cache coherence on multi-core platforms is well known and is solved by means of coherence fabrics implementing protocols like MESI [1] or MOESI [2]. However, there are multi-core systems that do not implement such hardware-based cache coherence. Especially in the embedded systems domain, hardware designers may choose to omit such coherence logic in favor of smaller chip areas and fabrication costs. Also heterogeneous platforms, as for example the Cell BE architecture [3], often do not maintain coherent caches. Moreover, there is an ongoing discussion whether current approaches of cache coherence will scale to future many-core architectures with hundreds or even thousands of cores [4].

The absence of hardware cache coherence poses challenges to the software developer. The software design has to ensure that concurrent access to shared objects in memory does not lead to wrong results. We will present two examples that illustrate the problems that can occur.

Example 1: Assume two cores cooperate in the processing of an algorithm and need to access a shared variable. Figure 1 shows an example access sequence to such a shared variable. On platforms providing cache coherence logic, one core's update of the shared variable is immediately reflected on the other core, e. g. through invalidation of the corresponding cache line. In systems without such logic, cores can have different values stored for the same memory location in their local caches, and the shared memory will only reflect changes to memory after a writeback operation. In the illustration (Figure 1), the value of `X` in the shared memory is only affected after the writeback operation in step 4. When reading a memory



				X_A	X_B	X
1.	A:	read x	= 3	3	–	3
2.	B:	read x	= 3	3	3	3
3.	B:	write x, 5		3	5	3
4.	B:	writeback x		3	5	5
5.	B:	invalidate x		3	–	5
6.	A:	read x	= 3	3	–	5

Figure 1. Illustration of a typical bug while using shared variables on platforms without hardware-based cache coherence logic.

location, the change made by another core becomes visible only after explicitly invalidating local cache contents causing a read from memory the next time the address is accessed. Thus, the subsequent read operation of core `A` in step 6 still returns the outdated value for `X` from the local cache as a consequence of the missing preceding invalidation of the local cache.

It is clear that without a coherence fabric implemented in hardware, some sort of coherence protocol must be implemented in software. We will approach this problem by defining that there can at most be one unique "owner" for each memory location. By "owner" we mean one thread of execution running on one core. This thread is exclusively allowed to access the memory location, i. e. only the core currently executing the owning thread, should have the location in its local cache. This notion of "ownership" is a convention that is not enforced by hardware logic; it is the software's responsibility to ensure that data is kept in a consistent state. Ownership of a memory location can change over time, and can be transferred from one application or core to another. If such a change of ownership

occurs, it has to be guaranteed that all cache lines involved are invalidated on the in the cache of the previous owner before memory is accessed by the new owner.

Example 2: In a software system, buffers may be used to transfer data between software components running on different cores. It is in this case essential to always invalidate all respective cache lines before a buffer is transferred to a new owner. Assume a software component A finishes using a buffer, and due to a programming mistake, the memory area in question is not completely invalidated in the local cache. If at a later time, this buffer is ever used to transfer new data to the same core, but to a different application B, instead of the new information, the outdated contents from the local cache will be used instead.

Programming mistakes like this are difficult to detect and reproduce. In the second example the effect of the incomplete invalidation might not always be visible, because by the time the buffer is re-used by application B, the respective cache lines may have been evicted from the local cache in the course of normal system operation. Even if a test case is created that reliably reproduces the issue, pinpointing the mistake can be difficult. In our example, a programming mistake made in application A causes a failure in B, while the two applications may be entirely unrelated.

Currently there are no tools supporting a developer to deal with the described problems. We therefore propose a method and tool for online checking of cache contents to detect cache coherence violations, the Cache Checker. With our Cache Checker, we target embedded systems, where memory is often allocated statically, and thus valid cache contents are known in advance, or are easily verifiable at run-time.

Note that cache coherence issues only affect the handling of private (mostly first level) caches, as this is the only situation in which cores can observe conflicting contents for the same memory location. There are potential problems also with shared caches. It is possible that these do not reflect actual memory contents, e. g. when memory is changed by devices using direct memory access. However, these effects lie out of the scope of this paper. Furthermore, we only examine data caches. The possibility of coherence problems related to instruction caches arises only in two situations: a) code is loaded dynamically at run-time, or b) code can be self-modifying. We assume that neither is the case.

The rest of the paper is structured as follows. In Section II we present the concept of our Cache Checker. Section III describes the details of a prototype implementation and gives an assessment of its performance. In Section IV we evaluate the use of the Cache Checker in the development process and present related and future work, before Section V concludes the paper.

II. Cache Checker Concept

The objective of our Cache Checker is to provide early hints to cache related problems that are otherwise hard to detect. This is achieved by online inspection of the cache contents. If an indication for a software defect is found, a warning message is generated, presenting details of the incident to the developer. The advantage of an online inspection is that it enables the Cache Checker to point out even those software defects that are not provoked by any regression or black-box test. Moreover, during an online analysis all data symbols involved in a defect are known, which can help pinpointing the error location in the source code.

However, to be able to perform the type of online checking of cache contents we propose, it is required that the internals of the core's caches are accessible by software. It must be possible to retrieve the corresponding memory addresses for any given cache line i. e. cache tag, along with information whether a cache line is currently valid or not. Unfortunately, not all hardware platforms offer access to this internal information.

The Cache Checker's main concept is the classification of cache contents, and thus their related memory accesses, into two categories: *allowed* and *not allowed*. To perform this classification, system specific rules need to be defined that specify, which data is accessed by which cores, and thus which memory addresses are expected to be found in the cache of the respective cores. There are cases, especially in embedded systems, when these rules describing the software's memory accesses can be defined statically, i. e. at compile time. An example is a software component that is scheduled to only run on a single core. All working data of the software component is exclusively accessed by this core, so if parts of the data are found in the cache of a different core, this indicates a software defect, and will be classified as *not allowed*. We will call this kind of access scheme *statically owned memory*.

In other instances, the decision is not as easy, and cannot be taken with information available at compile-time. This is for example the case, when an array of buffers is used to transfer data between different cores. Each single buffer will be accessed by several cores over time. Thus, the time is important at which an address related to a buffer was found in the private cache of a core. In these instances, when the ownership can only be decided at run-time, we speak of *dynamically owned memory*.

Ideally, we would want our checks to reflect the complete contents of all caches at all times. This would permit a complete analysis of all memory accesses and guarantee that all violations of the cache coherence can be detected. In practice however, this is not feasible. A complete analysis of all memory accesses would result in an unacceptably high computational overhead, which would in turn change the timing behavior of the target application. Thus, we have to limit our checks to certain points in time. These points in time can be carefully chosen by the developer, to reduce the influence of the Cache Checker on the target application. Concentrating the cache checks within certain intervals in time results into a program flow, where between two consecutive checks of the cache contents, there are phases of normal program execution, in which coherence violations can occur. Accordingly, the Cache Checker can only detect those violations that persist until the next check, i. e. concerning data that has not been evicted from cache by subsequent operations. Accordingly, the Cache Checker's capability to detect cache coherence bugs is limited by two

constraints: a) the completeness of the specified system specific rules and b) the epoch based execution scheme. It is therefore expected that the Cache Checker will miss some possible cache coherence violations. The usage scenarios of our tool however are long test runs. In this way, the longer the test runs, the more likely it becomes that the Cache Checker sees violations, even if they do not persist for a long time. Accordingly, we suggest the idle-loop to perform the cache checks, to reduce the influence on the observed system to a minimum.

To reliably decide if an access to a certain memory region is *allowed* or *not allowed*, however, requires detailed knowledge of the application. Thus, the configuration of the Cache Checker models the memory usage of the system, specifying which memory accesses, and thus which cache contents, are expected. To specify this memory usage model, the developer provides rules, which of the program's data symbols (e. g. variables) may be accessed by which core, and under which conditions. These rules form the configuration of the Cache Checker. One factor that influences the memory usage model is the selection of points in time at which the checks of cache contents are performed. It is expected that the complexity of the memory usage model varies, depending on the decision when to invoke the Cache Checker.

The actual analysis and classification of the observed memory accesses into the two categories *allowed* and *not allowed* is done by the so called *classification function*. This function compares the program symbols specified in the Cache Checker configuration rules against the internal cache information obtained from the hardware. However, this internal cache information contains physical addresses that cannot be compared directly to the program symbols from the configuration, as the linker translates the symbols into virtual addresses only. Thus, the classification function would first need to calculate the physical addresses for those virtual addresses, before it could start the comparison. To avoid this computational overhead at run-time, we pre-calculate these addresses offline. To obtain the addresses related to the defined symbols at compile-time, we analyze the compiled binary (.elf) image file of the target application. However, as the Cache Checker is part of this image, but contains code that requires information from the image, we need to adapt the compilation process of the target application: First we compile without the classification function and extract the required address information from the image, after which we re-compile the application, this time including the classification function. The classification function itself is automatically generated from the configuration files and the information from the binary image. This approach was chosen because of the following advantages: (i) The maintenance of the classification function is greatly simplified when the developer does not have to deal with the actual code himself. Corrections and adaptions can be made easily, and the developer can focus on what he wants classified instead of how classification is performed in detail. (ii) The actual classification happens based on physical memory addresses. This means that the compiler usually does not know the addresses of the symbols at compile-time. Classification code thus deals with raw addresses, which is tedious and error prone when done manually. In addition, the addresses can change when modifications are made to the target application. This means that all addresses in the classification function would need to be manually adapted. (iii) Code generation offers the possibility to automatically optimize the classification function. We will discuss this aspect in Section III.

In addition to whether an access is allowed or not, our classification also returns a numerical value. This *classification code* lets the developer know, to which memory region the classified address belongs.

When the Cache Checker detects violations of the configured rules, an alert should be raised. The information available at this point includes the physical memory address that was found in cache, the core on which the violating memory access was detected, and the result of the classification. The simplest option to handle alerts is to collect them in memory, from where they are accessible through a debugger. This does not require any means of communication like a UART or a network interface. If other means of communication between the target and the development host are available, a more comfortable way of alert handling is possible. As an example, we implemented the sending of alert messages and performance data of the Cache Checker in network packets that are received by a user interface on the development host. Here, some additional information is available through the linker information, like the virtual memory address of the data found in cache and the symbol that is associated with the address. This additional data is collected and presented to the developer.

The transmission of alerts by the Cache Checker has some influence on the run-time behavior of the target system. However, this is tolerable, since the Cache Checker will only generate alerts if possible software flaws are detected. This case should be the exception, which means that the overhead is limited. Also, once a violation is detected, we consider the further, time-accurate execution of the target software to be of lower priority.

III. Implementation and Evaluation

This section first introduces the target hardware and software platform, then continues with a discussion of the concept and implementation of the Cache Checker, and closes with a performance evaluation of the proposed Cache Checker implementations.

A. Target Platform

We implemented our Cache Checker on Lantiq's "Vinetic SVIP" platform [5], which features multiple MIPS32-cores [6]. The platform contains separate core-private first level caches for code and data, and a shared second level cache. The L1 data caches targeted by the Cache Checker are implemented as 4-way set associative caches. The on-chip interconnect has the form of a crossbar, linking the processor cores to the L2 cache, the memory controller and other on-chip hardware devices. The SVIP supports two kinds of memory: a fast on-chip SRAM, which is accessed via core-private L1 caches, and a larger

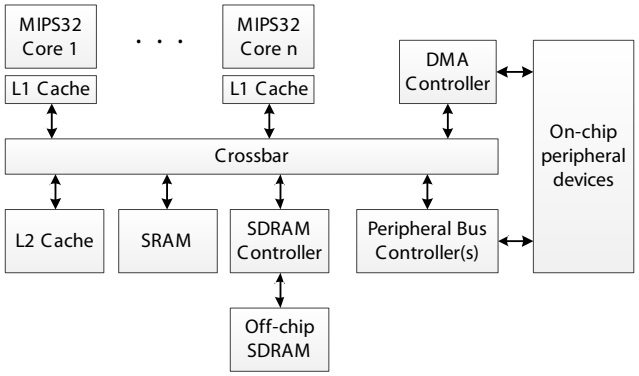


Figure 2. The SVIP hardware architecture.

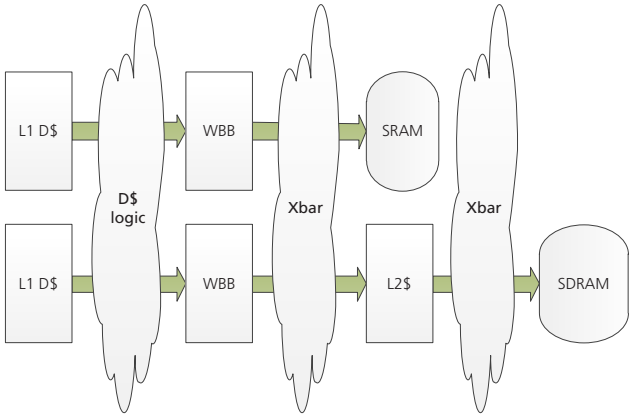


Figure 3. Procedure of two exemplary memory accesses, illustrating the SVIP cache hierarchy.

off-chip SDRAM accessed via core-private L1 caches and the shared L2 cache. Figure 2 depicts the relevant aspects of the hardware architecture.

In the absence of hardware support for maintaining cache coherence, the complex architecture of the memory subsystem is exposed via software interfaces. Figure 3 shows the components involved. When a core executes a store-operation, it only updates its core-private L1 cache. For the data to be propagated further, a writeback needs to occur, either because the cache line is evicted and replaced by another in normal operation, or through explicit issuing of a writeback command by the core. Before data actually leaves the core, it is held in a writeback buffer. Here it is delayed until either cache logic decides it is time to flush the buffer, or until such a flush is explicitly requested by the core. After leaving the writeback buffer, data is processed by the crossbar. The crossbar itself exposes no software interface and thus is out of the control of the individual cores. However, queuing and access reordering within the crossbar are possible and intended by design, making it unpredictable when and in which order memory accesses reach the L2 cache or the internal SRAM. The only operations available to influence the local caches are explicit writeback and invalidate instructions, as well as a flush instruction for the local writeback buffer. It is not possible for one core to invalidate cache lines in the private cache of another core.

B. Target Application

The software we use as target for our evaluation performs voice and telephony coding. It consists of roughly 150,000 lines of C code, and the resulting binary image has a size of about 800 kBytes.

C. Memory Model of the Cache Checker

The Cache Checker performs its address classifications based on a model of the systems memory usage, i. e. a list of memory objects (symbols) and conditions under which an access to the memory region in question is allowed from a specific core. The following conditions were implemented, based on the requirements of our target application:

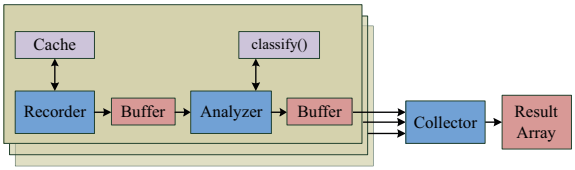
- *Always:* Access is allowed unconditionally, from all cores.
- *Never:* Access is forbidden unconditionally, regardless of core.
- *Core:* Access is allowed from certain cores.
- *Static array:* The memory region in question is an array, and there is a static assignment of ownership of array elements to cores. This is a convenience function that implements a number of core conditions.
- *Dynamic array*: The memory region in question is an array, and a decision on the ownership can only be made at run-time. Additional code is required, which can either implement instrumentation or evaluation of target software state to decide on the ownership.

D. Cache Checker Architecture and Configuration

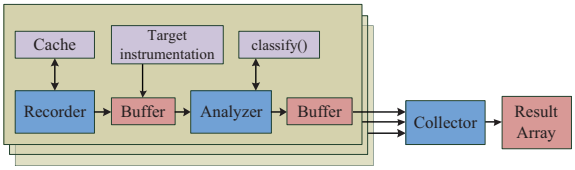
We evaluated several architectures for our Cache Checker, which are depicted in Figure 4. The presented architectures are based on a subset of the following components:

1. *Recorder.* One instance of the Recorder is needed on each core, which is responsible for examining the core-private caches. During this process interrupts need to be disabled to avoid a change of cache contents while the examination takes place.

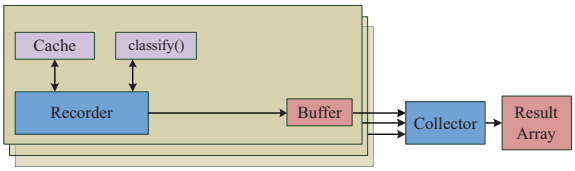
   The Recorder marks the first step of the cache inspection; it retrieves the currently cached memory locations, i. e. those locations that were previously accessed by the core, by examining the different cache lines of the core-private caches. All invalid cache lines are ignored and the corresponding addresses of valid cache lines are stored in a buffer for later analysis. The actual analysis can be postponed to reduce the time during which interrupts are disabled.
2. *Analyzer.* The stored addresses corresponding to the valid cache lines, as retrieved by the Recorder, are received and processed by the Analyzer. The Analyzer classifies the recorded memory accesses into the afore mentioned categories: *allowed* and *not allowed* accesses. In case an access is classified as *not allowed*, it is stored in another buffer for further processing. One instance of the Analyzer is present on each core.



(a) Basic "two-pass" architecture. The analysis is decoupled from the recording of the cache lines.



(b) "Two-pass" architecture with target instrumentation. The target software tracks changes in ownership of dynamically owned memory.



(c) Architecture for "one-pass" operation.

Figure 4. Alternative architectures of the Cache Checker

3. *Collector.* Unique among the complete Cache Checker system is the Collector. It receives the classification results from all Analyzer instances and stores them in memory. Periodically, this data is then transmitted via the network interface to the development host.

Three different architecture variants were evaluated for the Cache Checker: basic two-pass operation, two-pass operation with target instrumentation, and one-pass operation.

Basic two-pass operation: The basic architecture (cf. Figure 4a) aims at decoupling the different phases of the Cache Checker. Interrupts are only disabled during the actual accesses to the cache contents, while the analysis happens at a later time. However, in some situations, this approach is not applicable: When classifying an access to dynamically owned memory, the Cache Checker has to use internal information of the target software at run-time to determine whether the access is allowed or not. This information needs to be up-to-date, so it has to be guaranteed that the system state at the time of classification has not changed since the recording of the cache line.

Two-pass operation with target instrumentation: To overcome the limitations of the basic two-pass operation architecture, we identified two possible solutions. One is to instrument all code locations within the target software that cause a change of ownership of dynamically owned memory. We call this approach the two-pass operation with target instrumentation.

With the help of the instrumentation code it is possible to record ownership changes for all dynamically owned memory locations. During data collection it is important that the correct ordering of the change-of-ownership events and the observation of cache lines is maintained, e. g. through the use of one shared queue for all events. In this way, the information, which accesses were allowed at which time, is preserved for the Analyzer. Figure 4b shows the changes in architecture: The Analysis is still decoupled from the recording of cache data.

One-pass operation: The second solution to ensure that the system state does not change between observation of a specific cache content and classification, is to perform the classification immediately after the Recorder found a valid cache line, with interrupts still disabled. We call this one-pass operation, as opposed to the two passes executed when using the decoupled approach with a dedicated Analyzer component, as described before. This approach changes the Cache Checker architecture, as shown in Figure 4c: There is no dedicated Analyzer component; instead, the classification function is now called directly by the Recorder.

It depends on the target application which of the three architectures represents the best choice. The basic two-pass operation minimizes time at which interrupts need to be disabled by decoupling recording and analysis. It is however only applicable if there are no dynamically owned memory regions. The basic two-pass operation has similarly short interrupt disable times, but makes it necessary to instrument the target application. With the third option this instrumentation can be avoided at the cost of longer interrupt disable times. In our case, the application makes heavy use of dynamically owned memory regions, so the basic two-pass operation is unsuitable because it is conceptually unable to track dynamic changes of ownership. Of the two remaining options we gave preference to the one-pass operation, as it allows for the target application to remain unchanged.

In a system without hardware cache coherence, transferring data between software components is not trivial. In the Cache Checker we use ringbuffers for communication, which is safe even without the use of locks, provided that each buffer has only one exclusive reader and writer. In case the reader and writer reside on two different cores, additional care has to be taken. To guarantee a consistent state of the buffer, it has to be ensured that write access to individual cache lines can only happen from one single core. In practice this means that the head and tail pointers of the ringbuffers need to be located in dedicated cache lines.

Because of the disabling of interrupts during its operation, the Cache Checker is expected to change the behavior of the target application. To minimize this intrusion, we do not inspect the whole cache on each invocation. This means that the probability of detecting a coherence violation is decreased, leading to a tradeoff between higher inspection frequency and more severe influence on the run-time behavior of the target application. The lower probability to detect present cache coherence issues can however be compensated by performing longer test runs.

```
unsigned long classify(unsigned long pa) {
   if (pa >= 0x800000 && pa < 0x900000) {
      return 0x1;
   } else if (pa >= 0x900000 && pa < 0xc00000) {
      if (pa == 0x900000) {
         return 0x3;
      } else if (pa == 0x900020) {
         return 0x4;
      } else if (pa >= 0x900040 && pa < 0x900840) {
            ⋮
      } else if (pa >= 0x910640 && pa < 0x911240) {
            ⋮
      } else { // sram_cached
         return 0x108f;
      }
   } else { // unknown
      return 0x0;
   }
}
```

Listing 1. Automatically generated classification function: linear-if approach

```
unsigned long classify(unsigned long pa) {
   if (pa < 0xa88bc0) {
      if (pa < 0x986e20) {
         if (pa < 0x92d040) {
            if (pa < 0x915a40) {
               if (pa < 0x900040) {
                  if (pa < 0x900000) {
                     if (pa < 0x800000) {
                        return 0x1000;
                     } else { // (pa >= 0x800000)
                        return 0x1;
                     }
                  } else { // (pa >= 0x900000)
                     ⋮
                  }
               } else {
      .......
   }
}
```

Listing 2. Automatically generated classification function: binary-if approach

```
const unsigned long classification_table[] = {
/* 0 */ 0x00a88bc0, 0x000200e4,
/* 2 */ 0x00986e20, 0x00040074,
   ⋮
/* 452 */ 0x0096080c,
/* 453 */ 0x1f208000, 0x01c701c8,
/* 455 */ 0x10970001, // sram_cached(fragment)
/* 456 */ 0x00980001, // unknown(fragment)
};

unsigned long classify(unsigned long pa) {
   while (1) {
      x = table[current_line];
      c = x & 0xf;
      if (c == 0) { // compare
         y = table[current_line + 1];
         if (pa < x) {
            current_line = y >> 16;
         } else {
            current_line = y & 0xffff;
         }
      } else if (c == 0x1) { // no condition
         return x >> 16;
      } else if (c == 0xc) { // core condition
         if ((1 << core) & (x >> 8)) {
            return x >> 16;
         } else {
            return (x >> 16) ^ 0x1000;
         }
      }
   }
}
```

Listing 3. Automatically generated classification function: binary-table approach

E. Optimized Classification Function

We already mentioned above that the actual classification of physical memory addresses is performed by automatically generated code, the classify function. We provide three different implementation variants of this function. Source code extracts of the three variants are presented in Listings 1–3, and their discussion follows below.

1. *Linear-if:* The classification function takes the form of long nested if statements that reflect the memory usage model. This means that the address to be classified is compared with the lower and upper bounds of known memory regions, one by one. The if statements are nested to reduce the mean number of comparisons, but the nesting is only done at points that are reflected in the configuration. The result is that the top level if statement might select the memory segment, while the next might select the symbol and the last level of if statements might select the portions of an array that is used by different cores.
2. *Binary-if:* To minimize classification time and improve scalability, we provide the binary-if implementation variant, which is also based on nested if statements. Instead of employing long if/else if chains, which have a high mean execution time, we perform a binary search. This has two advantages over the previous approach: First, each if statement only has one *then* and one *else* branch, leading to a deterministic $O(\log n)$ execution time, and second, the condition of each if statement contains only one comparison.
3. *Binary-table:* Our third implementation also performs a binary search, with the exact same comparisons as the binary-if approach. But instead of many if statements that contain the addresses for comparison, we now operate based on a table. There is only little code executed to walk through this table.

Options 1 and 2 are designed to stress the data cache as little as possible. They contain many comparisons with immediate operands that reside in the executable code. In contrast, option 3 uses only little program code, but performs more memory accesses, thus influencing the contents of the data cache.

F. Performance Evaluation

We evaluate the performance of the Cache Checker within the real-world application described in Section III-B. The internal

architecture used is the one-pass operation (cf. Figure 4c). This option was chosen to a) support dynamically owned memory, and b) leave the application itself unchanged and free of instrumentation code. To classify dynamically owned memory, internal data structures of the target application are examined at run-time by the Cache Checker. The results presented in this section were obtained through measurements using the internal cycle counter of the CPU cores.

It should be noted that these measurements are not supposed to set the performance of the Cache Checker into relation with other tools, nor are they supposed to provide an indication of the speed at which the Cache Checker might discover bugs. Instead, they are intended to give an indication of the processing overhead induced by the Cache Checker in various configuration scenarios.

With regard to the code size of the Cache Checker, we examine first the checker itself, without the automatically generated classification function, and then the classification function. We compile using the GNU Compiler Collection, with an optimization setting of *–O2 –G1 –mabi=32 –mdsp –mips32r2 –mtune=24kec –nostdinc*.

The complete code size of the Cache Checker, without the classification function, is 2824 bytes. The required memory for data representing internal state (including ring buffers and data for performance analysis and run-time measurement) is $n_{\text{cores}} \cdot (188 \text{ bytes} + 8 \text{ bytes} \cdot l_{\text{ringbuffer}})$, for the result array $12 \text{ bytes} \cdot l_{\text{resultarray}}$, and for the notification message $88 \text{ bytes} \cdot n_{\text{cores}} + 248$ bytes, where n_{cores} is the number of cores, and $l_{\text{ringbuffer}}$ and $l_{\text{resultarray}}$ denote the number of entries in the ringbuffers and the result array.

For the classification function, code and data sizes depend on the number of distinctive memory regions that are classified. Figure 5 shows the memory required for a classification of statically owned memory only. The figure shows the combined code and data memory used. In case of the linear-if and binary-if approach, the memory used contains only program code and no data elements, while for the binary-table option, the memory usage includes a constant code size of 216 bytes and a variable data size for the classification table.

The addition of dynamically owned memory regions to the memory usage model increases the code size by 120 bytes per dynamically owned region.

To assess the execution time of the Cache Checker, we examine its components (Recorder, Classification, and Collector) separately. First we consider the case when the target system is idle, later we also give a statement on the behavior while the system is under load.

The mean execution time of one invocation of the Recorder component of the Cache Checker is composed of:

$$t_{\text{invocation}} = n_{\text{lines}} \cdot \rho_{\text{valid}} \cdot (t_{\text{record}} + t_{\text{classify}}) + n_{\text{lines}} \cdot (1 - \rho_{\text{valid}}) \cdot t_{\text{don't record}}$$

where n_{lines} denotes the number of cache lines examined per invocation, ρ_{valid} the fraction of cache lines that turn out to be valid, and t_{classify} the execution time of the classification function. The two times t_{record} and $t_{\text{don't record}}$ are the times within the Recorder that are required to process a cache line, either by classifying it and passing it to a ringbuffer if the cache line is valid, or by discarding it if it is not. Our measurements found that $t_{\text{record}} \approx 90$ cycles and $t_{\text{don't record}} \approx 40$ cycles.

The execution time of the classification function t_{classify} depends on the classification method (cf. previous section) and the number of different memory classes that are distinguished. Figure 6 shows the mean execution time of the classification function when only statically owned memory is used. For dynamically owned memory, in our case, an additional ≈ 50 cycles are needed to complete the classification.

The Collector component, which is only executed on one core, receives cache violations from the Recorder components and stores them in a result array. When a violation is received, it is first checked if the same address is already present in the array. If it is, then we simply increment a counter, otherwise we create a new entry in the result array. It is clear that storing an incoming violation into the result array takes longer, if many violations have been collected previously. In our scenario, the mean execution time of the Collector is ≈ 300 cycles when the result array is empty.

To observe the Cache Checker performance when the system is loaded, we measured execution times and cache misses in several different load scenarios. Due to the specific target application, it was not possible to stress the system systematically and with a representative load. However, we observed the following two patterns:

1. When there are more modules of the target application active, the fraction of valid cache lines increases, leading to more classifications that need to be performed.
2. When the system is loaded, the stress on the caches increases, leading to reduced overall performance, which includes the performance of the Cache Checker.

Our evaluation of the three classification approaches revealed that the binary-if classification performs best in terms of execution times, followed by the liner-if and the binary-table approaches. The binary-if and linear-if solutions have similar memory footprints, while the memory usage of the binary-table approach is considerably smaller. However, it has to be noted that when using the binary-table approach, the Cache Checker does not observe all violations that are visible to the other two approaches. This is due to the fact that this approach makes heavy use of the data cache, which can lead to the eviction of cache lines of the target application, thus hiding their presence from the Cache Checker.

One other parameter that was examined during our study is n_{lines}, the number of cache lines checked per invocation of the Cache Checker. In the 4-way set associative cache, our implementation checks in one step all four ways corresponding to one index. Choosing higher values for n_{lines} increases the frequency at which the complete cache is examined, but also leads to higher overall execution times of the Cache Checker.

After completing the evaluation of different implementation approaches for the Cache Checker, we chose a configuration suitable for our use case: We chose the architecture option depicted in Figure 4c, which supports dynamically owned

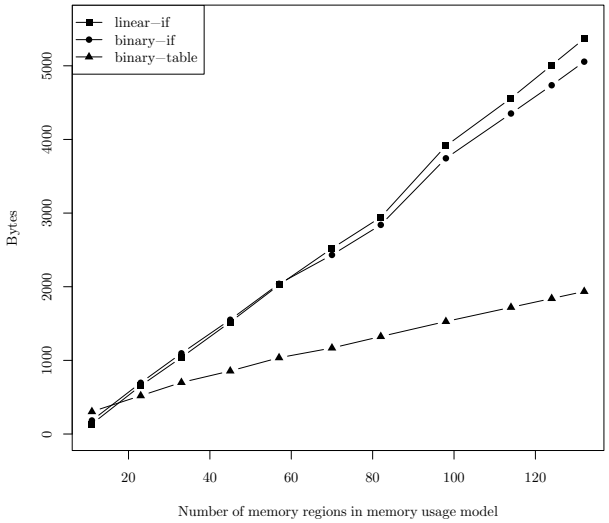


Figure 5. Code and data size of classification function

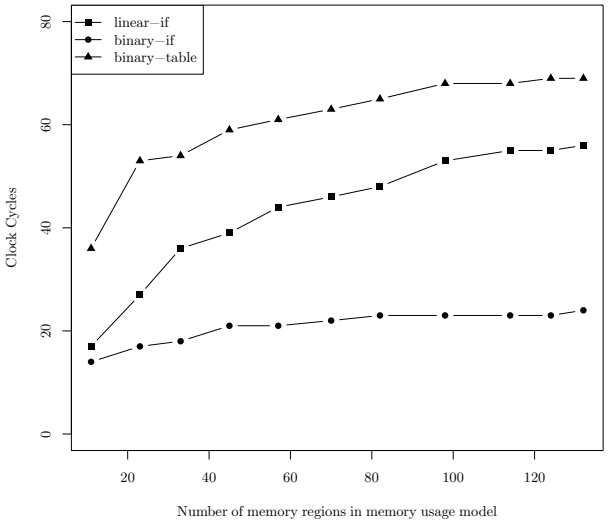


Figure 6. Execution times of classification as a function of the number of memory classes

memory, but does not need changes to the application. For classification we use the binary-if approach, which provides best performance and scalability out of the three options. Finally, we decided to limit our checks to the minimum value of 4 cache lines per invocation to minimize overall Cache Checker execution time.

IV. Discussion

After assessing the performance of the Cache Checker in the previous section, we now want to discuss its value to the software developer. As it is intended to be a tool to help in the detection and localization of software defects, it is hard to quantify its usefulness. We report here our experience after using the Cache Checker within the large real-world application described in Section III-B.

We re-introduced to the application a number of programming mistakes related to the handling of cache coherence which had previously been detected and fixed manually. In several of these cases, our tool generated a warning even after short application runs, providing information on the memory region in question, and thus helping in the identification of the affected program module. In addition, our Cache Checker also detected previously unknown issues. These violations of cache coherence occurred only occasionally during long test runs, and had so far gone undetected, because they did not cause any visible application malfunctions or data corruption.

We designed the Cache Checker to be a tool that examines core-private caches based on rules. We have seen that the concept is indeed valuable in two different situations:

1. It helps to find the location of software defects and significantly reduces the time needed to fix them.
2. It can be used in software testing to uncover issues before they become visible as application malfunctions and thus help increase software quality.

Maintaining cache coherence with hardware logic is complex. For the platform used in our evaluation, it is estimated, that 10–12% chip area was saved by not including such logic in the design. Especially in embedded devices, these savings present an opportunity, if the system software can handle the difficulties. Our approach presents a first step into this direction.

Other approaches to simplify software development on platforms without hardware-support for cache coherence can be based on a supporting compiler, middleware, or a simulation environment. The compiler could rearrange the code within an optimization pass in such a way that cache safe memory accesses are guaranteed. A specifically designed middleware or operating system could provide an interface for cache safe memory accesses, reducing the occurrence of critical memory access code to view locations within the operating system or middleware. Finally, an elaborated simulation environment could execute i. e. simulate an application, detect critical memory accesses and log them. With the help of debugging symbols this log could indicate actual code locations to the developer that require special attention. Our approach, however, is based on a minimal software system that can be integrated into arbitrary system code without efforts. Nevertheless, it provides as much information to the developer as the other approaches do.

DeOrio et al. [7] proposed a post-silicon verification concept for MESI cache coherence protocols. Their concept is based on periodical checks of the cache contents, but in contrast to our approach, DeOrio et al. require hardware support in form of logging components within the L1 and L2 cache controllers. They quantify the area overhead of these components to only 0.002%. The logging components record all cache operations within a time period (a so called epoch). The resulting log includes, among other information, the timestamp and the MESI state for each cache operation. After each logging period all

processor cores are stalled and a checking phase is initiated. During the checking phase DeOrio et al. propose to compare the histories of the different cache instances, i. e. all local L1 caches and the shared L2 cache. When inconsistencies between the histories are detected, a bug within the cache coherence protocol was found. However, DeOrio et al. strictly focused their approach on the verification of an existing cache coherence protocol, it is not clear whether their approach is transferable to platforms without cache coherence and how much area overhead it would induce on such a platform.

Denisco and Beaverson [8] propose a software based cache examination and coherence protocol verification concept, based on an arrangement of multiple slave (collector) processes and a central master process (checker). The slave processes continuously collect information about the data addresses and control flags contained in the local caches and save them in memory. Every time the master / checking process is invoked it checks these collected information against a predetermined table with valid cache states of the system. Inconsistencies between the predetermined and the collected cache states mark errors in the coherence protocol. Denisco and Beaverson, however, left open how such a predetermined table could be created and how this creation process might scale to other systems with bigger cache sizes or more cores.

Cheong and Veidenbaum [9], as well as Choi et al. [10] propose techniques for compiler-directed cache coherence. Both groups suggest purely software-based concepts along with concepts that utilize small hardware extensions to reduce the processing overhead. However, to the best of our knowledge their exists no completely software based solution that could be used to detect and locate cache coherency bugs in system software. Thus, our Cache Checker represents the first tooling support for developing system software on platforms without hardware based cache coherence logic. Moreover, current research focuses mainly on hardware based cache coherence protocols and their verification.

Future work to improve the current version of the Cache Checker can go in several directions. One possibility is the extension to cache examinations triggered by interrupts, to not be limited to certain points in time. Another option would be synchronized cache examinations on all cores. This way we could detect, if two cache lines are simultaneously active on more than one core.

V. Conclusion

In this paper we presented our Cache Checker, a new kind of software tool that performs online checks of cache contents. It is useful on multi-core platforms that do not contain hardware support to maintain cache coherence, and thus require that all cache handling be done in software. On these hardware platforms, there are so far no tools to support the developer; our approach significantly simplifies debugging and testing of software on such systems.

We evaluated several implementation variants and optimizations of our tool, which resulted in a low-overhead solution for online cache inspections.

Use of the Cache Checker in a large-scale software system proved its success. We showed that it can detect cache-related programming mistakes, thus simplifying development and testing of software on platforms that do not provide hardware support to maintain cache coherence.

Acknowledgement

The research that lead to this paper was funded by the Bavarian Ministery of Economic Affairs. Work was carried out in cooperation with Lantiq Deutschland GmbH.

References

[1] M. S. Papamarcos and J. H. Patel, “A low-overhead coherence solution for multiprocessors with private cache memories,” *SIGARCH Comput. Archit. News*, vol. 12, no. 3, pp. 348–354, Jan. 1984.

[2] P. Sweazey and A. J. Smith, “A class of compatible cache consistency protocols and their support by the IEEE futurebus,” *SIGARCH Comput. Archit. News*, vol. 14, no. 2, pp. 414–423, May 1986.

[3] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, “Introduction to the Cell multiprocessor,” *IBM Journal of Research and Development*, vol. 49, no. 4.5, pp. 589–604, Jul. 2005.

[4] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin, “Programming the Intel 80-core network-on-a-chip terascale processor,” in *Proceedings of the ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 38:1–38:11.

[5] *Vinetic-SVIP16 Hardware Unit Architecture Specification*, Infineon Technologies, 2007.

[6] *MIPS32 24KEc Processor Core Datasheet*, MIPS Technologies, 2008.

[7] A. DeOrio, A. Bauserman, and V. Bertacco, “Post-silicon verification for cache coherence,” in *Proceedings of the IEEE International Conference on Computer Design*, Oct. 2008, pp. 348–355.

[8] J. A. Denisco and A. J. Beaverson, “Multiprocessor cache examiner and coherency checker,” U.S. Patent 5,406,504, Apr. 11, 1995.

[9] H. Cheong and A. Veidenbaum, “Compiler-directed cache management in multiprocessors,” *Computer*, vol. 23, no. 6, pp. 39–47, Jun. 1990.

[10] L. Choi, H.-B. Lim, and P.-C. Yew, “Techniques for compiler-directed cache coherence,” *Parallel Distributed Technology: Systems Applications, IEEE*, vol. 4, no. 4, pp. 23–34, Winter 1996.