Light-PREM: Automated Software Refactoring for Predictable Execution on COTS Embedded Systems

Renato Mancuso*, Roman Dudko*, Marco Caccamo*,
*University of Illinois at Urbana-Champaign, USA, {rmancus2, dudko1, mcaccamo}@illinois.edu

***Abstract*—As real-time embedded systems become more complex, there is the need to build them using high performance commercial off-the-shelf (COTS) components. However, tasks can exhibit hard to predict worst case execution times (WCET) when executing on commodity hardware, due to contention among shared physical resources. Past work has introduced the PRedictable Execution Model (PREM) [1] to solve this issue, but unfortunately, the time required to manually refactor existing code according to this model is too high. Light-PREM proposes a novel technique that automates the refactoring process needed to convert legacy software applications to PREM-compliant code. The advantage of Light-PREM is twofold. On one side, it makes the adoption of PREM more attractive from an industrial point of view, because it significantly reduces the amount of work that is needed to generate PREM-compliant code. On the other hand, the proposed methodology is general enough to be used with any embedded software design. Experimental results show that Light-PREM significantly improves the predictability of real-time applications without requiring software engineers to gain a deep understanding about software memory usage.**

I. Introduction

Real-time embedded systems are becoming increasingly more complex and have stringent performance requirements. Hardware components specifically designed for real-time systems ensure temporal predictability, but often fall behind in terms of performance when compared to commercial off-the-shelf (COTS) components whose time-to-market is significantly shorter. As such, there is a need to use COTS components to improve performance of real-time embedded systems, but their usage introduces significant challenges at integration time since COTS components are not designed to support real-time systems. COTS-based systems are highly optimized for average performance, but introduce execution time spikes in the worst-case due to non-realtime contention policies that are in use to regulate the access to shared physical resources (such as main memory, system bus, I/O). Specifically, if multiple I/O peripherals and a CPU attempt to access the bus at the same time, the resulting unregulated contention may unpredictably delay activity of I/O peripherals or the CPU.

The PRedictable Execution Model developed in [1] focused on solving this problem. PREM introduces a new execution model that aims to improve predictability of worst case execution times (WCET) by using a high-level co-scheduling mechanism for peripherals and CPU. As recalled by Section II, PREM enforces a structured memory access pattern on executables, so that exclusive access to the shared system bus can be granted to either a single task or the I/O subsystem, regulated by a peripheral scheduler [2]. Use of PREM allows system designers to use COTS components to run real-time applications, and gain the advantage of higher performance without experiencing unpredictable execution delay. Migrating from legacy code to PREM-compliant code requires adding PREM annotations correctly inside the application code. Thereby, one of the major disadvantages of PREM is that converting a legacy application to a PREM-compliant executable requires either having a deep understanding of the application code or performing reverse engineering to place PREM annotations correctly. Such process is time consuming, and difficult for large-scale applications. The presented work aims to overcome this problem by introducing a novel technique to automatically perform legacy-to-PREM application refactoring. In summary, this work proposes an automated process, named *Light-PREM*, to support the adoption of the PREM model for arbitrary complex applications.

In PREM, the focus is on enforcing the aforementioned deterministic memory access pattern inside single C functions, defined as *predictable functions*. Thus, Light-PREM automatically generates PREM annotations which include prefetch statements for majority of memory locations accessed inside the function under analysis. In this way, when the predictable function executes, the prefetch block placed at the beginning of the function will bring into the local CPU cache those cache lines that will be accessed during its execution. Light-PREM determines which memory accesses to prefetch by using memory profiling tools. It then determines how to perform such prefetches by translating the absolute memory addresses recorded at profiling time into chains of pointer dereferences that always begin with an in-scope variable. It does so by intercepting memory allocation procedures and constructing a graph representing pointers between various memory blocks. This technique is compiler-independent and experimental results show that it reduces cache misses by 95% in a non-trivial benchmark suite like the JPEG Image Encoding Benchmark.

The contribution of this work is twofold. First, Light-PREM is able to perform automatic refactoring of arbitrarily complex legacy applications according to the PREM model. This is crucial to make PREM a viable solution for real-time and embedded industry. Second, it proposes a novel, compiler-independent strategy to generate prefetch statements relying on off-line data profiling. As we show, the technique can succesfully produce PREM-compliant code if (A) the predictable interval contains no system calls and references to dynamically liked objects, and (B) dynamic memory allocations are performed outside the predictable interval.

The rest of the paper is organized as follows. First, Section II briefly recalls the basic principles and key ideas behind PREM. Section III provides a high-level description of Light-PREM. Next, Section IV provides insights on the implementation of Light-PREM. The presented results in Section V show how Light-PREM is able to perform automatic refactoring of a legacy application according to the PREM model. An overview of the related work is available in SectionVI. Finally, the paper concludes in Section VII.

II. Background

In [1], the PRedictable Execution Model (PREM) was proposed. The intuition behind PREM is that in the general case,

the memory access patterns of tasks executed on commodity hardware can exhibit a high variance. In particular, predicting an exact sequence of fetches in main memory is extremely difficult, while, at the same time, assuming the worst case scenario leads to very pessimistic assumptions. Thereby, the key idea in PREM is to enforce, for a given set of tasks, a novel execution model with three main features:

1) jobs are divided into a sequence of non-preemptive scheduling intervals;
2) some of these scheduling intervals (named **predictable intervals**) are executed *without cache misses* by prefetching all required data at the beginning of the interval itself;
3) the execution time of the **predictable intervals** is kept constant by monitoring CPU time counters at run-time.

More in detail the code for each task τ_i is divided into a set of N_i scheduling intervals $\{s_{i,1}, \ldots, s_{i,N_i}\}$, which are executed sequentially at run-time. The timing requirements of τ_i can be expressed by a tuple $\{\{e_{i,1}, \ldots, ei, N_i\}, p_i, D_i\}$, where p_i, D_i are the period and relative deadline of the task, with $Di \leq pi$, and $e_{i,j}$ is the maximum execution time of $s_{i,j}$, assuming that the interval runs in isolation with no memory interference. A job can only be preempted by a higher priority job at the end of a scheduling interval. This ensures that the cache content can not be altered by the preempting job during the execution of an interval. The scheduling intervals are classified into *compatible intervals* and *predictable intervals*.

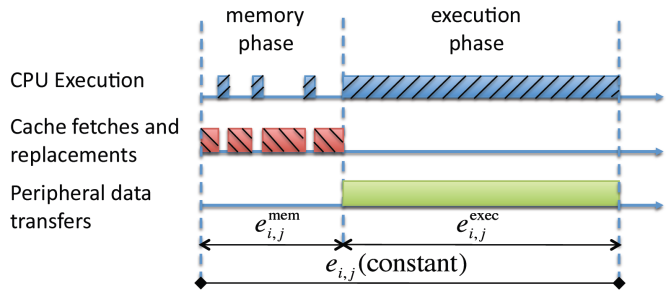

Fig. 1. Predictable Interval with constant execution time.

Compatible intervals are not characterized by any special property (they are backwards compatible). Cache misses can happen at any time during these intervals. The task code is allowed to perform OS system calls, but blocking calls must have bounded blocking time. Furthermore, the task can be preempted by interrupt handlers of associated peripherals. It is assumed that the maximum execution time $e_{i,j}$ for a compatible interval is computed based on static analysis techniques. However, to reduce the pessimism in the analysis, peripheral traffic is prohibited from being transmitted during a compatible interval. Ideally, there should be a small number of compatible intervals which are kept as short as possible.

Predictable intervals are specially factored to execute according to the PREM model shown in Figure 1: they are divided into two different phases and exhibit three main properties.

- First, during the initial memory phase, the CPU accesses main memory to perform a set of cache line fetches and replacements. At the end of the memory phase, all cache lines required during the predictable interval are available in last level cache.
- Second, during the following execution phase, the task performs useful computation without suffering any last level cache misses. Predictable intervals do not contain any system calls and can not be preempted by interrupt handlers. Hence, the CPU does not perform any external main memory access during the execution phase. This property allows peripheral traffic to be scheduled during the execution phase of a predictable interval without causing any contention for access to main memory.
- Third, at run-time, the time length of a predictable interval is forced to always be equal to $e_{i,j}$.

Note that in the original formulation of PREM, predictable intervals cannot contain any system call and cannot be preempted by interrupt handlers, to prevent the CPU from performing external main memory access during the execution phase. Not enforcing such constraint would result in two undesirable effects: (1) the OS would perform some transactions in main memory violating the separation between memory phase and execution phase; (2) the memory transactions performed by the OS could displace some memory locations that have been allocated in cache during the memory phase.

Such limitations can be addressed by relaxing the mentioned constraints on predictable intervals. Specifically: problem (1) can be solved by computing an upper bound on the number of memory accesses performed by the OS to satisfy a system call (or to execute an interrupt handler) and accounting them when computing the WCET; problem (2) can be addressed by using deterministic cache allocation mechanisms that have been largely studied [3, 4, 5]. We are currently investigating the exploitation of cited cache allocation mechanisms as part of our future work.

The big advantage of having a real-time task executing according to the PREM model is that it enforces a predictable memory access pattern. Thereby, it becomes possible to perform high-level coarse-grained real-time scheduling among multiple masters contending for access to a shared physical resource (like a shared bus and/or memory controller). For instance, it becomes possible to co-schedule memory transactions from a real-time I/O subsystem [2] in a way that they do not interfere with the execution of PREM-compliant real-time tasks. Recently, a Memory-Centric Scheduling framework [6] was also proposed that uses the PREM model to co-schedule memory accesses from contending CPUs of a multi-core platform: from the point of view of each core, the memory subsystem is seen as a slower but isolated one avoiding the experiencing of unpredictable temporal behavior due to a shared memory architecture. Finally, since the overall length of the *predictable interval* is constant, it is always possible to perform schedulability analysis of I/O transactions in the way it has been discussed in [1].

III. LIGHT-PREM

In this section, we recall the motivation for Light-PREM. Moreover, we provide some insights about the high-level approach as well as the key ideas behind it.

A. Motivation

Configuring a system to exploit the benefits of the PREM model requires reworking application code so that tasks are split into scheduling intervals, which must either conform to the requirements of a compatible interval or a predictable interval. However, such legacy-to-PREM refactoring requires dividing tasks into scheduling intervals, and to manually add prefetch statements for all those memory locations that are needed in the execution phase of each predictable interval. This last step involves either having a perfect knowledge of the semantics or reverse engineering the code. Thus, as the size of the source code increases, so does the time required to manually perform this process. As such, this method does not scale with the size of the code base.

Light-PREM solves this problem by automatically generating prefetch statements for the memory phase of predictable intervals. In other words, it can be considered as a code refactoring tool that is able to enforce the separation between memory phase and execution phase inside what has been tagged by the programmer to become a predictable interval. It is worth highlighting one more time that such simplicity in performing legacy-to-PREM conversion of an arbitrarily large number of real-time tasks enables the use of novel scheduling techniques that can rely on a highly predictable memory access pattern to coordinate the access of several masters to the shared bus [2, 6, 1].

A possible solution to this problem could involve operating at a compiler level. Since the compiler already parses the source code, it generates data structures, such as the Abstract Syntax Tree (AST), which would allow much easier manipulation of the program. We investigated utilizing compiler passes such as those offered by the LLVM [7] compiler framework to automatically add prefetch statements at compile time. Unfortunately this method would introduce a dependency on the compiler tool-chain, thus breaking the property of Light-PREM to remain **compiler-independent**. This property is fundamental because industries often use proprietary compilers, so that: (1) reimplementing Light-PREM on a custom compiler would result in an unreasonable effort; (2) they would be unwilling to switch to third-party compilers. Thus, we implemented Light-PREM at the source code level instead. As we show, this technique yields excellent results, even though Light-PREM can achieve a knowledge of the code semantics which is significantly lower than what can be achieved by the compiler. As shown in Section V, Light-PREM is able to achieve performance comparable to manual refactoring.

B. Overview

The refactoring process performed by Light-PREM requires (1) analyzing the function to determine what memory accesses occur, and (2) generating prefetch statements to construct the memory phase (note that the memory phase is located at the beginning of the function). From an high-level point of view, the steps performed by Light-PREM are the following:

1) **access collection**: memory accesses performed inside the predictable function are recorded;
2) **chunk detection**: allowed memory ranges that can be accessed by the application are determined;
3) **handle detection**: in-scope variables (handles) at the beginning of a function are detected;
4) **graph construction**: chunks are interconnected detecting the existence of pointers from one chunk to another;
5) **relative expression construction**: prefetch statements are built for any memory access that was detected at Step 1.
6) **prefetch aggregation**: subsequent, contiguous prefetch statements are aggregated together to minimize the overall number of added statements.

C. Access collection

Since analyzing the behavior of a function from a semantic point of view requires either programmer or compiler knowledge, Light-PREM approaches the problem with a sample-based technique instead. Specifically, we perform a dry run of the function under analysis to record all the performed memory accesses. The output of this phase is a raw list of all the virtual memory addresses referenced by the predictable function. Such list represents the first step that is required to attain the goal of generating prefetch statements for all the memory misses in the execution phase.

The described memory access collection procedure can be performed using standard and architecture independent profiling tools, as detailed in Section IV. It is also important to underline that having a perfect knowledge of the accessed memory locations is not enough to generate prefetch statements that can be permanently added to the source code. This is because virtual memory addresses change from run to run and thus, a prefetch on a constant memory address may result in an invalid addressing request in subsequent runs.

D. Chunk detection

Each application owns a set of ranges of virtual addresses that are accessible at execution time. Such ranges are assigned at loading time or obtained at execution time with dynamic memory allocation. We refer to a valid range of virtual addresses as a *memory chunk*, or simply *chunk*. Detecting the memory chunks belonging to the application under analysis has two main purposes. First, it is needed to locate each observed memory access in the corresponding chunk. Second, it is required to understand how chunks are interconnected between each other.

As it will be clear in the graph construction step, the majority of nodes in the memory graph are chunks. They are defined in terms of starting address and size. To capture chunks, Light-PREM intercepts memory allocation routines and updates internal bookkeeping data structures accordingly. Light-PREM also considers global variables, and other memory segments (e.g stack, text, data) treating them as chunks.

E. Handle detection

As previously defined, variables that are in-scope at the beginning of the function under analysis are considered *handle* variables. Thus, from this definition it follows that a handle can be either a global variable or a local function parameter[1]. Since a final prefetch statement will explicitly reference a handle, the names of such variables must be determined.

Retrieving function parameters is relatively simple and can be done by using code formatting tools and regular expressions on the source file. Conversely, finding the names of global variables can be nontrivial if several files are involved in the compilation process. For this reason, this step is performed by analyzing the executable file produced at the end of the compilation process. Standard Linux tools can be used for this purpose, as detailed in Section IV. Once the name of the handles is discovered, what gets recorded is the correspondence between handle name and memory address. This last information is important to detect in which chunk a given handle is contained and thus which other chunk can be reached from it.

F. Chunk linking and graph construction

Once the description of each chunk is available, in terms of start address and size, Light-PREM attempts to link them together. For example, two chunks C_1 and C_2 are linked if in C_1 there exists a pointer to any location inside C_2. Performing this operation on each couple of chunks allows building a directed multigraph which reflects the memory layout of the program under analysis.

This graph is formally defined as $G = \{V, E\}$ where V is the set of vertices and E is the set of edges. Each vertex C_i represents a memory chunk, and it has additional data associated, such as the starting address a_i and its size s_i in bytes. Assuming that there are n memory chunks, $V = \{C_1 = \{a_1, s_1\}, \ldots, C_n = \{a_n, s_n\}\}$. Each edge in the graph represents a pointer that connects two vertices C_i

[1]This definition of a handle is valid for code written in C.

and C_j. However, more than one pointer directed to C_j can be found in C_i. What differentiates two pointers that are outgoing from the same chunk and are directed to the same node is: (1) the relative position inside the source chunk and (2) the pointed location in the destination chunk. For this reason, each edge holds additional data to keep track of the discussed pieces of information. Respectively, we will refer to them as δ^s and δ^d.

Such graph is constructed at the beginning of execution of the target function. Figure 2 depicts the graph reflecting the memory layout of a sample application. In the figure, there are four vertex correspondent to four memory chunks. One of them - the vertex represented with a rectangle - is also an handle. For a given pair of vertexes, say C_i and C_j and a given interconnecting edge δ_k the following property holds:

$$[a_i + \delta_k^s] - \delta_k^d = a_j$$

Where the square brackets denote the "cast & dereference" operator, i.e. what is contained inside the brackets is interpreted as a pointer and accessed to obtain the value of the referenced memory location.

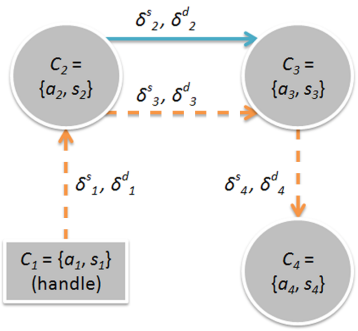

Fig. 2. Memory layout in a sample scenario

Note for instance that according to the chain of arrows highlighted in green in Figure 2, it is possible to reach chunk C_4 from the handle C_1 with the following expression:

$$[[[a_1 + \delta_1^s] - \delta_1^d + \delta_3^s] - \delta_3^d + \delta_4^s] - \delta_4^d = a_4$$

G. Relative expression construction

Once the graph has been built and all the interconnections between the chunks have detected, it is possible to generate the prefetch statements performing graph traversal. Specifically, the procedure for the creation of a prefetch statements accepts three inputs: (1) the memory address accessed inside the predictable function and recorded at run-time; (2) the set of variables that are in-scope (handles) at the beginning of the function; (3) the memory layout of the targeted application expressed as a directed multigraph as explained in the previous section.

Without loss of generality, a variable that is accessed inside a function is either a in-scope variable or a memory location that can be reached with an arbitrarily long chain of dereferences. Light-PREM is able to capture all the interconnections between the different chunks in the memory layout of an executable. Thereby, for an observed memory access in one of the considered chunks, Light-PREM can build a valid dereference chain from a handle to any memory location recorded at execution time. Such a dereference chain, starting from a handle, is called a *relative expression*.

Provided the aforementioned oriented multigraph of the memory layout, finding the *best* dereference chain translates into navigating the graph upwards, i.e. in a reverse direction with respect to the orientation of the edges. As detailed in Section IV, a combination of two metrics is used to determine which chain is the preferred one in a pair of equivalent dereference chains. Considering the example reported in Figure 2, a memory access at a location λ positioned at an offset δ_λ inside chunk C_4, will lead to the generation of the following prefetch statement:

$$\texttt{PREFETCH(handle} \xrightarrow{\delta_1} C_2 \xrightarrow{\delta_3} C_3 \xrightarrow{\delta_4+\delta_\lambda} \lambda)$$

Where the notation $\xrightarrow{\delta_k}$ denotes the dereference operation applied using the offsets δ_k^s and δ_k^d as previously explained.

As previously stated, the generated prefetch statements are placed back into the source code of the executable. However, if the memory layout graph of the application under analysis at run-time is different from what observed in the profiling stage, invalid memory references can be encountered while following a dereference chain. In general, it can be the case if there is a dependency between a) the input vector and the allocation order or the size of the chunks; b) the input vector and the existence or position of the interconnecting links (pointers). Unless the targeted code has a well known deterministic structure in this sense, the inherently heuristic nature of some of the processing steps involved in Light-PREM can lead to the generation of invalid memory references.

To address this critical issue, our approach consists of setting up a defensive mechanism. Specifically, a segmentation fault handler (i.e. a handler for the SIGSEGV signal) is installed at the beginning of the memory phase. Furthermore, at process setup, the permissions of those memory pages of the text section corresponding to the memory phase of predictable intervals are changed, so to allow write operations[2]. In this way, if during the memory phase a broken dereference chain triggers a segmentation fault, it is caught by the handler which replaces the faulting instructions with NOPs, so to turn them inoffensive when re-executed after the handling procedure returns. The described handler is used after the refactoring has been completed to remove all those prefetch statements that are found to be faulty upon a change in the input vector. Moreover, since we have observed that it has a negligible overhead in terms of execution time and number of memory references (the handler has less than 10 lines of code), it can be left in production code. This ensures that the correctness of the refactored application is preserved.

H. Prefetch aggregation

According to the locality principle, there is an high probability that, if a given memory location is accessed, subsequent accesses will interest surrounding memory locations. For this reason, instead of generating a prefetch statement for every single observed memory access, we aggregate single prefetches in longer prefetch sequences.

The advantage of performing the aggregation step is twofold. First, the number of lines that becomes part of the executable code is minimized. Second, it is important to consider that a single prefetch statement can result in several memory dereferences. Thus, by aggregating them it is possible to calculate the starting position once and to perform the prefetch sequentially with a remarkable run-time improvement.

IV. Implementation Details

In this section, we will detail our implementation of Light-PREM to target general purpose applications written in a language that allow memory manipulation through pointers, like C.

[2]This can be done in Linux through the `mprotect` system call

A. Run-time data collection

As discussed in Section III, Light-PREM employs a multistage technique to produce PREM-compliant code out of a legacy application. Due to the fact that PREM, as a proof of concept, is currently implemented in C, the presented Light-PREM sets its focus on programs written in C.

1) Access collection: As previously discussed, the first step performed by Light-PREM is collecting a raw list of all the virtual memory addressed referenced inside the predictable function. For this step, the targeted code is compiled and run inside a profiling environment. Specifically, Light-PREM uses a memory profiling tool called Lackey (a sub-tool of the Valgrind suite) to analyze the task at run-time. Since the goal is to detect all the memory accesses that are performed inside the body of the function under analysis, no additional data are stored about each accessed location.

2) Chunk detection: In order to place the accessed memory locations in the appropriate chunk, it is fundamental to detect which memory regions are accessible by the application at the beginning of the predictable function. Such chunks will also represent the nodes of the memory layout graph. As previously mentioned, they are defined by a starting address and size.

At the beginning of the predictable function, there is a set of memory regions that are added to the list of chunks by default. Specifically, these are: (1) the stack area belonging to the predictable function (whose size is always known at run-time); (2) the text area of the executable; (3) .data, .rodata and .bss sections if not empty; (4) global variables and (5) predictable function parameters. Each chunk in the last two categories is also a handle.

Furthermore, to capture all the possible chunks created through dynamic memory allocation routines (*malloc*, *mmap*, etc.), Light-PREM intercepts such routines and updates its internal book-keeping data structures accordingly. Intercepting the aforementioned calls without modifying the executable code can be done by temporarily overriding the targeted primitives at dynamic linking time.

3) Handle detection: In a standard C program, only two kinds of variables are in-scope at the very beginning of a function (before any automatic variable is declared): function parameters and global variables. This is the reason why the only variables that need to be considered as handles are all those variables which fall in one of these two categories.

Since each generated prefetch statement will involve dereferencing a handle, the name of these variables must be determined. Retrieving function parameters can be done in a relatively simple way by using regular expression matching based on the function name. This procedure can be made resilient to different coding styles performing a preliminary pass in a code formatting tool (e.g. Uncrustify). To capture the name of all the global variables, we rely on the data that are contained inside the final executable. In particular, ELF[8] files define a *symbol table* in which objects with binding attribute set to `GLOBAL` and default visibility properties refer to global variables. Variables that are global but not exported at linking time (static) are also considered in a similar way. As a final step of the run-time data collection phase, handle variables are recorded in terms of name and memory address.

B. Prefetch generation

The remaining part of the process involves steps that can be performed offline relying on the information collected during the previous phase.

1) Graph Construction: As described earlier, while the chunks are the vertex of the memory layout graph, edges represent pointers, i.e. links from one vertex to another. Edges are generated after all vertices (chunks and handles) are discovered and created. This step involves recognizing the links between the chunks and adding the detected vertex with appropriate weights. The algorithm which is used to perform this operation is called "Pointer Probing". It can be described as follows: let the size of a pointer in the targeted platform be P bytes. For each chunk C_i, Light-PREM reads the value v of every contained P byte block. Next, it checks if the given value can be interpreted as a pointer. That is, v should be an address which falls into the range $\langle a_j, a_j + s_j \rangle$ for some chunk C_j.

If a given value v satisfies the above property, an edge is created from chunk C_i to chunk C_j. For said edge, the value of δ_s is the position where v was found in C_i. Conversely, δ_d is the offset in chunk C_j (i.e. $v - a_j$). Note that the graph allows self-loops, since a pointer can point to the same chunk that it resides in. The displacement δ_s is called *pointer offset*.

The aforementioned Pointer Probing algorithm takes its name from the fact that chunks are treated as opaque blocks of memory, but P bytes sized blocks are interpreted as pointers and checked against the boundaries of existing chunks. As such, this algorithm may occasionally generate edges that do not actually represent pointers. This can happen if some datum has a value that, when interpreted as an address, coincidentally represents a valid memory address inside some chunk. This issue is resolved by running the graph construction on several separate executions of the program, and only accepting edges that appear in every execution. Generally, two runs are enough to eliminate false edges, however Light-PREM can be configured to perform more runs to increase the confidence about the validity of the recorded edges. Moreover, additional security measures can be adopted to prevent a faulty prefetch statement from causing an invalid dereference as mentioned in Section III-G.

2) Relative expressions: In the current Light-PREM C-based implementation, a relative expression for an observed memory access at location λ is a navigation path inside the memory layout graph that starts from a handle and ends in the chunk which contains λ. As shown below, any relative expression can be translated using only pointer arithmetic, pointer dereference operations, and name of handle variables. The key insight to relative expressions is that any location of an observed memory access inside one of the considered chunks can be reached using said expressions. For instance, a structure reference `some_struct->field` can be expressed as `*(some_struct + off)`, where `off` is the constant offset in bytes of `field` from the beginning of the structure.

Relative expressions reduce the complexity of Light-PREM and allow it to stay compiler independent, since the content of memory blocks is considered as opaque binary data.

3) Relative expression translation: Dereference chains are used to create relative expressions. To show how they can be translated to produce the final C-compliant relative expression, let us consider Figure 3 where, similarly to Figure 2, a sample memory layout is reported, but this time with explicit address and size values for the chunks. Specifically, suppose that the application under analysis allocates two blocks of memory prior to calling the target function, with sizes of 256 bytes and 4 KiB respectively.

These chunks get assigned starting addresses of `0xAAAA0000` and `0xBBBB0000` respectively. Furthermore, a function parameter named `handle1` is located at `0x11110000` and has value `0xAAAA0001`. Finally some pointers exist that link the three chunks together as depicted in the figure. The figure reports only the most significant 16 bits of each address.

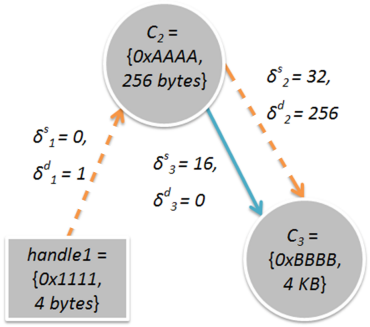

Fig. 3. Memory layout in a sample C application

Given the depicted memory layout, the final relative expression includes: (1) a prefetch statement of a given size; (2) a dereference chain which follows the dashed path highlighted in orange for some location λ at offset δ_λ in C_3. In our notation, it becomes:

$$\texttt{PREFETCH(handle1} \xrightarrow{\delta_1} C_2 \xrightarrow{\delta_2+\delta_\lambda} \lambda\texttt{, size)}$$

The resulting translation with trailing prefetch statement in the targeted language[3] is:

```
void * C_2 = * (void **)((void*) handle1 +0) -1;
void * C_3 = * (void **)(C_2 +32) -256;
PREFETCH(C_3 + delta_lambda, size);
```

Being able to generate relative expressions involves two steps. In the first, detailed in Section IV-B1 the interconnections between the recorded chunks lead to the construction of an oriented multigraph. In the second step, a reverse graph traversal strategy is employed to decide the optimal path through the vertices to connect a *handle* with a leaf memory access recorded during the data collection phase. This last step is detailed below.

4) Graph traversal strategy: After the construction of the memory graph, Light-PREM is ready to convert memory accesses (recorded in the very early stages of the process as absolute memory addresses) to relative expressions. To produce the relative expression, Light-PREM simply needs to find a path from a handle to the chunk that the memory access resides in. Finding such a path can be easily achieved by using existing graph traversal algorithms such as Depth First Search (DFS) and Breadth First Search (BFS). However, since the graph represents the memory organization of the application under analysis, additional traversal policy are employed to support the selection of the best path.

To understand the reason, let us consider again the example in Figure 3. To convert a memory access inside C_3, Light-PREM can use the dashed path highlighted in orange, and we have already seen what the correspondent relative expression looks like. However, when multiple paths exist, a simple depth or breath first search does not always yield the best results. In fact, due to the speculative nature of Light-PREM's analysis, there is no guarantee that a path, and consequently a relative expression, will be valid for all possible inputs to the program. To mitigate these effects, two heuristics are used to improve the performance of Light-PREM. The two heuristics that are used can then be summarized as: (1) requiring all pointer offsets to be non-negative; (2) given two pointer offsets for two edges linking the same chunks, selecting the smaller one.

The first heuristic (non-negative pointer offsets) comes from the fact that typical programs only utilize positive integers in pointer arithmetic. In fact, both array and structure references can be rewritten using non-negative pointer offsets.

The second heuristic arises from the fact that the smaller a pointer offset is, the more likely that the offset remains inside the bounds of the datatype the pointer refers to. In the example of a structure pointer, adding a small constant to the pointer is more likely to result in a pointer that is still inside the memory of the structure, as opposed to adding large offsets. A similar argument can be applied to arrays. To solidify this concept, consider the memory graph in Figure 4.

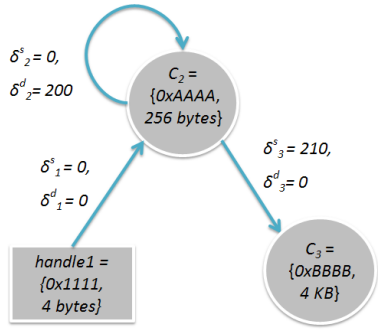

Fig. 4. Heap Memory Graph with a self-loop

In this graph, there exists a self loop. To travel from handle1 to C_3, it is preferable to take the self loop in C_2 on the way to C_3 as opposed to not taking it. To understand why, consider the relative expressions that these two different paths represent:

Path with self-loop:

$$\texttt{handle1} \xrightarrow{+0} C_2 \xrightarrow{+0} C_2 \xrightarrow{+10} C_3$$

Path without self-loop:

$$\texttt{handle1} \xrightarrow{+0} C_2 \xrightarrow{+210} C_3$$

In the path without the self-loop, the pointer offset 210 is likely to be dependent on the input of the executable. To see why this might be the case, let us consider an example scenario of the memory layout of the chunk represented by node 1. This scenario is depicted below in Figure 5

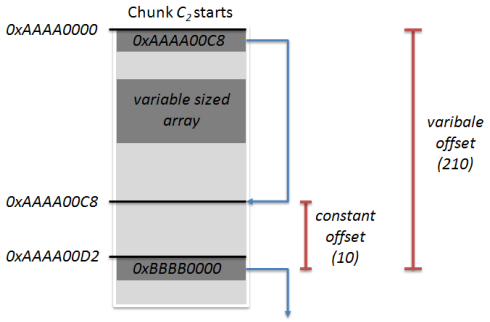

Fig. 5. Sample organization of virtual memory based on graph from Figure 4

In this scenario, there exists a variable sized array in the middle of the chunk, where the size of the array is dependent on the input of the executable or some other unknown factor. Thus the offset of 210 will change from run to run, whereas the offset of 10 will not. This means it is preferable to take the pointer (shown as an arrow in the diagram) from `0xAAAA0000` to `0xAAAA00C8` to skip over the variable sized array. This simple heuristic works very effectively.

With these two heuristics in mind, Light-PREM actually performs a reverse depth-first search. That is, it inverts the direction of all edges, and performs a depth-first search starting

[3]the C-like relative expression is simplified for the ease of the reader, however Light-PREM combines the three statements into one statement

at the chunk that the memory access resided in, stopping when it finally reaches a handle node. In line with the second heuristic, a path is allowed to go through the same node twice. The path exploration is bounded with a maximum exploration depth to break potentially infinite loops.

V. Evaluation

In this section we present the evaluation performed on Light-PREM. Three main metrics are used to understand the effectiveness of the proposed methodology: (A) *coverage* in terms of captured cache misses with respect to the manual PREM approach and to a non-PREM execution; (B) impact on *temporal predictability* of the execution phase of a task refactored with Light-PREM; (C) time needed to perform the Light-PREM *code refactoring*. Coverage analysis is carried out on a set of 7 EEMBC benchmarks that are reflective of a real-time scenario for their deterministic properties and code simplicity. To stress our implementation and understand the overhead of our technique, 7 additional and more complex benchmarks are used from the MiBench suite. The same set of EEMBC benchmarks is used to provide an idea of the time required for a full run of Light-PREM.

A. Testbed Setup

Since we want the evaluation to be to comparable with what presented in [1], we configured out testbed in a similar way. Specifically, we used an Intel Q9500 CPU in which we disabled the speculative CPU HW prefetcher since it negatively impacts the predictability of any real-time task. The Q9500 is a quad-core CPU and each pair of cores shares a common level 2 (last level) cache. Each cache is 16-associative with a total size of 6 MiB and a line size of 64 bytes. Since we use a PC platform running a COTS Linux operating system, there are many potential sources of timing noise, such as interrupts, kernel threads and other processes, which must be removed for our measurements to be meaningful. For this reason, in order to best emulate a typical uni-processor embedded real-time platform, we divided the 4 cores in two partitions. The system partition, running on the first pair of cores, receives all interrupts for non-critical devices (e.g., the keyboard) and runs all the system activities and non real-time processes (e.g., the shell we use to run the experiments). The real-time partition runs on the second pair of cores. One core in the real-time partition runs our real-time tasks. The other core is turned off. Note that the cores of the system partition can still produce a small amount of unscheduled bus and main memory accesses or raise rare inter-processor interrupts (IPI) that cannot be easily prevented. However, in our experiments, we found these sources of noise to be negligible.

Another source of unpredictability comes from the fact that the cache does not support deterministic allocation. However, our work can be easily integrated with practical solutions described in [4, 5, 9]. Thereby, we currently mitigate this problem by making sure that in our experiments the amount of prefetched memory is far less than the available cache (6 MiB). Self-evictions are also a minor concern since our platform uses an Intel SmartCache which does not perform a direct mapping from physical addresses to cache lines [10]. The cache is trashed before any measurement is taken in order to perform a worst-case oriented analysis.

Evaluating a task using Light-PREM requires 2 main steps: (A) the Light-PREM refactoring as discussed throughout the previous sections, and (B) the actual task run once Light-PREM has completed its refactoring. In this section, we will refer to A and B using the term *Analysis* and *Run* respectively.

B. Coverage Analysis

As previously stated, the main purpose of the PREM model is to enforce a structure in the memory access pattern of the applications so that all the memory accesses for a particular predictable interval are performed at the beginning, leaving the bus available for scheduling I/O flows or memory phases of tasks scheduled on other CPUs [6].

Manually performing such refactoring requires a deep understanding of the code and often requires non-trivial modifications to the code itself that can break correctness and optimizations. Light-PREM is able to perform this refactoring without changing the existing code and automatically inserting prefetch statements. Also, we have discussed how with Light-PREM we can relax the assumption that the code of the predictable interval is self-contained in a single function. However, the increased flexibility can have a cost. In particular, if the code performs calls to dynamically linked libraries or system calls inside the predictable function, Light-PREM will not be able to generate a corresponding prefetch for those memory accesses and more cache misses could be experienced during the execution phase. Thereby, the effectiveness of Light-PREM can be measured looking at how many cache misses[4] are being issued during the memory phase and how many are experienced during the execution phase. In this sense, the coverage is defined as the number of misses that are avoided in the execution phase thanks to the inserted prefetches. The desired property would be to have a perfect coverage for simple, self-contained predictable intervals and a sharp reduction of misses for complex code with calls to external functions.

We first analyze 7 benchmarks from the EEMBC automotive benchmark suite [11]. In past work [1], these benchmarks were modified to run multiple times, instead of just once, so that timing could be more accurately measured. The number of iterations needed to complete a pass on the input varies from benchmark to benchmark but this number is always below 10,000 iterations. Thus, 10,000 is the number of iterations used to perform the analysis in order to consume the input. Moreover, the code of the considered EEMBC benchmarks is simple enough so that the core of the computation is almost contained in a single function without having any system call in our candidate predictable interval.

Figure 6a shows the comparison between Light-PREM and manual PREM in terms of coverage for the mentioned benchmarks. The ratio between misses captured by Light-PREM and by the manual version with respect to the non-PREM case are reported in the first two bars of each cluster. The plot also reports the number of prefetches issued by Light-PREM as a fraction of prefetched inserted in the manual version in the last bar of each cluster. It is important to notice that in the manual PREM case, the code has been modified to (1) make indirect references explicit for the ease of writing prefetch statements and (2) inline all the functions called in the predictable interval. These modifications are not present in the code analyzed by Light-PREM, hence the slightly worse coverage observed.

Nonetheless, the results shown in the figure highlight that Light-PREM is always able to detect more than 92% of memory accesses (except for cacheb, where even the manual refactoring was not able to achieve a good coverage[5]). Two more behaviors can be noted. First, that, looking at the first pair of bars of each cluster, Light-PREM exhibits a coverage

[4]The number of cache misses is retrieved from the hardware performance counters relying on the `rdpmc` instruction.

[5]This is mainly because this particular benchmark performs function calls using function pointers inside the predictable function.

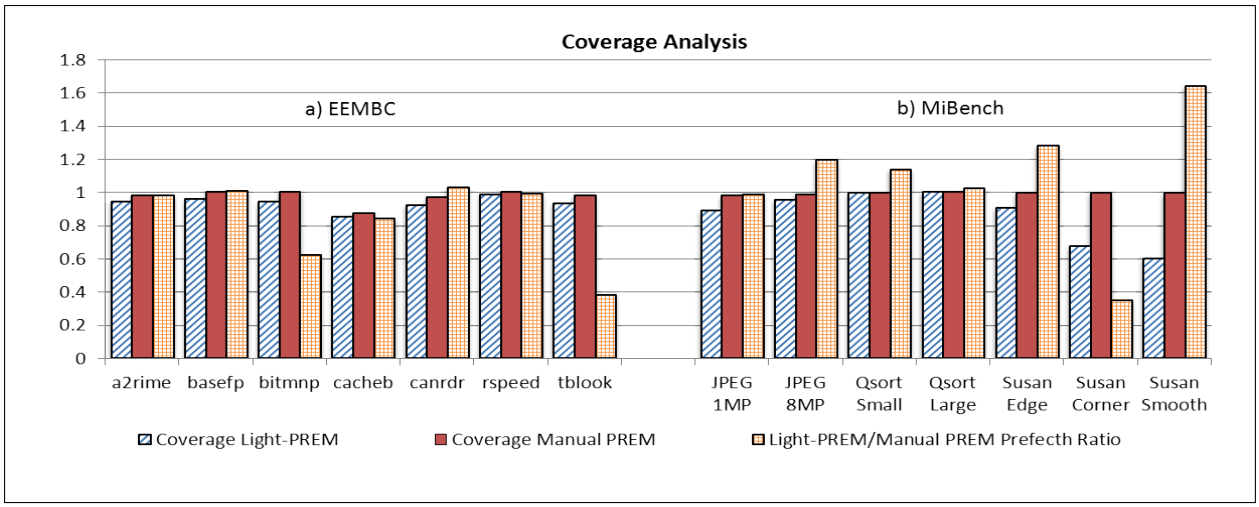

Fig. 6. Coverage measurements: ratio between misses captured by Light-PREM and manual PREM with respect to the misses observed in the non-PREM case (first two bars); Ratio between prefetches issued by Light-PREM and prefetches issued by manual PREM (third bar). Both EEMBC (a) and MiBench (b) are reported.

which is always comparable to what was achieved with a manual approach, even though it requires almost no changes to the source code in order to attain its goal. Second, that, according to the third bar of each cluster, the amount of prefetches issued by Light-PREM is always comparable to or much less than what results with a manual approach. This is because the generation of prefetch statements is driven by the observed memory accesses rather than reasoning about what portions of code/data can be reached/accessed.

The same evaluation approach has been used to understand the performance of Light-PREM on more complex code bases. Specifically, we ran our analysis on a set of applications from the MiBench suite [12]. In this case, the benchmarks have a much more complex code which brings them closer to consumer applications rather than real-time tasks. Nonetheless, it is interesting to understand the capabilities of Light-PREM to extract prefetch statements out of highly connected memory layouts.

Figure 6b shows the results obtained for: the Jpeg compression benchmark considering an input of 1 and 8 Mega Pixels; the Qsort vector sorting application; the Susan image processing benchmark that can be used to detect corners, edges in the inputted image or to smooth it. As reported in the table, Light-PREM achieves a high coverage for the Jpeg, Qsort benchmark and Susan with edge detection. In the last two tests, performance gets worse because the algorithms for corner detection and image smoothing, as they are, heavily rely on Libc library functions to perform memory and arithmetical operations. Accesses inside such memory areas cannot be captured in the current Light-PREM implementation.

Specifically, in the case of Susan with corner detection, the number of captured memory accesses is low. This results in a small number of produced prefetch statements and consequently to a low coverage (around 70%). Similarly, the smoothing algorithm of the Susan benchmark causes Light-PREM to build long dereference chains to reach some of the accesses in Libc areas, generating a large number of prefetches which still result in a low coverage. The problem could be solved by statically linking the Libc library[6], but this would affect the non-predictable part of the executable. Instead, as a part of our future work, we plan to include in Light-PREM a strategy to statically link only the subset of library functions used in the predictable interval.

[6]We have seen that doing so would boost the coverage of Light-PREM to 97%

C. Temporal Predictability

The aim of Light-PREM is to automatically refactor the targeted code without relying on compile-time knowledge in a way that the resulting application is compliant to the PREM model. It follows by the definition of PREM (PRedictable Execution Model) that predictability is one of the key features that has to be provided. An experimental way, to evaluate how predictable a fragment of code is, consists in running it a large number of times and observe how stable its timing properties are. Specifically, we are interested in understanding the timing properties of the *execution phase* in a predictable interval.

As shown in the previous section, Light-PREM is able to achieve a good coverage. This means that the number of bus accesses performed during the execution phase is sharply reduced. Thus, we argue that as a result the code executed inside the predictable interval exhibits better predictability. To evaluate this, bus activity is generated on the system partition of the testbed using an interfering task, which consists of a synthetic benchmark that generates high-bandwidth DRAM traffic.

The evaluation has been performed on the full set of benchmarks presented in the previous section. The last two Susan benchmarks are excluded from this analysis since Light-PREM does not achieve a good coverage and thus they are not representative for this experiment. Figure 7 reports average time duration, as well as variance, of the execution phase in the considered benchmarks. For each of them, the comparison is with the non-PREM (legacy) case.

As shown in the figure, the prefetches inserted by Light-PREM have two effects. First they sharply decrease the average execution time. Second they contribute to an improvement for the observed variance in the execution time of the considered code fragment. This demonstrates the effectiveness of Light-PREM as a way to increase the predictability in terms of timing behavior, as well as memory access pattern, in line with the advantages of the PREM model and subject to its limitations, as discussed in [1].

D. Analysis Runtime

Light-PREM involves performing several computation steps to carry on a complete analysis of the targeted executable.

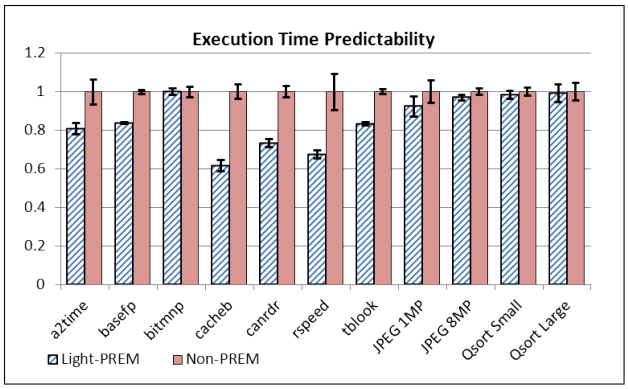

Fig. 7. Exec. phase runtime and variance for Light-PREM and non-PREM.

Some of these steps involve running the actual code to collect memory accesses and chunks, while other perform post-processing on acquired data.

As previously mentioned, memory access collection is performed using Valgrind which instruments the profiled code at runtime. This intuitively means that there is an expected overhead when running the observed code in the profiling environment. It is worth to mention that in earlier implementations of our technique, the analysis runtime could easily take hours even for simple benchmarks like a2time. In the latest version, however, the running time has been sharply reduced to the order of tens of minutes. The main improvements involved (1) patching Valgrind to analyze memory accesses for a selected fragment of code only and (2) postponing the actual data processing after the data collection phase has complete to avoid unnecessary slowdown.

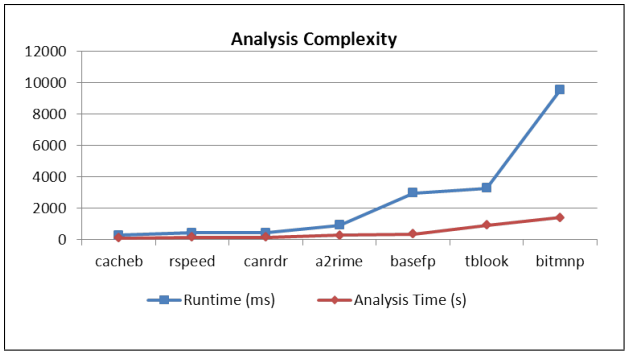

Fig. 8. Trend of time required to perform the Light-PREM analysis (in seconds) compared to the execution time of the benchmark (in milliseconds). Benchmarks ordered by runtime.

Figure 8 shows the ratio between the time required to perform the analysis of a given task and the runtime of the predictable interval of the task itself. This gives an idea of how the analysis technique scales as the complexity of the profiled application increases. It can be noted that, although the trend is linear, it can require minutes to analyze predictable intervals with execution time in the order of seconds. However, since the Light-PREM analysis and refactoring need to be done offline only once per task, the incurred cost not only does not represent a limitation to its applicability, but it is also a significant improvement considering that the time needed to perform manual refactoring could easily take days.

VI. Related Work

Several strategies ave been proposed and studied in literature for the automatic generation of prefetch statements for general purpose applications. The main reason why prefetching is employed is to issue in advance memory transaction that are known (or very likely) to occur at execution time. In this way, the data is brought from DRAM to cache before it is required, so that a cache miss is not suffered when the prefetched data is finally accessed for computation purposes [13]. In other words, prefetching works as a technique to perform execution speedup by masking processor stalls due to DRAM latency. Two main approaches have been considered. First, hardware based techniques have been proposed, which require architectural modifications to support platform-assisted prefetching. Second, strategies that are based on software mechanisms do not require any hardware modification and can be applied at run-time or at compile-time.

In [13] Callahan et al. demonstrate how prefetch instructions inserted at compile time can lead to a significant reduction of cache misses. Following this work, software strategies aimed at improving prefetching efficiency. In [14] three algorithms are proposed to perform efficient prefetching of array-based structures in scientific applications. Nonetheless, in general purpose applications, array-based structures coexist with pointer-based ones and objects. The extension of such algorithm that exploits multiprocessing to mask memory latency is detailed in [15]. In [16, 17] Wu et al. propose a more general algorithm that is able to generate prefetch sequences by analyzing stride memory access patterns to handle both array-based and pointer-based structures. Efficient compile-time strategies to automatically insert prefetch statements, when pointer-based and recursive structures are in use, are studied in [18, 19]. As a further step, Inagaki et al. in [20] developed a strategy that leverages on partial interpretation of a method that can be performed in a dynamic compiler to detect stride access patterns in object-oriented code. Similarly to Light-PREM some prefetching techniques rely on profile data [21, 22]. Specifically, Luk et al. in [21] use profiling to drive the detection of stride addressing patterns. In [22], profiling is used in a preliminary phase to extract recurrent sequences of memory references, while prefetching code is inserted subsequently in a dynamic fashion.

Abstract Interpretation (AI) to perform static analysis tools have been studied in [23, 24]. AI can be employed to understand the control flow behavior of a task, together with its memory usage profile. However, since they performs a partial interpretation of the code semantics, AI tools are unable to detect final addresses of memory accesses that can only be discovered at runtime. Moreover, in the general case, AI tools cannot detect the presence of links between areas of memory since it would require solving the undecidable problem of determining the complete set of reachable states of a program.

Techniques that involve modifications to the platform or that rely on specific architectural hardware support are classified as hardware-based techniques. Noticeably, in [25] a minimal set of hardware changes are proposed to run a prefetching thread on a separate processor located on the memory controller or the DRAM chip. The purpose is to keep a correlation table of the memory accesses and to run a customizable prefetching algorithm. Similarly, in [26] substantial changes are proposed to the memory and coherency controller. The key idea is to enable the execution of a restricted set of instructions used to deploy complex prefetching strategies performed directly on the memory hierarchy. Other prefetching strategies rely on hardware performance counters [27, 28]. In particular, in [27] prefetches are injected in the binary execution flow, while in [28] inefficient prefetch statements are filtered out according to run time data collected from performance counters.

It is important to underline that even if Light-PREM falls into the category of software techniques to generate prefetch

statements, its purpose is remarkably different from what proposed in literature. Whereas past research has used memory prefetching to increase performance (and thus improve the average case execution times), the PREM model uses prefetching to eliminate contention to memory, thus aiming at a reduction of worst case execution times. Moreover, most memory prefetching strategies uses compiler techniques, whereas Light-PREM aims to remain compiler-independent. Also, Light-PREM places prefetch statements at the beginning of an execution interval (predictable function), whereas existing memory prefetching research places prefetch statements throughout the execution blocks. Due to these major differences, Light-PREMs prefetching strategy manages to perform a predictability-oriented refactoring, while remaining practically applicable for industrial applications.

VII. Conclusion

Enforcing the PREM model on a group of real-time tasks allows performing a high-level coarse grained scheduling of shared hardware resources. In this way it becomes possible to regulate the contention for accessing such resources, determining an overall improvement of the predictability of the system.

In this work, we have presented Light-PREM: a software technique to automatically generate the prefetch statements that are needed to perform a legacy-to-PREM porting of a given executable. Light-PREM leverages on profiling and memory analysis strategies to produce a PREM-compliant code in an automatic fashion and without almost any knowledge about the semantics of the ported code. We have tested the proposed technique on a real testbed. The presented results effectively show that Light-PREM is able to achieve performances that are comparable (and in some cases superior) to the manual approach.

As a part of our future work, we plan to integrate Light-PREM with the Colored Lockdown presented in [5] in order to guarantee the persistence in cache of prefetched data.

Acknowledgment

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS-1302563, and CNS-1219064. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for COTS-based embedded systems. In *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, pages 269–279, Washington, DC, USA, 2011. IEEE Computer Society.

[2] S. Bak, E. Betti, R. Pellizzoni, M Caccamo, and L. Sha. Real-time I/O management system with virtualized COTS peripherals. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, pages 193–203, Washington, DC, USA, 2009. IEEE Computer Society.

[3] A. Arnaud and I. Puaut. Dynamic instruction cache locking in hard real-time systems. In *Proc. of the 14th International Conference on Real-Time and Network Systems (RNTS)*, Poitiers, France, May 2006.

[4] J. Liedtke, H. Haertig, and M. Hohmuth. OS-Controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, RTAS '97, pages 213–, Washington, DC, USA, 1997. IEEE Computer Society.

[5] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS'13, pages 45–54, Philadelphia, PA, USA, April 2013. IEEE Computer Society.

[6] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. In *Real-Time Systems*. Springer, 2011.

[7] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis &transformation. In *Code Generation and Optimization, CGO 2004. International Symposium on*, pages 75–86, march 2004.

[8] E. Youngdale. Kernel Korner: The elf object file format: Introduction. *Linux J.*, 1995(12es), April 1995.

[9] T. Liu, M. Li, and C. J. Xue. Instruction cache locking for multi-task real-time embedded systems. *Real-Time Syst.*, 48(2):166–197, March 2012.

[10] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 191–205, Washington, DC, USA, 2013. IEEE Computer Society.

[11] EEMBC benchmarkig suite. http://www.eembc.org/, 2012.

[12] MiBench benchmarking suite. http://www.eecs.umich.edu/mibench/, 2002.

[13] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. *SIGPLAN Not.*, 26(4):40–52, April 1991.

[14] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *SIGPLAN Not.*, 27(9):62–73, September 1992.

[15] T. C. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Trans. Comput. Syst.*, 16(1):55–92, February 1998.

[16] Y. Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. *SIGPLAN Not.*, 37(5):210–221, May 2002.

[17] Y. Wu, M. J. Serrano, R. Krishnaiyer, W. Li, and J. Fang. Value-profile guided stride prefetching for irregular code. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 307–324, London, UK, UK, 2002. Springer-Verlag.

[18] C. K. Luk and T. C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers*, 48, 1999.

[19] C. K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. *SIGPLAN Not.*, 31(9):222–233, September 1996.

[20] T. Inagaki, T. Onodera, H. Komatsu, and T. Nakatani. Stride prefetching by dynamically inspecting objects. *SIGPLAN Not.*, 38(5):269–277, May 2003.

[21] C. K. Luk, R. Muth, H. Patil, R. Weiss, P. G. Lowney, and R. Cohn. Profile-guided post-link stride prefetching. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 167–178, New York, NY, USA, 2002. ACM.

[22] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 199–209, New York, NY, USA, 2002. ACM.

[23] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise wcet determination for a real-life processor. In ThomasA. Henzinger and ChristophM. Kirsch, editors, *Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer Berlin Heidelberg, 2001.

[24] R. Heckmann and C. Ferdinand. Verifying safety-critical timing and memory-usage properties of embedded software by abstract interpretation. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 618–619 Vol. 1, 2005.

[25] Y. Solihin, J. Lee, and J. Torrellas. Correlation prefetching with a user-level memory thread. *IEEE Trans. Parallel Distrib. Syst.*, 14(6):563–580, June 2003.

[26] Z. Fang, L. Zhang, J. B. Carter, S. A. Mckee, A. Ibrahim, M. A. Parker, and X. Jiang. Active memory controller. *J. Supercomput.*, 62(1):510–549, October 2012.

[27] A. R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. *SIGPLAN Not.*, 39(6):267–276, June 2004.

[28] O. Gamoudi, N. Drach, and K. Heydemann. Using runtime activity to dynamically filter out inefficient data prefetches. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 338–350, Berlin, Heidelberg, 2011. Springer-Verlag.