Predictable Binary Code Cache: A First Step Towards Reconciling Predictability and Just-In-Time Compilation

Adnan Bouakaz, Isabelle Puaut, Erven Rohou

Predictable Binary Code Cache: A First Step Towards Reconciling Predictability and Just-In-Time Compilation

Adnan Bouakaz
University of Rennes 1 / IRISA
Rennes, France
adnan.bouakaz@irisa.fr

Isabelle Puaut
University of Rennes 1 / IRISA
Rennes, France
isabelle.puaut@irisa.fr

Erven Rohou
INRIA Rennes – Bretagne Atlantique
Rennes, France – HiPEAC member
erven.rohou@inria.fr

***Abstract*—Virtualization and just-in-time (JIT) compilation have become important paradigms in computer science to address application portability issues without deteriorating average-case performance. Unfortunately, JIT compilation raises predictability issues, which currently hinder its dissemination in real-time applications. Our work aims at reconciling the two domains, i.e. taking advantage of the portability and performance provided by JIT compilation, while providing predictability guarantees. As a first step towards this ambitious goal, we study two structures of code caches and demonstrate their predictability. On the one hand, the studied binary code caches avoid too frequent function recompilations, providing good average-case performance. On the other hand, and more importantly for the system determinism, we show that the behavior of the code cache is predictable: a safe upper bound of the number of function recompilations can be computed, enabling the verification of timing constraints. Experimental results show that fixing function addresses in the binary cache ahead of time results in tighter Worst Case Execution Times (WCETs) than organizing the binary code cache in fixed-size blocks replaced using a Least Recently Used (LRU) policy.**

Keywords: Virtualization, Just-in-time (JIT) compilation, Worst Case Execution Time (WCET) estimation, static analysis, binary code cache.

I. Introduction

The productivity of embedded software development is limited by the growing diversity of hardware platforms. This trend is driven by many reasons. The main forces at play include economics and technology. Time-to-market and price pressure favor designs that integrate off-the-shelf components over complete redesigns. Process variability in the most recent technologies increases the probability that, for a given product, parts of a component are defective and must be turned off [1].

One way to address this heterogeneity is the introduction of virtualization [2]. Programs are no longer compiled to native code, but rather target a platform-neutral bytecode representation. A virtual machine is in charge of efficiently running the bytecode on actual hardware. Virtualization is a well-established technology that has become important in computer science research and products; virtual machines are used in a number of subdisciplines ranging from operating systems to processor architectures.

Traditional compilation flows consist in compiling program source code into binary objects that execute natively on a given processor. Processor virtualization splits that flow in two steps: the first step consists in compiling the program source code into a processor-independent bytecode representation, whereas the second step provides an execution platform that can run this bytecode on a given processor. This latter step is achieved either by a virtual machine interpreting the bytecode, or by a just-in-time (JIT) compiler translating the bytecodes into binary code at run-time for improved performance.

Many applications have real-time requirements. The success of real-time systems relies upon their capability of producing functionally correct results within defined timing constraints. To validate these constraints, most schedulability analysis methods require that the worst-case execution time (WCET) estimate of each task be known. The WCET of a task is the longest time it takes when it is considered in isolation. Sophisticated techniques are used in static WCET estimation techniques [3] to produce both safe and tight WCET estimates.

JIT compilers, because they perform many operations at run-time, raise predictability issues that hinder their dissemination in real-time applications. Our work aims at reconciling the two domains, i.e. benefit from the portability and performance provided by JIT compilation while providing predictability guarantees. This is an ambitious goal which requires introducing determinism in many currently non-deterministic features, e.g. bound the compilation time, provide deterministic data and code allocation schemes, provide a predictable binary code cache. This paper is a first step towards the use of JIT compilation in real-time systems: we study two structures of binary code caches and compare their predictability. On the one hand, the studied binary code caches avoid too frequent function recompilations, providing good average-case performance. On the other hand, and more importantly for the system determinism, we show that the behavior of the code cache is predictable, in the sense that one is able to safely predict, upon every function call or return, if the target function is resident in cache. This lets us compute an upper bound of the number of function recompilations, which can be incorporated to compute functions' WCET and thus verify if the overall system meets its timing constraints.

The contributions of our paper are threefold:

- Firstly, we study two structures of binary code caches that

speed-up program *average-case performance*, for systems using JIT compilation;
- Secondly, we propose a static analysis method for each cache structure, that provides safe upper bounds of the number of function re-compilations, thus demonstrating the code caches *predictability*;
- Lastly, experimental results compare the worst-case performance of the two cache structures, and give insight on the parameters having an impact on their behavior.

While there is a wealth of literature concentrating on the worst-case behavior of hardware caches, to the best of our knowledge, our work is the first analysis of a binary code cache in the context of just-in-time compilation.

The rest of the paper is organized as follows. Section II surveys related work. Section III gives a more detailed specification of the problem addressed in the paper, as well as the underlying assumptions. Section IV presents two code cache structures and the static analysis methods, based on abstract interpretation, used to bound the number of function recompilations. Section V provides experimental results that demonstrate that the worst-case number of recompilations can be predicted in reasonable time, and analyzes the factors impacting the superiority of a code cache structure over another one. Section VI concludes the paper with a summary of the paper contributions, as well as future directions.

II. Related Work

Processor-independent bytecode representations (e.g. Java bytecode, CLI) ease the execution of applications on a wide range of hardware platforms. Two main directions can be followed to execute the bytecode *efficiently*. On the one hand, hardware support for bytecode execution can be used: entirely hardware-implemented virtual machines like JOP [4], that provides full support for the predictable execution of Java (time-bounded execution of Java bytecodes, hardware-managed object and method caches); Jazelle [5] moves interpretation into hardware for the most common simple Java bytecode instructions. On the other hand, JIT compilation may be used to generate binary code on-the-fly. Our work focuses on application portability at the bytecode level and relies on JIT compilation to achieve the same performance as classical compilers (referred to as ahead-of-time — AOT — compilers)[1].

Validating timing constraints requires that an upper bound of the Worst Case Execution Time (WCET) of every piece of software be known. Many WCET estimation methods and tools have been devised in the last two decades [3], mostly for code generated by AOT compilers.

Static WCET estimation methods need to work at a high-level to determine the structure of a program's task (so-called *high-level analysis*). They also work at a low-level, using timing information about the target hardware (*low-level analysis*). By combining those two kinds of analyses through a *calculation phase*, the WCET estimation tools give an upper bound on the time required to execute a given task on a given hardware platform. WCET analysis for interpreted bytecode was first considered in [7] for Java bytecode, concentrating on high-level analysis and WCET computation. In contrast to [7], we focus on JIT compilation rather than interpretation. The work [7] was extended to address the machine-dependent low-level timing analysis in [8]. This latter paper concentrates on bytecode timing information and on the support for pipelining effects and does not consider caches.

At the low-level, static WCET analysis is complicated by the presence of architectural features that improve the performance of the hardware. In particular, for general-purpose processors, many static analysis methods have been proposed in order to produce WCET estimates on architectures with hardware caches: instruction and data caches, cache hierarchies, support for multiple cache replacement policies [9], [10], [11], [12]. Related studies have focused on code positioning for minimizing respectively average-case execution times [13] and WCET estimates [13], [14]. The static analyses of the two binary caches presented in this paper are based on the same foundations as the analysis of hardware caches [9], the differences coming from the more complex cache structures that can be implemented in software, and the granularity of cache entries (entire functions vs. memory blocks).

WCET analysis for architectures with hardware support for bytecode execution is presented in [15], [16], [17] for the JOP Java processor. WCET estimation in JOP is simplified by the presence of time-predictable execution of instructions, the most complex architectural element to be analyzed being a hardware-implemented method cache. The method cache implements three management policies: a fixed-size 1-method cache, a fixed-size 2-method cache and a more complex *variable-method* cache organized in fixed-size blocks, using a FIFO replacement policy. [15] includes simple analysis methods for the two most simple cache structures, while [16] and [17] analyze the *variable-method* cache. Similar to our work, the static analysis of JOP method cache transposes the methods initially designed for general-purpose hardware caches. However, unlike [15], [16], [17], since our binary code cache is managed in software, more sophisticated cache structures and replacement policies can be designed and analyzed.

While many studies have been conducted for the timing analysis of real-time software, to the best of our knowledge, there is no attempt so far to provide predictability guarantees for systems using JIT compilation. Our work is a first step towards this ultimate objective.

III. Problem Statement

Reconciling predictability and JIT compilation is an ambitious goal. After a brief introduction to JIT compilation and its use of a code cache to avoid too frequent re-compilations (§ III-A and III-B), we present in § III-C the assumptions and restrictions made throughout the paper to focus on a specific predictability issue of JIT compilation: the management of the code cache.

[1]Note that JIT compilers can even surpass AOT compilers because they take advantage of run-time information, not available ahead of time, to apply focused optimizations [6].

A. JIT compilation

Just-in-time compilation is used to speedup the execution of bytecode compared to interpretation alone. The JIT compiler is in charge of translating the bytecode representation of the application into native code, ready to be executed. This translation occurs *just-in-time*, when a function is about to be executed, when no native code exists for it yet. Never executed functions are never compiled. Some systems compile all functions ahead of time [18], at the cost of an increased startup time. Some other systems have a policy to first interpret the bytecode to avoid the penalty of compiling a function that executes only a few times. In this case, the JIT compiler will run as soon as the virtual machine has detected a *hot* function [19]. Even more advanced systems have a policy in which functions are first interpreted; they are compiled when the execution frequency reaches a first threshold, but without applying any optimization to keep compilation time low; when a second threshold is reached, the function is compiled again at a higher optimization level. The compilation time overhead increases each time, but there is a hope to recoup it when running a faster version of the function many more times.

B. Code cache

The JIT compiler places the generated native code in a dedicated memory area called the *code cache*. This size of this area is limited by the RAM size. Similar to a hardware cache, when no room is available for a new entry, some other entries must first be evicted. Note that arbitrarily complex cache structures and replacement policies can be used since the cache is implemented in software. Some systems may even implement several techniques and choose the most appropriate one based on the results of the static analysis.

In the case of the binary code cache, cache entries are entire functions. Because they are of different sizes, memory fragmentation might be introduced.

C. Problem statement and assumptions

JIT compilation has many advantages regarding average-case performance as compared to bytecode interpretation. Its major drawback in a real-time environment is that it is a new source of non-determinism, since in the most complex systems, a given function may be compiled several times at unpredictable time instants. A function will necessarily be compiled the first time it is called (*cold* miss), but could also be recompiled in two other situations:

1) when the run-time system decides that a higher optimization level would be beneficial;
2) when the function has been evicted from the cache since the last time it was executed. This might occur because of a *capacity* miss (the function has been evicted because the code cache is full) or because of a *conflict* miss (a function with an overlapping address range has been loaded since the last function execution).

One of our contributions is to analyze the predictability of two code cache structures, i.e. the computation of the maximum number of times each function will be compiled. The determination of the worst-case number of compilations is based on static analysis of the application code, that detects at each call and return point if the access to the code cache will be a hit or a miss; the compilation time can then be added to the task WCET.

In order to concentrate on the binary code cache predictability, it is assumed that:

- all executed code is compiled, no code is interpreted;
- all code to be executed is statically known (no dynamic code loading);
- function sizes for the target architectures are known (measured or estimated);
- functions are compiled at function call or return points only, when absent from the code cache;
- there is only one level of optimization.

Moreover, since our focus is on code cache management, other sources of unpredictability raised by JIT compilers (predictability of compilation time, garbage collection, object allocation) are considered outside the scope of the paper.

IV. Code Cache Structures and Analysis

In this section, we first briefly review the concepts and notations used by our static analyses. We then describe the proposed cache structures and their analyses.

A. Background and notations

a) Abstract interpretation: Abstract interpretation (see [20] for details) is a formal method used to determine program properties statically (without program execution). Instead of representing properties in a *concrete domain* D (in our case, the actual code cache contents), abstract interpretation represents program properties in an *abstract domain* $\widehat{D}$. The relation between the concrete and the abstract domains are given by an abstraction function $\alpha : \mathcal{P}(D) \rightarrow \widehat{D}$ and a concretization function $\gamma : \widehat{D} \rightarrow \mathcal{P}(D)$, with $\mathcal{P}(D)$ the powerset of D.

The results of an abstract interpretation are invariants for each program point, which are represented by values in the abstract domain $\widehat{D}$. The analysis consists of a fixed-point computation on a set of data-flow equations operating on values in the abstract domain. A function $Update : \widehat{D} \times I \rightarrow \widehat{D}$ describes how an instruction $i \in I$ transforms the abstract value that existed before the instruction. When an instruction has multiple predecessors, a commutative and associative function $Join : \widehat{D} \times \widehat{D} \rightarrow \widehat{D}$ combines all incoming abstract values into a single one.

b) Abstract interpretation applied to a code cache: A binary code cache is a cache containing the compiled code of the functions likely to be reused in the future. What we wish to determine is whether or not the function's binary code is *guaranteed* to hit the code cache when a function call/return instruction occurs (*Must* analysis). In such a context, the concrete domain D is the actual contents of the binary code cache. The abstract domain $\widehat{D}$ is the domain of function identifiers $\mathcal{F}$ (f_i, f_j denote function identifiers). Upon the fixed point analysis convergence, all function calls and returns are assigned a classification: HIT if the function is in the abstract

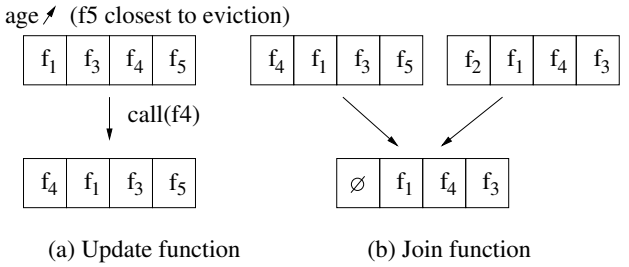

Fig. 1. Update and Join functions for fixed-size blocks with LRU replacement

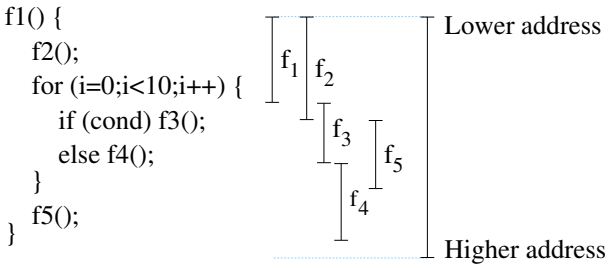

Fig. 2. Example of code fixed layout

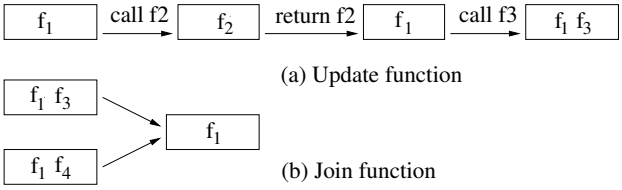

Fig. 3. Update and Join functions for the fixed layout cache structure

cache state (ACS), meaning that the function is guaranteed to hit the code cache, or MISS otherwise. The classification will be used to determine function WCETs using classical WCET estimation methods [3].

B. First structure: fixed-size blocks with LRU replacement

a) Cache structure: For this first structure, the code cache is decomposed in fixed-size blocks. The block size is equal to the size of the largest binary code of all program functions. The rationale behind the selection of a fixed size for cache blocks is to eliminate external fragmentation, possibly at the cost of increased internal fragmentation. The replacement policy for this cache structure is the LRU (*Least Recently Used*) replacement policy, selecting the least recently executed function in case of eviction. LRU is known to be the most amenable to accurate analysis [21].

b) Analysis: Analyzing such a cache structure is a straightforward extension of the analysis of fully associative hardware caches proposed in [9]. To determine whether a function is definitely in the cache, we use abstract cache states (ACS) where the position of a given function in the abstract cache state is an upper bound of the function *ages* in the LRU stack at run-time.

Fig. 1 depicts how the ACS are modified by the *Update* and *Join* functions in the abstract domain.

- The *Update* function updates the ACS when call and return instructions are found in the control flow graph (for all other instructions, *Update* is the identity function). The function simply updates the ages in the abstract cache state according to the LRU replacement policy. In the figure, when a call to f_4 is encountered, the ages of f_1, f_3 and f_5 in the ACS are updated accordingly.
- The *Join* function is applied whenever a block in the control flow graph has multiple predecessors. A function identifier f_i is present in the resulting ACS if it was present in both input ACSs. Its age is the maximum function age found in the input ACSs. In the example, f_5 is not guaranteed to be in the method cache, since only the left path compiles f_5. f_4 is definitely in the cache at the considered program point.

C. Second structure: fixed layout cache

a) Cache structure: In this second structure, every function is assigned a start address ahead of time. Start addresses are computed such that functions with heavy caller-callee relationships do not overlap in the code cache (see next paragraph for details). Upon a cache miss when calling, or returning to, a given function f, all functions that conflict with f are evicted from the cache, before f is compiled and its binary code inserted in the cache. An example of function layout is given in Fig. 2.

b) Analysis: The ACS for this second cache structure is the set of function identifiers for the functions guaranteed to be in the code cache at the considered program point. In contrast to the first cache structure, there is no concept of function age.

Functions *Update* and *Join* are defined as follow (see Fig. 3 for an example based on the code layout of Fig. 2):

- The *Update* function modifies ACSs in case of function calls and returns only. If the ACS already contains the function, it is left unmodified. Otherwise, the function identifiers of all conflicting functions are removed from the ACS, and the identifier of the called function is inserted. On the example, after f_1 starts execution, the ACS contains f_1. When f_2 is called within f_1, the ACS contains f_2 instead of f_1 since f_1 and f_2 conflict. When inside the loop f_1 calls f_3, both function identifiers are in the ACS because they are not in conflict with each other.
- The *Join* function is the intersection of the input ACSs. On the example, two ACSs are joined at the end of the conditional construct; only f_1 is guaranteed to be in the code cache at this point.

Layout computation

The second studied cache structure requires the address range of every functions to be computed. We developed two methods for that purpose.

a) Sequential (baseline): Functions are statically assigned an address range through a sequential scan of the functions according to their declaration order. A function is placed after the previously placed function if there is enough available room in the code cache, or at the start address of the cache otherwise. As an example, Fig. 2 corresponds to the

layout generated for an applications with five functions declared according to the order $f_1; f_3; f_4; f_2; f_5$. This first layout generation method ignores caller-callee relationships between functions and therefore will only be used as a baseline layout generation method to evaluate the WCET-directed approaches detailed hereafter.

b) WCET-directed heuristics: We defined two greedy WCET-directed layout computation heuristics, named respectively *WCEP* and *WCfreq*. They operate as follows:

- First, functions are ordered according to their potential for WCET reduction *weight(f)* (as shown later, the two WCET-directed variants differ by their definition of *weight(f)*).
- Functions are scanned in decreasing order of *weight(f)*. Each function f is placed by executing the algorithm shown below:

Algorithm 1 Start address generation for function f

```
Require: f: function to be placed
Require: PlacedF: list of already placed functions
 1: ListIntervals LI_f = [0,cacheSize[;
 2: sort PlacedF by decreasing conflict_cost(fi,f); with fi ∈ PlacedF
 3: for all fi in PlacedF do
 4:    ListIntervals NewLI_f = LI_f – [fi.start,fi.end[;
 5:    if (NewLI_f = ∅) then
 6:       break;
 7:    else
 8:       LI_f = NewLI_f;
 9:    end if
10: end for
11: f.start = first address in LI_f;
12: f.end = f.start + f.size;
```

In the algorithm, variable LI_f is the set of address intervals that minimize the conflicts between f and already placed functions (variable *PlacedF*). LI_f, initially set to the whole cache area (line 1), is progressively refined to avoid conflicts with already placed functions (loop in lines 3–10). Placed functions f_i are sorted by decreasing values of a cost function *conflict_cost(f_i, f)*, a large value meaning that a conflict between f_i and f should be avoided as much as possible. *conflict_cost(f_i, f_j)* is defined as $\frac{1}{dist(f_i,f_j)} * compilation\_time(f_j)$, with $dist(f_i, f_j)$ the distance between f_i and f_j in the program call graph considered as non-directed (in the example given in Fig. 2, $\forall i \neq 1, dist(f_1, f_i) = 1$, and $\forall i, j, i \neq j, dist(f_i, f_j) = 2$). $compilation\_time(f_j)$ is an upper bound of the compilation time of function f_j (see Section V).

The set of possible addresses for function f reduces (line 3) every time a conflict with a previously placed function is avoided. When all avoidable conflicts have been treated, the address range for function f can then be selected. The selected address range is the last non empty interval found which can contain f (lines 11 and 12).

TABLE I
ANALYZED TASKS

Name	num functions	total size	max size
Acquisition	157	44732	5052
Hit_ISR	71	16212	2448
Monitoring_Task	132	36476	5052
SU_Self_Test	139	38708	5052
TC_ISR	76	16668	2448
TM	96	28000	5052

The two WCET-oriented heuristics *WCEP* and *WCfreq* estimate the potential for WCET reduction of a function f through a function *weight(f) = freq(f)*compilation_time(f)*. In the first heuristic, *freq(f)* is the frequency of execution of function f along the worst-case execution path. The idea here is to optimize function layout to reduce the length of the critical path, with the risk that the critical path changes after layout generation[2]. In the second heuristic, *freq(f)* is the worst-case number of times function f can be executed in the program according to the program structure only (i.e. N if a call to f is embedded in a loop that iterates at most N times, regardless of the control flow within the loop). The rationale behind this heuristic is to avoid the problem of unstable critical paths of the first heuristic while still considering function execution frequency during layout generation.

V. Experimental Evaluation

A. Experimental setup

We experimented our technique on the Debie software [22] developed by European Space Agency, that monitors space debris and micrometeoroids *in situ* by detecting impacts using mechanical and electrical sensors. The program contains periodic and sporadic tasks. Table I lists the characteristics of the analyzed tasks. The second column shows the number of functions of the task, the third and fourth column respectively show the total size of the benchmark, and the size of the largest function. Some functions are shared between tasks (hence the repeated maximum size across rows).

The performance metric used to compare the two proposed cache structures is the WCET of every task. We estimate it using the state-of-the-art static WCET technique IPET [23] (Implicit Path Enumeration Technique). In IPET, program flow and basic-block execution time bounds are combined into sets of arithmetic constraints. Each basic block and edge in the task control flow graph is given a time value (t_i), expressing the upper bound of the contribution of that entity to the total execution time, and a count variable (f_i), corresponding to the number of times the entity is executed. An upper bound is determined by maximizing the sum of products of the execution counts and times ($\Sigma f_i \times t_i$), where the execution count variables are subject to constraints reflecting the structure of the task and possible flows. The result of an

[2]Critical path changes could be handled by re-evaluating the critical path in the course of the algorithm execution, at the cost of longer execution time for layout generation. We show in Section V that for the codes we have analyzed, management of critical path instability does not deserve the cost of an iterative solution.

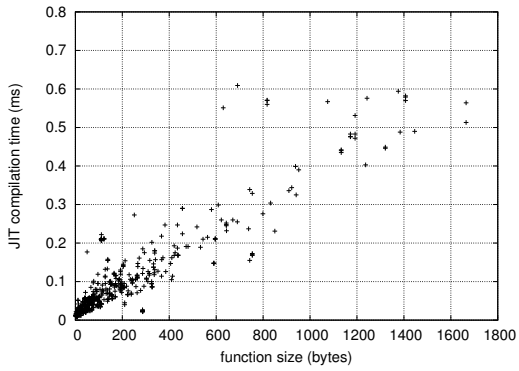

Fig. 4. JIT compilation time vs. function size

IPET calculation is an upper bound of WCET of the analyzed program. IPET also provides a worst-case count f_i for each basic block, thus identifying the critical path. We used the SCIP ILP solver [24] for WCET computation.

Time values t_i for basic blocks are computed as follows. First, for the sake of simplicity, we assume that each bytecode instruction takes one cycle to execute. In particular, hardware cache effects are not taken into account (hardware caches have an orthogonal impact and can be handled by extensions of classical techniques like [9]). Second, when a call or return instruction is found in a basic block, the analyses detailed in Section IV classify it as a hit or a miss. In the latter case, a compilation time is added to the time value of the basic block containing the call/return instruction.

We opted for a compilation time of the form $a \times x + b$, where x is the size of the function in bytes, and a and b are two constants characterizing the JIT compiler. The rationale behind this formula is that compile time consists in a startup time and compilation speed (a and b respectively). JIT compilers run under severe constraints and cannot afford any non linear algorithm[3]. We also confirmed this behavior by measuring the compile time of the open source JIT compiler Mono [26] for a wide range of functions. Fig.4 depicts the variation of compilation time (in milliseconds) with the function size (in bytes) for over 600 functions taken from the Mediabench suite [27], on a 3.07 GHz Intel Xeon. Note that the objective of our present work is not determining the WCET of the JIT compiler itself, rather the behavior of the code cache. Hence, these numbers are only meant to inject realistic numbers in our model, and to determine realistic worst-case execution paths.

B. Results

Experimental results are presented in Fig. 5 and Fig. 6. For every analyzed task, the figures give the task's WCET for varying cache sizes (in bytes). We observed that the fixed-layout heuristics *WCEP* and *WCfreq* give identical results in all cases, they are reported only once under the name *FL-heuristic*. For the six tasks we analyzed, the problem of unstable critical path we anticipated in Section IV has no impact on the heuristics' results.

Fig. 5 illustrates all the proposed structures: fixed-size blocks with LRU (*FSB-LRU*), fixed-layout sequential (*FL-seq*) and *FL-heuristic*. Fig. 6 does not show *FSB-LRU* and focuses on a reduced range of cache sizes to magnify lower-order phenomena. It also shows a second version of each analysis in which each outermost loop has its first iteration peeled[4]. Finally, it depicts the WCET of every task assuming a *perfect* code cache (the compilation time of every function is accounted for exactly once).

a) General observations: As expected, the WCET globally decreases when the cache size increases. Nevertheless, there are some irregularities in the behavior of the layout determination heuristics. These glitches are explained by threshold effects: to lower the algorithmic complexity, the heuristics approximate conflict costs between functions. At the same time, functions cannot be split in the code cache. A single byte difference in the cache size may relocate a number of functions and have drastic effects. This is especially true at small cache sizes.

We observe that for all benchmarks and most cache sizes, the *FSB-LRU* structure performs worse (i.e. yields higher WCET) than the fixed-layout structure. This comes from two factors:

- *Memory fragmentation*: the block size of *FSB-LRU* is the size of the largest function, thus introducing potentially large internal fragmentation in blocks when function sizes are very heterogeneous;
- *Replacement strategy*: in the *FSB-LRU* structure, the cache replacement policy (LRU) is fixed and independent of the application call patterns. In contrast, *FL-heuristic* structures compute the layout ahead of time based on some knowledge of the application. They have an opportunity to adapt the cache replacement to the application call pattern.

Having a closer look at the behavior of the *FSB-LRU* structure, we notice its very poor worst-case performance when the cache is small. Recall from Table I that, for example, the largest function (and hence the cache block size) of *Acquisition* is 5052 bytes. As long as the cache is smaller than $2 \times 5052 = 10104$ bytes, it cannot accommodate more than one block. All calls/returns are classified as MISS. In such cases, the fixed-layout structures perform much better because the memory fragmentation is then much lower.

b) Comparing heuristics: In Fig. 6, we observe that *FL-heuristic* provides improvements over the baseline *FL-seq* for small cache sizes. For large sizes, the two layout generation methods behave identically, because the cache is large enough to store all functions after they are first compiled.

[3]Special algorithms are developed when standard techniques are too expensive — for example the linear scan register allocator [25].

[4]Loops are *virtually* peeled, for analysis purposes only, no physical loop peeling is undertaken.

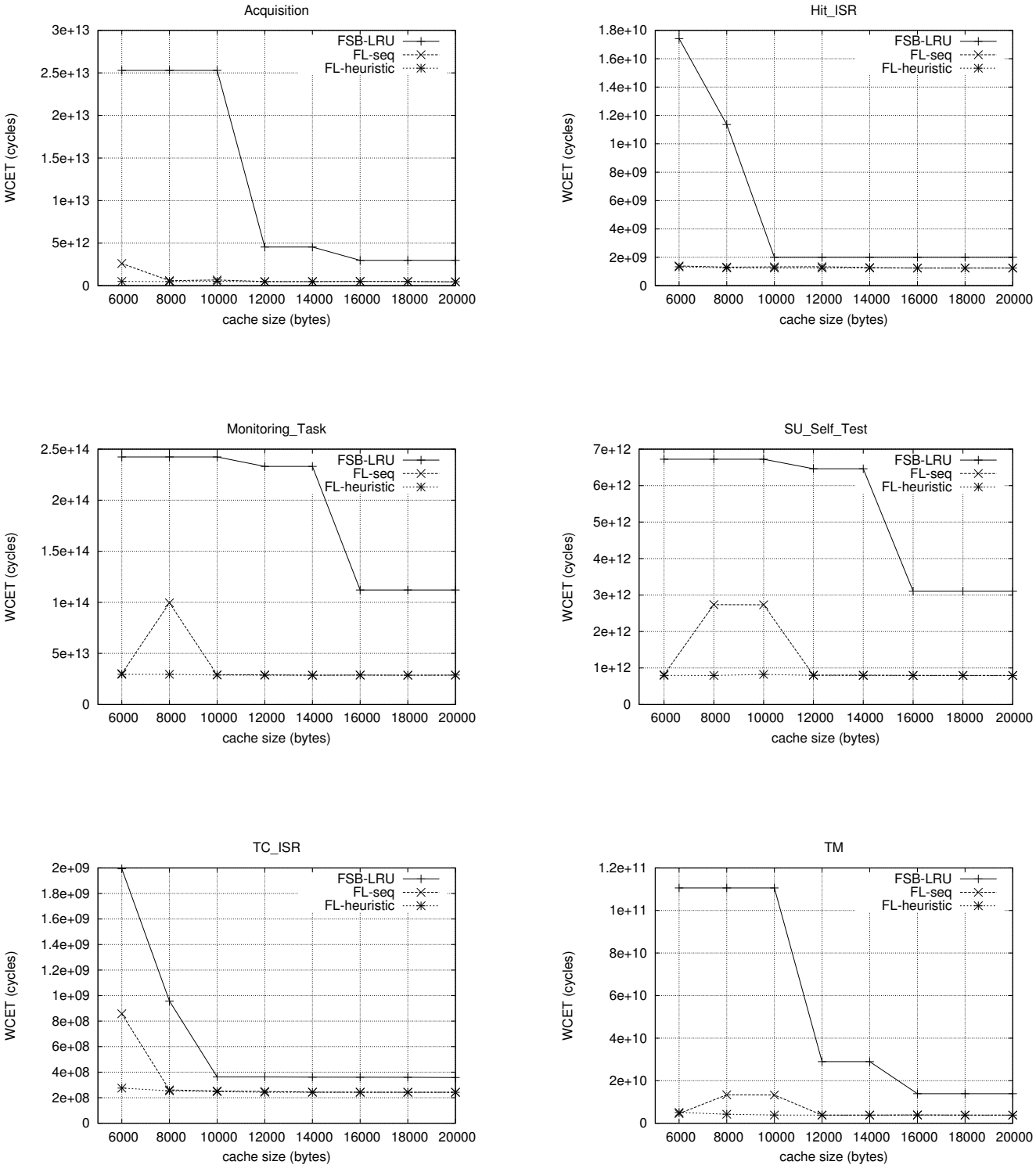

Fig. 5. WCET (in cycles) for each cache structure

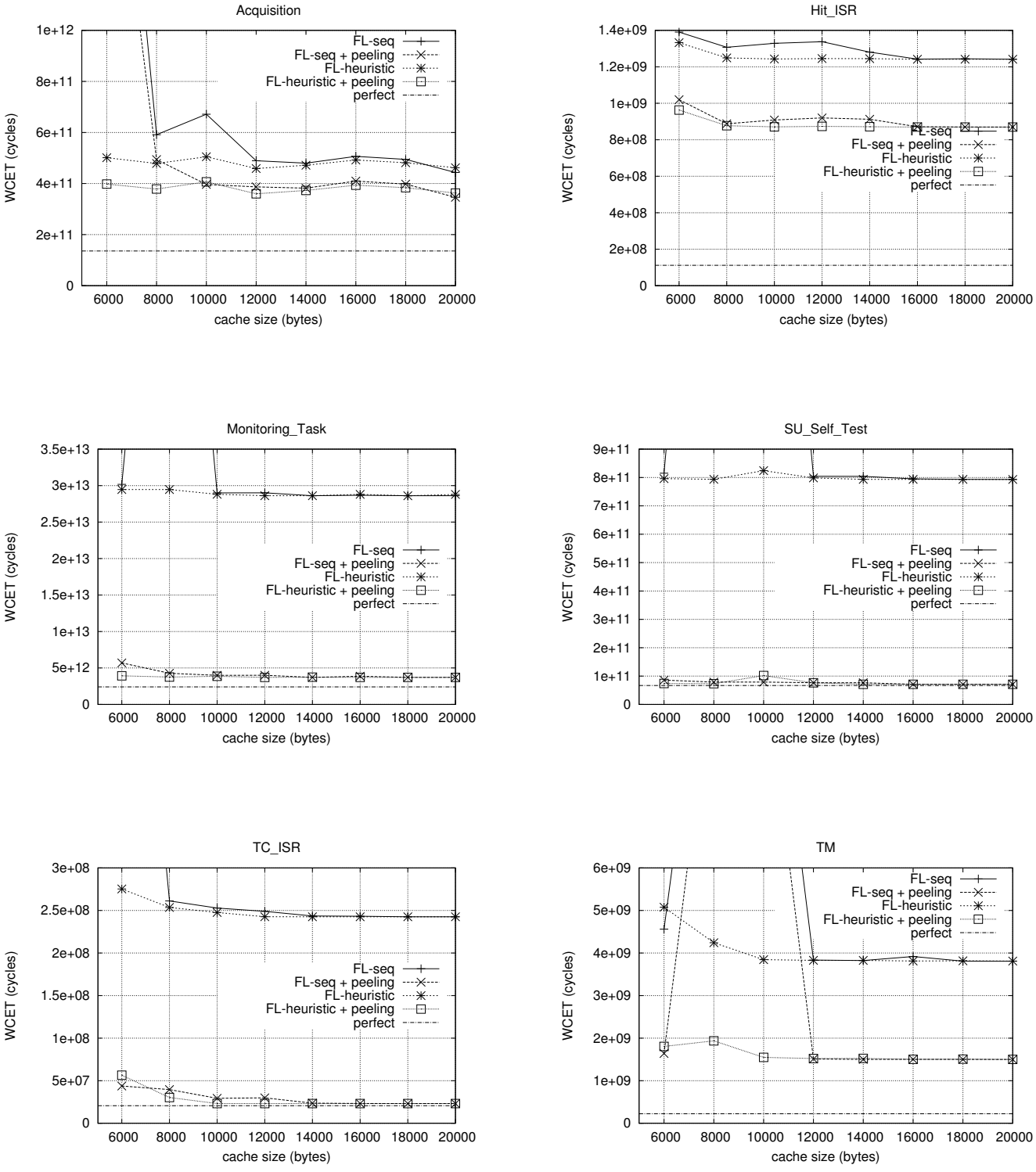

Fig. 6. WCET (in cycles) for fixed-layout cache and perfect cache (close-up)

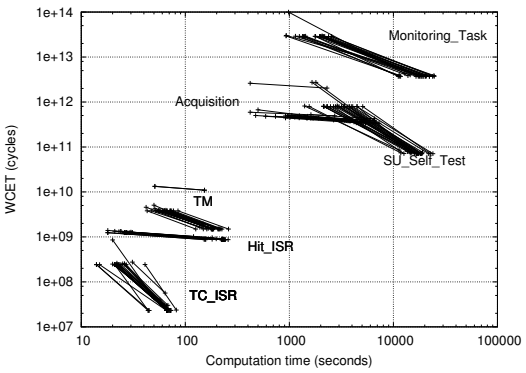

Fig. 7. WCET estimation vs. computation time (layout generation + fixed-point analysis)

However, for 3 out of 6 tasks, even a large cache does not result in a WCET comparable with a perfect cache. This is due to the presence of conditional constructs within loops, like in the example of Fig. 2 (calls to functions f_3 and f_4). Even if the loop is peeled (see below), none of the functions called in the conditional construct can be guaranteed to be in the cache after the first iteration, and thus both functions are classified as MISS. This phenomenon, well known when analyzing hardware caches, has a deeper impact on our analysis because of the granularity of cache entries.

c) Loop peeling: Loop peeling improves the analysis tightness by detecting that a function, called during the first iteration of the loop, will be resident in the following iterations. All tasks benefit from peeling, but its effectiveness clearly depends on the tasks code structure (number of function calls within outer loops). For instance, the improvement for *Acquisition* is lower than for the other tasks, because there are fewer function calls in loops.

Improved tightness is obtained at the cost of higher computation time. Fig. 7 illustrates this trade-off: all WCET data points of Fig. 6 are represented with their computation time (in seconds). The left-most part of segments represent analyses without peeling (faster to compute, looser WCET), the right-most part represents the same analysis with peeling (slower, but tighter WCET). The six clusters denote the six tasks. We observe that peeling always improves the WCET, sometimes by more than an order of magnitude. Computation time, however, can also increase significantly.

VI. Conclusion

JIT compilation has become an important tool to address application portability issues without deteriorating average-case performance. Unfortunately, JIT compilation raises predictability issues, which currently hinders its dissemination in real-time applications. As a first step towards reconciling the performance and portability benefits provided by JIT compilation with predictability considerations, we have proposed and evaluated two structures of binary code caches. On the one hand, the binary code caches we propose avoid too frequent function recompilations, providing good average-case performance. On the other hand, and more importantly for the system determinism, we show that the behavior of the code cache is predictable, and that it is possible to compute safe upper bounds of the number of function recompilations. Experimental results show that fixing the function layout ahead of time yields lower WCETs than transposing to software some standard hardware cache structures (fixed block code cache with LRU replacement).

In the short term, our future work will consider more refined cache structures, i.e. function code splitting for the FSB-LRU cache structure, or multiple statically defined cache block sizes. Our longer term future work will further explore predictability issues raised by JIT compilers in real-time systems, among others: predictability of compilation times, memory management, and code optimization.

Acknowledgments

The authors would like to thank Benjamin Lesage, Damien Hardy and André Seznec, for fruitful brainstorming.

References

[1] M. Duranton, S. Yehia, B. De Sutter, K. De Bosschere, A. Cohen, B. Falsafi, G. Gaydadjiev, M. Katevenis, J. Maebe, H. Munk, N. Navarro, A. Ramírez, O. Temam, and M. Valero, *The HiPEAC Vision*. Network of Excellence of High Performance and Embedded Architecture and Compilation, 2010.

[2] C. Bertin, C. Guillon, and K. De Bosschere, "Compilation and virtualization in the HiPEAC vision," in *DAC*, 2010.

[3] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Muller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *TECS*, vol. 7, no. 3, pp. 1–53, Apr. 2008.

[4] C. Pitter and M. Schoeberl, "A real-time Java chip-multiprocessor," *TECS*, vol. 10, no. 1, 2010.

[5] ARM, "ARM Jazelle technology." [Online]. Available: http://www.arm.com/products/processors/technologies/jazelle.php

[6] A. Cohen and E. Rohou, "Processor virtualization and split compilation for heterogeneous multicore embedded systems," in *DAC*, 2010.

[7] G. Bernat, A. Burns, and A. J. Wellings, "Portable worst-case execution time analysis using Java byte code," in *ECRTS*, 2000, pp. 81–88.

[8] I. Bate, G. Bernat, G. Murphy, and P. P. Puschner, "Low-level analysis of a portable Java byte code WCET analysis framework," in *RTCSA*, 2000, pp. 39–46.

[9] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and precise WCET prediction by separated cache and path analyses," *Real-Time Systems*, vol. 18, no. 2-3, pp. 157–179, 2000.

[10] F. Mueller, "Timing analysis for instruction caches," *Real-Time Systems*, vol. 18, no. 2-3, pp. 217–247, 2000.

[11] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1038–1054, Jul. 2003.

[12] D. Hardy and I. Puaut, "WCET analysis of multi-level non-inclusive set-associative instruction caches," in *RTSS*, 2008, pp. 456–466.

[13] M. Garatti, "FICO: A fast instruction cache optimizer," in *SCOPES*, 2003, pp. 388–402.

[14] P. Lokuciejewski, H. Falk, and P. Marwedel, "WCET-driven cache-based procedure positioning optimizations," in *ECRTS*, 2008, pp. 321–330.

[15] T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad, "A modular worst-case execution time analysis tool for Java processors," in *RTAS*, 2008, pp. 47–57.

[16] R. Kirner and M. Schoeberl, "Modeling the function cache for worst-case execution time analysis," in *DAC*, 2007, pp. 471–476.

[17] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber, "Worst-case execution time analysis for a Java processor," *Software: Practice and Experience*, vol. 40/6, pp. 507–542, 2010.

[18] J. S. Auerbach, D. F. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart, and M. G. Stoodley, "Design and implementation of a comprehensive real-time java virtual machine," in *EMSOFT*, 2007, pp. 249–258.

[19] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the Java HotSpot™client compiler for Java 6," *TACO*, vol. 5, no. 1, pp. 7:1–7:32, 2008.

[20] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*. ACM, 1977, pp. 238–252.

[21] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Systems*, vol. 37, no. 2, pp. 99–122, 2007.

[22] N. Holsti, T. Långbacka, and S. Saarinen, "Using a worst-case execution time tool for real-time verification of the DEBIE software," in *Data Systems in Aerospace (DASIA)*, 2000.

[23] P. Puschner and A. V. Schedl, "Computing maximum task execution times – a graph based approach," in *RTSS*, vol. 13, 1997, pp. 67–91.

[24] T. Achterberg, "Constraint Integer Programming," Ph.D. dissertation, Technische Universität Berlin, 2007, http://opus.kobv.de/zib/volltexte/2009/1153/.

[25] O. Traub, G. Holloway, and M. D. Smith, "Quality and speed in linear-scan register allocation," in *PLDI*. ACM, 1998, pp. 142–151.

[26] "The Mono Project," `http://www.mono-project.com`.

[27] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *MICRO*, 1997, pp. 330–335.