Predictability of just in time compilation

Adnan Bouakaz

Predictability of just in time compilation

Adnan Bouakaz

Supervisors: Isabelle Puaut, Erven Rohou
IRISA, ALF project

Research Master's degree in Computer Science

June 2010

Contents

Introduction

The productivity of embedded software development is limited by the high fragmentation of hardware platforms. To alleviate this problem and to allow an efficient execution of applications on a large spectrum of hardware platforms, the computer science community has to invent a new form of application portability that will replace traditional binary compatibility. Virtualization has become an important tool in computer science; and virtual machines are used in a number of subdisciplines ranging from operating systems to processor architecture. The processor virtualization can be used to address the portability problem.

While the traditional compilation flow consists of compiling program source code into binary objects that can natively executed on a given processor, processor virtualization splits that flow in two parts: the first part consists of compiling the program source code into processor-independent bytecode representation; the second part provides an execution platform that can run this bytecode in a given processor. The second part is done by a virtual machine interpreting the bytecode or by just-in-time (JIT) compiling the bytecodes of a method at run-time in order to improve the execution performance.

Many applications feature real-time system 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 scheduling algorithms assume that the worst-case execution time (WCET) estimation of each task is already 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 (e.g. to model caches) to achieve both safe and tight estimation.

Our work aims at recombining the two domains, i.e. using the JIT compilation in real-time systems. This is an ambitious goal which requires introducing the deterministic in many non-deterministic features, e.g. bound the compilation time and the overhead caused by the dynamic management of the compiled code cache, etc. Due to the limited time of the internship, this report represents a first attempt to such combination. To obtain the WCET of a program, we have to add the compilation time to the execution time because the two phases are now mixed. Therefore, one needs to know statically how many times in the worst case a function will be compiled. It may be seemed a simple job, but if we consider a resource constraint as the limited memory size and the advanced techniques used in JIT compilation, things will be nasty. We suppose that a function is compiled at the first time it is used, and its compiled code is cached in limited size software cache. Our objective is to find an appropriate structure cache and replacement policy which reduce the overhead of compilation in the worst case. This will be a static analysis due to the safety factor as we will see later.

The rest of the document is organized as follows: Chapter 1 introduces the notions of JIT compilation and WCET estimation. In Chapter 2, we propose a solution to the abovementioned problem. Three cache management systems and their analyses are proposed in this chapter. Finally, some results of the analyses are presented in the third chapter.

Chapter 1

JIT Compilation & WCET Estimation

This chapter introduce two largely different domains. First, the benefits of *virtualization* and JIT compilation are listed in §1.1 and §1.2; then a brief survey on WCET estimation methods is presented in §1.3. We conclude the chapter by asking a relevant question: What is needed to recombine the two domains to exploit their respective advantages?

1.1 Virtual Machines

Although nowadays computer systems are very complex, they continue to evolve. They are designed as hierarchies of different levels of abstraction separated by well-defined interfaces. Each abstraction hides a large amount of implementation details. In [31], we can find examples of such abstractions. For instance, computer's instruction set architecture (ISA) is a nice example of the well-defined interfaces.

Unfortunately, subsystems and components designed for one interface will not work with those designed for another (i.e. an *interoperability* problem). Virtualization provides a way to getting around such problems. Virtualizing a system or component (such as a processor, memory ...) at a given abstraction level maps its interface and visible resources onto the interface and resources of an underlying, possibly different, real system. Virtualization goals are not necessarily those of abstraction, i.e. simplifying or hiding details. It also provides *Software compatibility*: the virtual machine (VM) provides a compatible abstraction so that all software written for it will run on it, and *Isolation*: the VM abstraction isolates the software running in the VM from other VMs and real machines. To implement a VM, developers add a software layer to a real machine to support the desired architecture. There are two kinds of virtual machines [31]:

- **System virtual machines:** A system VM provides a complete environment in which an operating system and many processes of multiple users can coexist (e.g. Virtual PC in which a Windows system runs on a Macintosh platform). So, many isolated guest operating system environments can run on the same host hardware platform.
- **Process virtual machines:** A process VM provides a virtual ABI (Application Binary Interface) or API (Application Programming Interface) environment for user applications. Process VMs can offer replication, emulation and optimization. A key objective for process VMs is the cross-platform portability. Full cross-platform portability may be achieved by designing a process VM as a part of an overall high-level language (HLL)

application development environment. The resulting HLL VM does not directly correspond to any real platform; rather, it is designed for ease of portability and to match the features of a given HLL or set of HLLs. In this report, we are interested in HLL VMs only.

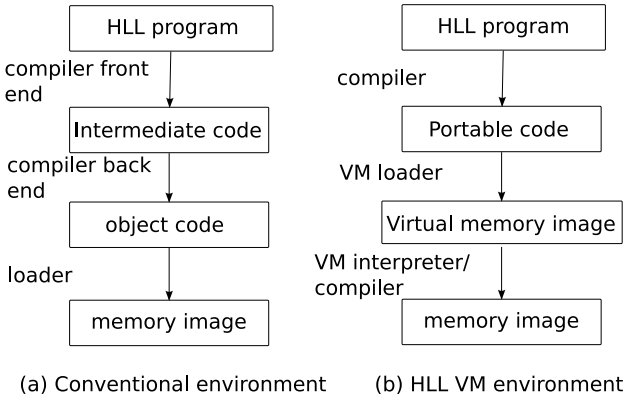


Figure 1.1: HLL environments

Figure 1.1 shows the difference between a conventional platform-specific compilation environment and an HLL VM environment. In a conventional system, a compiler front end generates intermediate code that is used by a code generator to generate a binary containing machine code for a specific ISA and operating system. The compiler front end of an HLL VM generates abstract machine code (called *bytecode*) in a virtual ISA that specifies the VM's interface. This code can be distributed for execution on different platforms (*portable* code). Each host platform must implement a VM able of loading and executing the virtual ISA.

As examples of HLL VMs, there are the Sun Microsystems Java VM architecture (JVM)[1] and the Microsoft Common Language Infrastructure (CLI)[2]. In both VMs, the ISAs are stack-based to eliminate register requirements and use an abstract data specification and memory model that supports secure object-oriented programming. There is not a big difference between the JVM and the Microsoft CLI specification, therefore Java is used as an example in the following sections.

1.1.1 Different Types of JVM Implementations

The specification of the JVM is detailed in [19] without introducing the implementation aspects. The early JREs (Java runtime environments) executed Java programs by interpreting the bytecodes. So, ignoring exceptions, the inner loop of the JVM execution is:

```
do {
        Fetches the next bytecode to execute and decodes it.
        If (operands) fetch operands.
        Execute the action for the opcode.
} while (there is more to do)
```

The advantage of this approach is its simplicity; and the drawback is its modest performance. Addressing the performance gap with languages such as C or C++ means developing

[1] http://java.sun.com/docs/books/jvms/
[2] http://www.ecma-international.org/publications/standards/Ecma-335.htm

native code compilation for the Java platform in such a way the portability is not sacrificed. The first technique to explore is the *Ahead* of time translation (AOT). The principle idea is to translate all program bytecodes to machine code before execution. A big challenge of such technique is the dynamic class loading. A class cannot be loaded until the executing code makes a reference to that class. Because AOT occurs before the program executes, the compiler cannot make any assumptions about which classes have been loaded.

Another technique is the just-in time (JIT) compilation in which the first time a method is called it is translated into object code (see §1.2). We can find in [21] a detailed comparison between the AOT and the JIT compilation techniques.

The fourth technique is the hardware VM: a processor that implements the JVM instruction set. As an example, the Java optimized processor (JOP) [30]. JOP is a stack computer with its own instruction set, called microcode. Java bytecodes are translated into microcode instructions or sequences of microcode in hardware. This technique removes the overhead of decoding instructions in software. It does not, however, address the problem of portable execution on existing processors.

1.2 JIT Compilation

Interpreting bytecodes is slow. Performing compilation prior to runtime and loading the program as native code would eliminate the portability and security of Java programs. On the other hand, compiling at runtime preserves these important properties. The program can still be distributed in platform-independent class files, and the bytecodes can still be verified prior to compilation. So unlike binary dynamic compilation, the process does not start with already compiled code but with bytecode. Both JIT and binary dynamic compilations have to manage a *compiled code cache*. There is a comparison between those two techniques in [8].

Unlike a static compiler, the compiling procedure is not entirely separated from the executing one in a JIT compiler. We can find in [36] a design of a Java JIT compiler. For a *mono-processor* architecture, the JVM includes four main components: class loader and linker, garbage collector, thread manager and a JIT compiler. For presentation clarity, we separate the JIT compiler from the JVM, so there are three states at runtime: JVM state, JIT compiling state and native code running state.

The state change in the JVM with a JIT compiler can be described as follows (Figure 1.2): at the beginning of the execution, the running state is in JVM. JVM loads and links the class file to be executed, and initializes the running environment. Before executing a method, JVM judges if the method has been translated into native code. If not, JVM calls the JIT compiler and the running state is changed into JIT compiling state. For translating a JVM method calling instruction, the JIT compiler must generate a native instruction which calls itself (a *trampoline*) to translate the Java method into native method. For some functions performed by the JVM (e.g. garbage collector), the JIT compiler should generate the native instructions which invoke them.

1.2.1 Challenges in JIT compilation

Although platform neutrality is maintained with JIT compilation, it comes at a price. Because compilation happens at the same time as program execution, the time it takes to compile code is added to the program's running time. Therefore, compilation speed is crucial. This is a very different situation from that facing a traditional, static compiler. To minimize overhead,

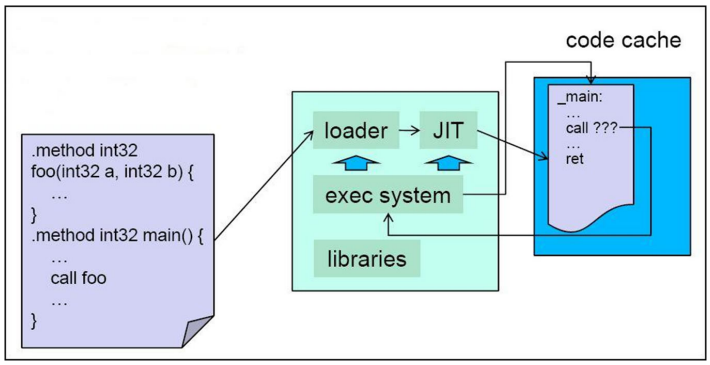


Figure 1.2: The JVM's state change

we can avoid compiling a method until it is certain that it will be executed. Compared to traditional static compilers, JIT compilers perform the same basic task of generating native code, but at much faster rate. The challenge for a JIT compiler is to find ways to generate efficient code without incurring the expense of traditional optimization techniques.

In many compilers, constructing an intermediate representation (IR) is a standard process. The bytecodes can be the IR for the JIT compiler. However, rather than treating bytecodes as literal descriptions of the code to be executed, a JIT can use them as implicit encodings of expressions. There are three major issues associated with transforming bytecodes directly into efficient machine instructions:

- The use of the operand stack constrains the order of evaluation specified by the bytecodes. More efficient orderings could be possible with a register-based processor.
- The best choice of machine instruction for a particular bytecode may depend on how a result of that bytecode is used. Indeed, some bytecodes are best translated to machine instructions in combination with the bytecodes that use their results.
- The bytecodes may include redundant operations.

More details about the JIT compilation process may be found in [7].

1.2.2 Optimization in JIT compilers

Current JIT compilers can substantially improve performance, and future JVM implementation will use more advanced techniques to realize further performance gains [7]:

Adaptive optimization

One would like to tailor how to compile a method to the amount of time the program actually spends in that method. A dynamic compiler can observe the program as it runs (*profiling*) and optimize the most frequently executed models. A simple example is deciding when to compile based on observed execution frequencies. Recompiling based on observed behavior is a form of adaptive optimization. In adaptively optimization system, initial executions of a method either are interpreted or use a simple fast compiler. The code is self-monitoring, using execution counters, to detect hot spots. When a method is found worthy of optimization, the system can spend more time on its compilation.

This approach uses two dynamic compilers, a fast non-optimizing compiler and a second optimizing compiler. The authors of [17] discuss the utility of off-line profile information to decide which compiler to use initially. They developed an annotation framework to communicate that information with the JIT compiler. Another work in [18] presents a study of whether side-effect information collected off-line improves performance in JIT compilers.

Adaptive inlining

Inlining methods (replacing calls with the actual code of the called methods) is an important optimization in any program with a high call density and small methods. The use of virtual calls in Java defeats traditional inlining techniques, because many target methods may exist for a virtual call. A dynamic compiler can inline even a virtual call with more potential targets. The runtime system can note call sites which invoke the same method repeatedly. The compiler can also emit specialized versions of the method.

1.3 WCET Estimation Methods

Unfortunately, HLL VMs are not widespread in real-time systems for a number of significant reasons (see §1.4) which mainly concern their unability of validating timing constraints. But, they remain a target solution for the problem of the fragmentation of platforms for embedded systems. Indeed, the bytecode is an effective deployment format for embedded systems and so for real-time system [4].

The success of real-time systems relies upon their capability of producing functionally correct results within defined timing constraints. A *hard* real-time system is a system where a failure in the temporal domain will cause the system to fail. A *soft* real-time system is a system that will fulfill its mission even if deadlines are missed occasionally, but it should in normal operation not miss any deadline.

In a hard real-time system, there must be a guarantee that it does not miss any deadline in all situations. In order to achieve this, it is of vital importance to know the Worst-Case Execution Time (WCET) of each task. Unfortunately, it is not possible, in general, to obtain upper bounds on execution times for programs, unless we use a restricted form of programming which guarantees that programs always terminate; and which is the case in real-time systems.

The two main criteria for evaluating a WCET estimation method are: *safety* (does it produce upper bounds of execution times or not?) and *precision* (are the upper bounds close to the actual WCET?) In this section, the two main families of methods to bound the WCET of a task are presented [34].

1.3.1 Static Methods

This class of methods does not rely on executing code on real hardware, but it analyzes the source code and/or the final executable, combines it with some model of the system's hardware, and obtains upper bounds from this combination. In this report, we are especially interested in source code analysis because we try to focus in the impact of virtualization and JIT compilation on the WCET estimation; and it is obvious that there is no generated object code for the whole task in a JIT compilation. The essential steps of the static analysis are depicted in Figure 1.3.

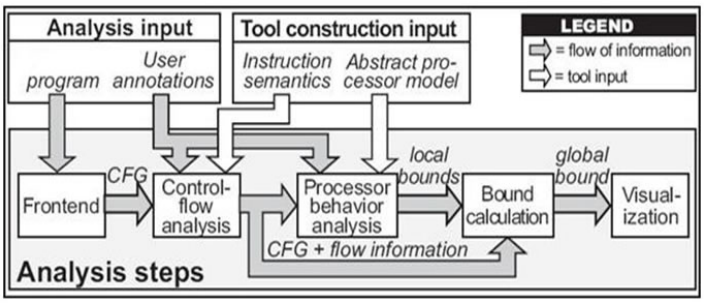


Figure 1.3: Core components of a timing-analysis tool

The input of such a method is a source code and some user annotations. The annotation of tasks, with information available from the developer, is a generic technique to supply information that the tool needs; e.g. ranges for the input values of the task, loop bounds, shapes of nested loops... The steps are:

Front end

The mostly used program representation is the control flow graph (CFG). This step aims at constructing it. Problems in such a construction are created by dynamic jumps and calls with computed target address. Dynamic calls occur in source code in the form of calls through function pointers and calls to virtual functions (which are a common situation in object-oriented programs).

Control Flow analysis (CFA, high-level analysis)

The purpose of CFA is to gather information about possible execution paths. The input of flow analysis consists of a task representation (e.g. CFG and call graph) and possibly additional information derived from annotations or by a value analysis. A value analysis is a static program analysis which aims at computing ranges for the values in the processor registers and local variables at every program point (it may be just sound approximations). This analysis is useful to predict the cache behavior, to determine loop bounds, and to detect infeasible paths. Indeed, some paths in the superset described by the CFG are infeasible, and it is better to eliminate them.

Processor behavior-analysis (low-level analysis)

Early approaches to the timing-analysis problem assumed context independence of the timing behavior; i.e. the execution time for individual instructions were independent from the execution history. So, if a task first executes a code snippet A (with an upper bound uA) and then a snippet B (with an upper bound uB), the worst-case bound for $"A;B"$ is $"uA + uB"$. This is no longer true for modern processors with caches and pipelines. The execution time of individual instructions may vary depending on the state of the processor in which they are executed.

To find precise execution time bounds for a given task, it is necessary to analyze what the occupancy state of these components is for all paths leading to the task's individual instructions. Processor-behavior analysis determines invariants about these occupancy states

for the given task. Most approaches use data flow analysis and abstract interpretation [5]. Processor-behavior analysis needs a model of the architecture: an abstract processor model which is a simplified model that is conservative with respect to the processor timing behavior.

An analysis of the behavior of hardware caches yields to a tighter WCET of programs. The idea is to derive for each memory reference its worst-case behavior (hit/miss). To enforce safety, static cache analysis methods have to account for every possible cache contents, at every point in the execution, considering all paths to gather. Possible cache contents can be represented as sets of concrete cache states (*collecting semantics*) or by a more compact representation called abstract cache states (*Abstract Interpretation*) as we will detail in Chapter 2.

The complexity of the processor-behavior analysis subtask and the set of applicable methods critically depend on the complexity of the processor architecture. Most powerful microprocessors suffer from timing anomalies. Timing anomalies are contra-intuitive influences of the local execution time of one instruction on the global execution time of the whole task (see [20]).

Bound calculation

This step computes an upper bound of all execution times of the whole task, based on the flow and timing information derived in the previous phases. There are three main classes of bound calculation methods: path-based, structure-based, and techniques using implicit-path enumeration (IPET). We will detail here the IPET method which is the most used technique.

In IPET, program flow and basic-block execution time bounds are combined into sets of arithmetic constraints. Each basic block and program flow edge in the task is given a time coefficient (t_{entity}) expressing the upper bound of the contribution of that entity to the total execution time, and a count variable (x_{entity}), corresponding to the number of times the entity is executed. An upper bound is determining by maximizing $\sum_{i \in entities} x_i * t_i$, where the execution count variables are subject to constraints reflecting the structure of the task and possible flows. Figure 1.4 shows an example of the application of the technique. The CFG of the task is depicted in the left side, and the ILP formulation in the right one. The variable x_A represents the number of execution of the basic block A, and x_{HA} is that of the edge from H to A. The constant 5 in the objective function corresponds to t_B,...

1.3.2 Measurement-Based Methods

These methods execute the task or task parts on the given hardware or a simulator for some set of inputs. If you have the worst-case input for a given task, the precise determination of its WCET is an easy job. The problem is that determining such input is a very hard task. Exhaustively exploring all the input domain is too expensive as the number of cases to be explored is exponential w.r.t to the number of input variables. Measurements of a subset of all possible executions produce *estimates*, not safe bounds, if the subset is not guaranteed to contain the worst case.

Other approaches measure the execution time of basic blocks. The measured execution times are then combined and analyzed to produce non-safe estimates of the WCET. Thus, measurements replace the processor-behavior analysis used in static methods. The main problem of dynamic methods to bound the WCET might be *safety*. In this context, measurement-based methods are mainly used to validate static analysis methods.

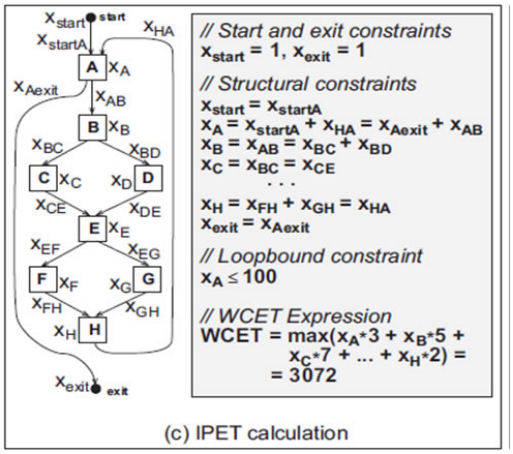


Figure 1.4: IPET technique

The WCET estimation is difficult because of two things: history-based execution and dynamic features. The analysis of a branch predictor behavior is difficult. Indeed, the actual decision of the predictor is based on the precedent decisions; so the analysis of its behavior is more complex. The principle of "adaptive optimization" is not far from that of a branch predictor: the level of optimization is determined by the execution history. Dynamic features like dynamic jumps are a great source of imprecision in the WCET estimation. And for the lack of luck, languages such as Java are full of these features: virtual calls, dynamic class loading...

1.4 Predictability of HLL VMs & JIT Compilation

The use of an HLL VM like Java for real-time systems is justified by the need for software portability. The Java language's design includes some non-deterministic performance effects such as class loading and dynamic dispatching (§1.4.1). The need for performance has introduced the notion of JIT compilation in a VM; but it contains also non-deterministic effects (§1.4.2). To use these two techniques (Java and JIT compilation) in hard-real time systems, one need to predict statically their effects and calculate the WCET of a given task.

1.4.1 Problems Caused by Java and other HLL VMs

We start first by describing some complexities appeared when using the object-oriented programming in general and the interpreted Java in particular. Object-oriented languages are not yet so widespread in real time systems. While simplifying for the programmer, they make WCET calculation of programs a harder task than for imperative programs. Some of those problems are listed in [12, 22]:

- **Class loading:** A Java-conformant JVM must delay loading a class until it is first referenced by a program. Loading a class can take a variable amount of time depending on the speed of the medium (disk or other) the class is loaded from, the class' size, and the overhead incurred by the class loaders themselves. The delay to load a class can

commonly be as high as 10 milliseconds. If tens or hundreds of classes need to be loaded, the loading time itself can cause a significant and possibly unexpected delay. Careful application design can be used to load all classes at application start-up, but this must be done manually because the Java language specification does not let the JVM perform this step early.

- **Dynamic binding:** This feature creates additional complexity, since it is not always known at compile time which method is going to execute. Even if one has knowledge of the WCET of all possible methods, dynamic binding may give a huge overestimation.
- **Dynamic allocation and deallocation of memory:** The time to allocate memory is not always predictable [12]. For deallocation, there are two mechanisms: explicit deallocation or automatic garbage collection. The garbage collector may interrupt the real-time system for an unpredictable time. However, many works consider it as a scheduling problem and not a WCET analysis problem [22, 30].
- **Thread management:** Standard Java provides no guarantees for thread scheduling or threads priorities. An application that must respond to events in a well-defined time has no way to ensure that another low-priority thread will not get scheduled in front of a high-priority thread. To compensate, a programmer would need to partition an application into a set of applications that the operating system can then run at different priorities. This partitioning would increase the overhead of these events and make communication between the events far more challenging.

Most approaches in the WCET estimation field for Java have simply assumed that dynamic dispatching features should be prohibited. In [14], E. Yu-shing et al. propose minimum annotations to address dynamic dispatching and so to reduce the overestimation of the WCET calculation. They use the user's knowledge about the application and the targets of a virtual call to increase the precision of the WCET estimation.

For JVMs, we find in [1] the proposal of a portable Java bytecode WCET estimation for Java bytecode (JBC) programs. The approach characterizes properties of a program that determine its WCET in a machine-independent, abstract way. Although the timing information is abstract, it represents the details that are necessary to port the information to a specific target platform and take into account its features (e.g. instruction pipeline or cache). After completion of the machine-independent WCET analysis, the WCET information is added to the Java class file of the program. Whenever the program is to be ported to a specific machine, the WCET information contained in the Java class file is augmented with machine-specific information (timing model) and evaluated for this machine. The models which are predictable and portable are: the hardware VM, the simple AOT translation and the interpreted VM because the compilation time of those models is bounded. So, the low-level analysis is portable to those different types of JBC execution.

1.4.2 Problems Caused by the JIT Compilation

Authors in [1] consider JIT compilation unsuitable for real-time systems as the worst-case computation time has to include the time to compile the code which may be large and difficult to predict. Actually, most of the difficulties in the WCET estimation are raising form the dynamic properties of programs, adding another dynamic dimension (JIT compilation) makes

things much harder. Techniques such as "adaptive optimization" may be a great source of unpredictability of JIT compilation.

Our work is a first attempt to using JIT compilation in hard real-time systems; therefore we do not care about the problems caused by Java (§1.4.1) since those problems are tackled by many researches. There are many steps to achieve before using a JIT compiler in a real-time system; our work is only an answer of a first question: *how many times a method will be compiled?*

Chapter 2

Predictability of the JIT Compiler Launching

In this chapter, we present the work that is done during the internship. Using a JIT compiler in a real-time system is not an easy job as we have seen in the first chapter. Our goal is to incorporate the overhead caused by JIT compilation in the WCET estimation. When the static analysis of the program ends, each call to a function (or return) in the program has to be classified as always hit (whatever the execution path, the function is cached at this point) or not. In the negative case, the worst case compilation time must be incorporated in the worst-case execution time of the program. So, the objective is to answer statically the following question: how many times a function will be compiled? If the compiled code of a method stay in memory until the end of the program, then a function will be compiled at most one time.

An embedded system usually has hard resource constraints as a limited amount of memory, therefore we are obliged to use a limited memory zone which acts like a software cache to cache the compiled code. We have taken the following assumptions:

- We consider *mono-processor* and *mono-task* real-time systems.
- A limited compiled code cache. When there is no free place in the cache to hold the new compiled function, the binary code of others functions will be evicted from the cache. This mechanism will complicate the abovementioned question.
- A function must be systematically compiled (if it is not cached), but only when it is used.
- We do not allow advanced techniques of JIT compilation such as adaptive optimization.

Our work is a proposition and an analysis of some cache management systems. The choice of a cache management system is guided by some real-time requirements. The analysis is essentially based on abstract interpretation, therefore we introduce data-flow analysis and abstract interpretation in Sections 1. In the rest of the chapter, we present three cache management systems: fixed-size blocks with LRU, binary buddy system with FIFO and a fixed-layout cache management system.

2.1 Background

The goal of this section is to introduce the needed information about data flow analysis and abstract interpretation. Those concepts are illustrated by examples related to cache analysis. This is an important step for presenting later a formal solution of the problem.

2.1.1 Data Flow Analysis

Data Flow Analysis (DFA) is the pre-execution process of ascertaining and collecting information about the possible run-time modification, preservation and usage of certain quantities in a computer program [13]. In our work the analysis is interprocedural and contextual, i.e. the analysis of a function depends on the context in which it is called. We represent a program as a control flow graph; a (CFG) per function and the set of CFGs are combined in an extended one by the call/return associations as depicted in Figure 2.1. There are three kinds of basic blocks: call, return and ordinary ones. An edge from a call node n_c to its successor n_s is replaced in the extended CFG by an edge from n_c to the entry node of the callee function and an edge from its return node to n_s.

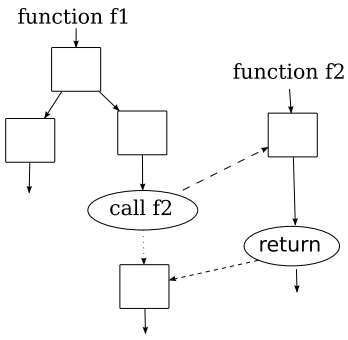


Figure 2.1: An extended CFG

Many classical DFAs exist in literature [24, 13]: available expressions, reaching definitions, etc. Kam and Ullman [15] provided a generic framework called the monotone data flow analysis framework. At each point k of the program P (each node of the CFG), we put $info_{in}(k)$ as the relevant information that is true on entry to the node, and $info_{out}(k)$ as the relevant information that is true at exit. Those two values represent the information derived form every possible execution path which reach the node k (the Meet Over all Path (MOP) solution). The idea behind the monotone framework is that the relevant information can be modeled as a mathematics structure and the effect of a node on $info_{in}(k)$ as a transfer function defined on the structure. We start first with a few mathematical facts (for more details see [2]).

Example: This example will be used to illustrate the concepts in following. In our work, the DFA is a software cache analysis. At each moment of the program execution, the cache holds some functions (this snapshot is called a concrete cache state). Let D the set of all possible concrete cache states. We try in this analysis to infer the information that for all possible paths, a function is always cached at a given node k (always hit). If we cannot decide, the compilation time must be incorporated in the WCET of the program.

Lattice Theory

Poset: A *partially ordered set* (poset) is a couple $(B, \sqsubseteq)$ with B a set and $\sqsubseteq$ a reflexive, anti-symmetric and transitive relation on B. A poset $(B, \sqsubseteq)$ verifies the ascending chain condition if for all ascending sequence $b_0 \sqsubseteq b_1 \sqsubseteq \cdots b_n \sqsubseteq \cdots$ there exists an index k from which the sequence is stationary ($\forall n \geq k : b_n = b_k$). A *finite* poset verifies the ascending chain condition. We often represent a poset as a Hasse diagram (a graph without the transitive relation).
Example: $(\wp(D), \subseteq)$ is a finite poset.

Lattice: A lattice is 4-tuple $(B, \sqsubseteq, \sqcup, \sqcap)$ with:

- $(B, \sqsubseteq)$ a poset.
- $\sqcup$ a binary least upper bound (*join*):
  - $\forall b_1, b_2 \in B : b_1 \sqsubseteq b_1 \sqcup b_2 \wedge b_2 \sqsubseteq b_1 \sqcup b_2$.
  - $\forall b_1, b_2, b_3 \in B : b_1 \sqsubseteq b_3 \wedge b_2 \sqsubseteq b_3 \Rightarrow b_1 \sqcup b_2 \sqsubseteq b_3$.
- $\sqcap$ a binary greatest lower bound (*meet*):
  - $\forall b_1, b_2 \in B : b_1 \sqcap b_2 \sqsubseteq b_1 \wedge b_1 \sqcap b_2 \sqsubseteq b_2$.
  - $\forall b_1, b_2, b_3 \in B : b_3 \sqsubseteq b_1 \wedge b_3 \sqsubseteq b_2 \Rightarrow b_3 \sqsubseteq b_1 \sqcap b_2$.

A poset with only the meet (resp. join) operation is called a meet-semilattice (resp. join-semilattice).
Example: $(\wp(D), \subseteq, \cup, \cap)$ is a lattice.

Complete lattice: A complete lattice is a triple $(B, \sqsubseteq, \bigsqcup)$ with

- $(B, \sqsubseteq)$ a poset,
- $\bigsqcup$ a least upper bound: for all parts S of B,
  - $\forall b \in S : b \sqsubseteq \bigsqcup S$
  - $\forall b_2 \in B : (\forall b_1 \in S : b_1 \sqsubseteq b_2) \Rightarrow \bigsqcup S \sqsubseteq b_2$

All finite lattices are complete, and all complete lattices have a greatest element (the top element $\top$) and a least element (the bottom element $\bot$).
Example: $(\wp(D), \subseteq, \bigcup)$ is a complete lattice with ϕ as the least element and D as the greatest one. The meaning of $\subseteq$ here is *more precise than*, so the more concrete cache states we have, the less information about the actual cache state we can extract. Therefore, if $info_{in}(k) = D$ then everything is possible.

Monotone function: A function $\psi : B_1 \longrightarrow B_2$ between posets $(B_1, \sqsubseteq_1)$ and $(B_2, \sqsubseteq_2)$ is *monotone* if $\forall b_1, b_2 \in B_1 : b_1 \sqsubseteq_1 b_2 \Rightarrow \psi(b_1) \sqsubseteq_2 \psi(b_2)$.

Fixpoint: Consider a monotone function $\psi : B \longrightarrow B$ on a complete lattice $(B, \sqsubseteq, \bigsqcup)$. A *fixed point* of ψ is an element $b \in B$ such that $\psi(b) = b$. The least fixpoint $lfp(\psi)$ of ψ exists [32].

Monotone Framework

The relevant information at each node k is modeled by an element of a join-semilattice $(B, \sqsubseteq)$ that has an element $\top$ and which verifies the ascending chain condition (or by duality as a meet-semilattice). The *join* (least upper bound) operation represents the effect of information converging from paths. The effect of each node is modeled by an operation ψ_k on the semilattice. Because there are many different operations (an operation per node), we must form a *monotone operation space* Ψ associated with B which verifies three conditions:

- each ψ_k is in Ψ and it is monotone.
- Ψ contains the identity function (because certain nodes have no effect).
- Ψ is closed under composition of functions. This condition reflects the action of passing information through successive nodes.

So, a *monotone* framework consists of a complete lattice and a monotone operation space. A *distributive* framework is a monotone framework where: $\forall \psi_k \in \Psi : \psi_k(b_1 \sqcup b_2) = \psi_k(b_1) \sqcup \psi_k(b_2)$.
Example: Let $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ be the list of program functions. A cache update function: $\delta : D \times \mathcal{F} \longrightarrow D$ describes the new cache state for a given cache state and a referenced function. The effect of an update function is determined by the cache management system (cache structure, replacement policy...). For a call node k (or a return one) which references the function f_i, we associate a function $\psi_k = \Delta_i$ such that,$\forall x \in \wp(D) : \Delta_i(x) = \{d' \in D / d' = \delta(d, f_i) \wedge d \in x\}$. For an ordinary node $\psi_k = id$.
$\forall x, y \in \wp(D), \forall i : x \subseteq y \Leftrightarrow \Delta_i(x) \subseteq \Delta_i(y)$. So, Δ_i is a monotone function on $(\wp(D), \subseteq, \bigcup)$. The composition of monotone functions is a monotone function and if we take the composition closure then we will have a monotone operation space. In summary, we will have a distributive framework.

The Meet Over all Paths (MOP) solution is undecidable [13], and instead of the MOP solution we compute the MFP one (the maximum fixed point). They are equivalent when the framework is distributive. The problem is usually formulated as an equation system (see [24] for the overall pattern). The solution of this equation system is the least fixpoint (or the greatest one).
Example: For our example the equation system is:

$$info_{in}(k) = \begin{cases} \{\bar{d}\} & \text{if } k \in \text{init}(P) \\ \bigcup info_{out}(k')/(k', k) \in \text{flow}(P) & \text{else} \end{cases}$$

Such that: $\bar{d}$ is the initial concrete cache state which contains the entry function.

$$info_{out}(k) = \begin{cases} info_{in}(k) & \text{if } k \text{ is neither a call block nor a return one} \\ \Delta_i(info_{in}(k))/f_i \text{ is the referenced} & \\ \text{function by the node } k & \text{else} \end{cases}$$

Such that: $init(P)$ is the entry nodes of the program and $flow(P)$ is the control flow represented as a set of edges.

2.1.2 Abstract Interpretation

Sometimes calculations on a complete lattice may be expensive or even uncomputable. Unfortunately this the case for the cache analysis; the concrete cache states domain D is huge and so more $\wp(D)$. We call the analysis developed in the previous example "the *collecting semantics*". The idea of approximating a program semantics corresponds to a restriction of the set of properties used to express the behavior of a program i.e. the restriction of $\wp(D)$ to $\bar{D}$ where $\bar{D} \subseteq \wp(D)$. Instead of representing properties in a concrete domain, one represents more abstract information in an abstract domain and which is the idea of abstract interpretation [5]. Abstract Interpretation is a formal method in static analysis used to determine statically (without executing the program) uncomputable properties of programs.

So, we have the concrete world represented by a complete lattice generally of the form $(\wp(B), \subseteq, \bigcup)$; an abstract world modeled as a complete lattice $(\widehat{B}, \sqsubseteq, \bigsqcup)$. The relation between the two complete lattices is expressed by two monotone functions:

$$(\wp(B), \subseteq, \bigcup) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} (\widehat{B}, \sqsubseteq, \bigsqcup)$$

Where:

- α is an abstraction function, $\alpha : \wp(B) \longrightarrow \widehat{B}$
- γ is a concretisation function, $\gamma : \widehat{B} \longrightarrow \wp(B)$

This relation explains the meaning of elements of $\widehat{B}$ in terms of elements of $\wp(B)$ and it is called a *Galois connection*.
Example: A very famous example is the abstraction by signs [5]. The concrete world is $(\wp(\mathbb{Z}), \subseteq, \bigcup)$ and the abstract world is represented by the lattice of signs depicted in Figure 2.2. As examples, $\alpha(\{x \leq 0 / x \in \mathbb{Z}\}) = -_0$ and $\gamma(0) = \{0\}$. Operations, such as addition, can be defined on this lattice; e.g. 0 *plus* $+ = +_0$.

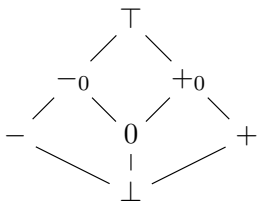


Figure 2.2: The sign lattice

To not lose safety by going back and forth between the two lattices, the two functions must verify the two following properties:

$$\forall x \in \wp(B) : x \subseteq \gamma \circ \alpha(x) \text{ and } \forall \hat{b} \in \widehat{B} : \alpha \circ \gamma(\hat{b}) \sqsubseteq \hat{b}$$

If γ is injective then $\forall \hat{b} \in \widehat{B} : \alpha \circ \gamma(\hat{b}) = \hat{b}$; and we call the relation a *Galois insertion*.

Function approximation: For a monotone function $\widehat{\psi} \in \widehat{B} \longrightarrow \widehat{B}$ and a function $\psi \in \wp(B) \longrightarrow \wp(B)$, $\widehat{\psi}$ is a correct approximation of $\psi \Leftrightarrow \alpha \circ \psi \circ \gamma \sqsubseteq \widehat{\psi}$. Hence, $\alpha \circ \psi \circ \gamma$ is the best abstract function $\widehat{\psi}$ and which verifies $\alpha(lfp(\psi)) \sqsubseteq lfp(\widehat{\psi})$.
You can find proofs and how we use the abstraction with a monotone framework in [24].

Reduced Product

Let us take two Galois connections:

$$(\wp(B), \subseteq, \bigcup) \underset{\gamma_1}{\overset{\alpha_1}{\rightleftarrows}} (\widehat{B}_1, \sqsubseteq_1, \bigsqcup_1) \text{ and } (\wp(B), \subseteq, \bigcup) \underset{\gamma_2}{\overset{\alpha_2}{\rightleftarrows}} (\widehat{B}_2, \sqsubseteq_2, \bigsqcup_2)$$

We can run the two analyses without any collaboration between them; but we can do better. The objective is to cooperate the two analyses to obtain information which is more precise than we can obtain with separated analyses. Because we have the same concrete domain, we can combine these two abstractions with a new connection:

$$(\wp(B), \subseteq, \bigcup) \underset{\bar{\gamma}}{\overset{\bar{\alpha}}{\rightleftarrows}} (\widehat{B}_1 \times \widehat{B}_2, \sqsubseteq, \bigsqcup)$$

Where:

$$\forall (\hat{b}_1, \hat{b}_2), (\hat{b}'_1, \hat{b}'_2) \in (\widehat{B}_1 \times \widehat{B}_2) : (\hat{b}_1, \hat{b}_2) \sqsubseteq (\hat{b}'_1, \hat{b}'_2) \Leftrightarrow \hat{b}_1 \sqsubseteq_1 \hat{b}'_1 \wedge \hat{b}_2 \sqsubseteq_2 \hat{b}'_2.$$

$\forall x \in \wp(B) : \bar{\alpha}(x) = (\alpha_1(x), \alpha_2(x))$ and $\forall (\hat{b}_1, \hat{b}_2) \in \widehat{B}_1 \times \widehat{B}_2 : \bar{\gamma}(\hat{b}_1, \hat{b}_2) = \gamma_1(\hat{b}_1) \cap \gamma_2(\hat{b}_2)$.

Example: If we combine the Galois insertion of signs which infers $\hat{b}_1 = +$ (i.e. $\gamma_1(\hat{b}_1) = \{n > 0\}$) with the Galois insertion of intervals which infers $\hat{b}_2 = [-5, 2]$, then we can refine $\hat{b}_2$ to $\hat{b}_2 =]0, 2] = \alpha_2(\gamma_1(\hat{b}_1) \cap \gamma_2(\hat{b}_2))$.

In abstract interpretation, we call this concept *reduced product*. We define a reduction operator ρ which allows computations on a reduced product $\rho(\widehat{B}_1 \times \widehat{B}_2)$. It allows to combine efficiently and for free two correct approximations. If $\hat{\psi}$ is an abstract function, then $\rho \circ \hat{\psi} \circ \rho$ is a more precise version (but not the most precise).

2.2 Problem Statement and Assumptions

Three cache management systems are proposed in the rest of this chapter. Each one of them has its own advantages and drawbacks. A cache analysis is developed for each system. But, we present first the assumptions and constraints behind our propositions.

Unlike a hardware cache, a compiled code cache needs neither a fixed structure nor a fixed replacement policy. Indeed, the JIT compiler can switch statically (at a program starting) from a cache management method to another accordingly to the designer choice. Our objective is to find an appropriate predictable management system. The software cache management is similar to the dynamic storage allocation (DSA), except there is no release operation. The application can only request blocks to load the binary code of a function at a call or a return instruction (and we say that the function is referenced) if the function is not cached. If there

is not enough free place in the cache to hold the requested binary code, some functions will be evicted according to the replacement policy. Dynamic storage allocators have no replacement mechanism. They just throw a memory overflow exception when there is no available memory to serve a new request.

This similarity between software cache management and DSA allows us to adjust some techniques of DSA for our purposes. Several efficient implementations of dynamic storage allocator exist [35], but if we are interested in using them for real-time systems, one will have to take in consideration the predictability factor and not only the performance issues. In [28], Puaut provides a guideline to developers of real-time systems to decide whether they will use DSA or not in their systems. The paper gives the analytic worst-case of the routines (malloc/free) of a comprehensive panel of allocators. The assumptions behind those allocators are imposed by the real-time nature. We have adopt some of those assumptions:

- The algorithms allocate areas of real memory, i.e. no address translation nor paging.
- There is no block relocation. Indeed, although the block relocation mechanism has its own advantages, it is a time consuming process, therefore we avoid to use it in JIT compilation.

The requirements for a real-time software cache management system are:

- **Temporal predictability:** The JIT compiler must keep information about free memory blocks to serve new requests; and it has to perform some form of search to find a suitable free block. The WCET of this search must be constant or bounded. In terms of dynamic storage allocators; *Binary buddy systems* [26], *Half-fit* [25] and *TLSF* [23] have been considered as real-time allocators. We will take a glance on the binary buddy system because we have used it in our work, not the allocator but the idea behind it (see §2.4 and §2.5).
- **Spatial predictability:** *Fragmentation* is the inability to reuse memory that is free. *Internal fragmentation* is the result of rounding up the requested size to the closest predefined block size. *External fragmentation* result from breaking available memory into blocks. Two blocks can be recombined once they are free only if they are neighbors. Real-time systems have to operate over very long periods; and therefore, fragmentation causes a significant performance degradation. For dynamic storage allocators, the fragmentation is unpredictable for almost them except for *Compact-Fit* [6]. This allocator use the relocation mechanism which we avoid. So, we must be careful about fragmentation when we design the management system, because it can be the most important reason that makes a given management method better than another (see §3.3.1).

2.3 Fixed-Size Blocks with LRU System

The cache is divided into a set of fixed-size blocks. The block size BS is equal to the size of the largest binary code of all the program functions. Let $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ be the list of program functions. $Size : \mathcal{F} \longrightarrow \mathbb{N}^+$ a function associates with each function f_i the size of its binary code $Size(f_i) = s_i$. So the size of a cache block will be $BS = \max(s_1, s_2, \ldots, s_n)$; and the number of blocks will be $NB = \lfloor \frac{C}{BS} \rfloor$ where C is the cache size. The replacement policy for this management system is the LRU (Least Recently Used) policy.

- **Advantages:** We can borrow the hardware cache analysis and apply it easily on software caches. This analysis is straightforward, fast and efficient.
- **Drawbacks:** The most significant drawback of this management system is *internal fragmentation* specially when the sizes of functions are heterogeneous. For each function f_i in the cache, $(BS - s_i)$ memory units are wasted. In terms of *external fragmentation*, $(C \bmod BS)$ memory units are never exploited. Another drawback is that the implementation of LRU in the allocator is more time consuming than other policies (such as FIFO) because we must keep track of age of each function in the cache and searching always for the function with the greatest age.

In hardware cache analysis, a framework based on abstract interpretation [5] is often used. For LRU replacement policy, there are usually two analyses [9]. A must-analysis which determines a set of memory blocks (in our case, a set of functions) that *must* be in the cache at a given program point upon any execution. A may-analysis which determines all the functions that *may* be in the cache at a given program point. The must-cache information is used to derive safe information about cache *hits*, and may-cache analysis is used to safely predict cache *misses*.

In follow, we transpose the analysis of fully associative hardware caches with LRU replacement policy [9] to our case. Instead of a collecting semantics, we will build two abstract domains: a *Must* domain to infer information about always hits and a *May* domain to infer the information about always misses. This is a formal construction to ensure the correctness of the approximation and the termination of the analysis.

2.3.1 Must Analysis

Let $Age = \{0, 1, \ldots, NB - 1\}$ and $Age_\omega = Age \cup \{\omega\}$ with the order $\forall i \in Age : i < \omega$ (we can take ω as NB). Let us take the abstract domain $Must = \mathcal{F} \longrightarrow Age_\omega$ (We associate with each function an age in the software maintained function cache).

- $m_1, m_2 \in Must : m_1 \sqsubseteq_{must} m_2 \Leftrightarrow \forall f_i \in \mathcal{F} : m_1(f_i) \leq m_2(f_i)$.
- $(Must, \sqsubseteq_{must})$ is a poset.
- $m_1, m_2, m_3 \in Must : m_3 = m_1 \sqcup_{must} m_2 / \forall f_i \in \mathcal{F} : m_3(f_i) = \max(m_1(f_i), m_2(f_i))$.
- $m_1, m_2 \in Must; m_3 = m_1 \sqcap_{must} m_2 / \forall f_i \in \mathcal{F} : m_3(f_i) = \min(m_1(f_i), m_2(f_i))$.

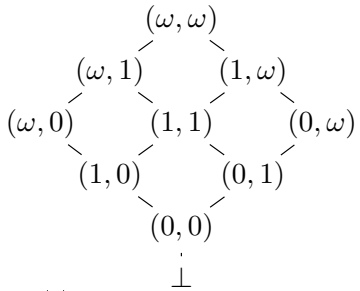
$(Must, \sqsubseteq_{must}, \sqcup_{must}, \sqcap_{must})$ is a finite lattice and therefore $(Must, \sqsubseteq_{must}, \bigsqcup_{must})$ is a complete one.

Example: For a cache with $NB = 2$ and $|\mathcal{F}| = 2$, we obtain the lattice in picture 2.3(c). An element $(1, \omega)$ means that the maximal age of f_1 is 1 and the maximal age of f_2 is ω.

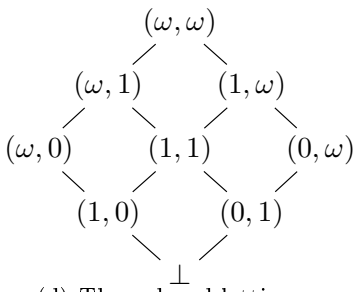
Now, to apply the abstract interpretation we need a Galois connection:

$$(\wp(D), \subseteq, \bigcup) \underset{\gamma_1}{\overset{\alpha_1}{\rightleftarrows}} (Must, \sqsubseteq_{must}, \bigsqcup_{must})$$

We define the abstraction function α_1 as:



(c) The original lattice



(d) The reduced lattice

Figure 2.3: A lattice for an LRU cache

- $\alpha_1(\phi) = \bot$. We extend $(Must, \sqsubseteq_{must}, \bigsqcup_{must})$ with the least element $\bot$.
- $\forall x \in \wp(D) : \alpha_1(x) = m \in Must / \forall f_i \in \mathcal{F} : m(f_i) =$the maximal age of the function f_i in all concrete cache states in x.

For a concrete cache state, if f_i it has just been referenced, then the age of f_i is 0; and if f_i is not cached then the age will be ω. So, $\alpha_1(x)(f_i) = \omega$ means that f_i *can be not* cached, and $\alpha_1(x)(f_i) = k$ means that f_i it will stay at least $NB - k - 1$ references (calls or returns) that are not in the abstract cache (or older than f_i) for all paths reaching the concerned node. The meaning of $\sqsubseteq_{must}$ is not "*more precise*" but "*better than*" in terms of hit information.

Example: Let x_0 a set of concrete cache states. x_0 contains two states: $[f_1(0), \phi]$ and $[f_1(1), f_2(0)]$ ($f(a)$ means that the age of f is a). So, $\alpha_1(x_0) = m_0 / m_0(f_1) = 1 \wedge m_0(f_2) = \omega$.

We define the concretisation function as:

- $\gamma_1(\bot) = \phi$.
- $\forall m \in Must \backslash \{\bot\} : \gamma_1(m) = \{d \in D / \forall f_i \in \mathcal{F} :$ age of f_i in d $\leq m(f_i)\}$.

Example: For the previous example, $\gamma_1(m_0) = \{d \in D /$age of f_1 in d $\leq 1 \wedge$ age of f_2 in d $\leq \omega\}$. So, $\gamma_1(m_0) = \{[f_1(0), \phi], [f_1(0), f_2(1)], [f_2(1), f_1(0)], [f_2(0), f_1(1)], [f_1(1), f_2(0)]\}$.

γ_1 is not injective because in addition to $\gamma_1(\bot) = \phi$ there are some $m \in Must \backslash \{\bot\}$ such as $\gamma_1(m) = \phi$ (e.g., if $m(f_i) = 0$ and $m(f_j) = 0$). From now, we suppose that $|\mathcal{F}| \leq NB$. Generally,

$$\forall m \in Must \backslash \{\bot\} : \gamma_1(m) = \phi \Leftrightarrow \exists 0 \leq k < \omega : \sum_{m(f_i) \leq k} 1 > k + 1.$$

In our analysis this case never happens, and if we eliminate those cases, then γ_1 will be injective and the analysis is still sound because the reduced set is also a complete lattice. We can easily prove that if m_1 is not valid then: $\forall m_2 \in Must \backslash \{\bot\} : m_2 \sqsubseteq_{must} m_1 \Rightarrow m_2$ is also not valid. From now, our operations are on the reduced lattice.

Example: For the lattice of the previous example, the element $(0, 0)$ is not valid. The reduced lattice is depicted in Figure 2.3(d).

We have, $\forall x \in \wp(D) : x \subseteq \gamma_1(\alpha_1(x))$, and $\forall m \in \mathit{Must} : m = \alpha_1(\gamma_1(m))$. So, this abstraction is a correct approximation.

The access function

Let the function $cl : \wp(D) \times \mathcal{F} \longrightarrow \{H, M, U\}$. $\{H, M, U\}$ is a join-semilattice (Figure 2.4). If $x \in \wp(D) \backslash \{\phi\}$ then $cl(x, f_i)$ describes the access type of a reference to f_i in a set of concrete states. So:

$$cl(x, f_i) = \begin{cases} \text{Hit } \mathbf{(H)} & \text{if } \forall d \in x : f_i \text{ is in } d \\ \text{Miss } \mathbf{(M)} & \text{if } \forall d \in x : f_i \text{ is not in } d \\ \text{Unclassified } \mathbf{(U)} & \text{otherwise} \end{cases}$$

The abstraction of the access function will be: $\widehat{cl} = cl \circ \gamma_1$. Let $m \in \mathit{Must} \backslash \{\bot\}$; if $m(f_i) \neq \omega$ then it is a hit **(H)**, else it is an unclassified reference unless if $\sum_{m(f_k) \neq \omega} 1 = NB$ and so it will be a miss **(M)**. This abstraction is exact. The only issue is if $\widehat{cl}(m, f_i) = \mathbf{U}$ but $cl(\gamma_1(m), f_i) = \mathbf{M}$, but this is impossible, because $\gamma_1(m)$ contains at least an element d where the age of f_i in d equals to $NB - 1$.

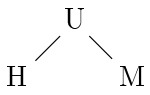


Figure 2.4: The access classification lattice

Example: Always using the same example, $\widehat{cl}(m_0, f_2) = \mathbf{U} = cl(\gamma_1(m_0), f_2)$.

Abstraction of the update function

For LRU, when a miss happens and there is no free place in the cache, the function with age equals to $NB - 1$ will be evicted, the inserted function will have the age equal to 0 and the age of all other function will be incremented by 1. If the access is a hit then the age of the referenced functions will be updated to 0 and functions with age less than the age of the referenced function will become older (ages incremented by 1).

As the theory of abstract interpretation says: the best abstraction of Δ_i is $\widehat{\Delta_i} = \alpha_1 \circ \Delta_i \circ \gamma_1$. We are lucky since this specification can be written easily as an algorithm: Let $m, m' \in \mathit{Must} \backslash \{\bot\}$ two abstract cache states such as $m' = \widehat{\Delta_i}(m)$, So: $m'(f_i) = 0$ and

$$\begin{cases} m'(f_j) = m(f_j) + 1 / m(f_j) < m(f_i) & \text{if } \widehat{cl}(m, f_i) = \mathbf{H} \\ m'(f_j) = m(f_j) + 1 / m(f_j) \neq \omega & \text{if } \widehat{cl}(m, f_i) = \mathbf{M} \vee \widehat{cl}(m, f_i) = \mathbf{U} \end{cases}$$

If the reference to f_i is *unclassified*, and because of the existence of an element d in $\gamma_1(m)$ such that age of f_i in d equals to $NB - 1$, then the effect of the update function is like when the access is a miss.

To conclude, the data-flow equations of the must-analysis will be: for each basic block k in the program P:

$$AC_{in}(k) = \begin{cases} \bar{m} & \text{if } k \in \text{init}(P) \\ \bigsqcup_{must} AC_{out}(k')/(k',k) \in \text{flow}(P) & \text{else} \end{cases}$$

Such that $\bar{m} = \alpha_1(\{\bar{d}\})$ i.e. $\forall f_i \in \mathcal{F} : \bar{m}(f_i) = 0$ if f_i is the entry function and $\bar{m}(f_i) = \omega$ else.

$$AC_{out}(k) = \begin{cases} AC_{in}(k) & \text{if } k \text{ is not a call block nor a return one} \\ \widehat{\Delta_i}(AC_{in}(k))/f_i \text{ is the referenced} & \\ \quad \text{function in node } k & \text{else} \end{cases}$$

Finally, the termination and the correctness of this analysis are guaranteed by using the theory of abstract interpretation [5].

2.3.2 May Analysis

The may-analysis is roughly dual to the must-analysis. Instead of taking the maximal age of a function, we take the minimal one. This analysis is used in [9] to predict misses in hardware caches. In our case, there are no timing anomalies, therefore we do not need for a may-analysis. But, we present it for two reasons: to explain easily the intuition behind the may-analysis which is used later in the analysis of an another management system (§2.5) and to explain also why the collaboration between the may-analysis and the must-analysis can be useful for some management systems and not for others.

Let us take the abstract domain $May = \mathcal{F} \longrightarrow Age_\omega$.

- $m_1, m_2 \in May : m_1 \sqsubseteq_{may} m_2 \Leftrightarrow \forall f_i \in \mathcal{F} : m_1(f_i) \geq m_2(f_i)$.
- $m_1, m_2 \in May : m_3 = m_1 \sqcup_{may} m_2 / \forall f_i \in \mathcal{F} : m_3(f_i) = \min(m_1(f_i), m_2(f_i))$.
- $m_1, m_2 \in May : m_3 = m_1 \sqcap_{may} m_2 / \forall f_i \in \mathcal{F} : m_3(f_i) = \max(m_1(f_i), m_2(f_i))$.

As before and without details, some elements are not valid, therefore we work on a reduced lattice. we have a Galois insertion:

$$(\wp(D), \subseteq, \bigcup) \underset{\gamma_2}{\overset{\alpha_2}{\rightleftarrows}} (May, \sqsubseteq_{may}, \bigsqcup_{may})$$

We define the abstraction function α_2 as:

- $\alpha_2(\phi) = \bot$.
- $\forall x \in \wp(D) : \alpha_2(x) = m \in May / \forall f_i \in \mathcal{F} : m(f_i) =$the minimal age of the function f_i in all concrete cache states.

So, $\alpha_2(x)(f_i) = \omega$ means that f_i *is not* cached, and $\alpha_2(x)(f_i) = k$ means that f_i it will be evicted after at most $NB - k$ references that are not in the abstract cache (or are older or the same age as f_i).

We define the concretisation function as:

- $\gamma_2(\bot) = \phi$.
- $\forall m \in May \backslash \{\bot\} : \gamma_2(m) = \{d \in D / \forall f_i \in \mathcal{F} : \text{age of } f_i \text{ in d } \geq m(f_i)\}$.

The access function: $\widehat{cl'}(m, f_i)$ is done as follows: if $m(f_k) = \omega$ then it is a miss (**M**) else it is an unclassified reference (**U**). The update function is noted $\widehat{\Delta'_i}$.

2.3.3 Collaboration must/may

We can combine these two abstractions with a new connection:

$$(\wp(D), \subseteq, \bigcup) \underset{\bar{\gamma}}{\overset{\bar{\alpha}}{\rightleftarrows}} (Must \times May, \sqsubseteq, \bigsqcup)$$

Where:

$\forall (m_1, m_2), (m'_1, m'_2) \in (Must \times May) : (m_1, m_2) \sqsubseteq (m'_1, m'_2) \Leftrightarrow m_1 \sqsubseteq_{must} m'_1 \wedge m_2 \sqsubseteq_{may} m'_2$.

$\forall x \in \wp(D) : \bar{\alpha}(x) = (\alpha_1(x), \alpha_2(x))$ and $\forall (m_1, m_2) \in Must \times May : \bar{\gamma}(m_1, m_2) = \gamma_1(m_1) \cap \gamma_2(m_2)$.

$\forall m = (m_1, m_2) \in Must \times May : \bar{\gamma}(m) = \gamma_1(m_1) \cap \gamma_2(m_2)$. The two analyses cannot contradict each other. So, can we refine m_1 or m_2?

Example: We take the same cache structure as before; if $m_1 = (1, \omega)$ and $m_2 = (1, 0)$ then:
$\gamma_1(m_1) = \{[f_1(0), \phi], [f_1(0), f_2(1)], , [f_2(1), f_1(0)], [f_2(0), f_1(1)], [f_1(1), f_2(0)]\}$ and
$\gamma_2(m_2) = \{[f_2(0), f_1(1)], [f_1(1), f_2(0)], [f_2(0), \phi], [\phi, \phi]\}$.
So, $\gamma_1(m_1) \cap \gamma_2(m_2) = \{[f_2(0), f_1(1)], [f_1(1), f_2(0)]\}$, therefore we can refine m_1 to
$\alpha_1(\{[f_2(0), f_1(1)], [f_1(1), f_2(0)]\}) = (1, 0)$.

One way to use this collaboration without computing the reduced product, is to collaborate the access functions.
Example: Let an analysis where the concrete semantics is $x = \{[f_1(0), \phi]\}$.
$\gamma_1(\alpha_1(x)) = \{[f_1(0), \phi], [f_1(0), f_2(1)], [f_2(1), f_1(0)]\}$ and $\gamma_2(\alpha_2(x)) = \{[\phi, \phi], [f_1(0), \phi]\}$.
We have $cl(\gamma_1(\alpha_1(x)), f_2) = \mathbf{U}$ but $cl(\bar{\gamma}(\alpha_1(x), \alpha_2(x)), f_2) = \mathbf{M}$.

For LRU, an access of type **U** is equivalent to an access of type **M** in the update function, therefore the previous observation is useless for the LRU policy but not for a FIFO replacement policy (see §2.5).

2.4 Binary Buddy with LRU System

The goal of this section is to illustrate the problem caused by external fragmentation in an LRU system with no constraint on the blocks sizes, and how we can refine the abstract domain to solve the problem. The same refinement approach is followed in the analysis of the next cache management system. Obviously, the non-fixed size blocks will eliminate the problem of internal fragmentation; but in the other side, it is the source of a great amount of external fragmentation. The problem is whether this management system will be predictable or not.

2.4.1 An Imprecise Abstraction

The must-analysis already defined is also correct in this case; correct until the obtaining of the Galois insertion. The problem now is to find the abstraction of the update function. The function $\widehat{\Delta_i} = \alpha_1 \circ \Delta_i \circ \gamma_1$ is no longer precise essentially when the reference to the function f_i is a *miss*.

Example: Let us take $\mathcal{F} = \{f_1, \ldots, f_6\}; s_1 = s_2 = 5, s_3 = s_4 = 10, s_5 = 15, s_6 = 20$ and $C = 35$. Let $m \in Must$ such as $m(f_1) = 0$ and $m(f_i) = \omega / i \neq 1$. In this case, we have not a fixed number NB of blocks, but we can easily obtain a precise upper bound NB on the number of functions can be cached in the same time. In this example $NB = 4$.

So, if the reference to f_6 is a miss, then we have: $\widehat{\Delta_6}(m) = \alpha_1 \circ \Delta_6 \circ \gamma_1(m) = \alpha_1 \circ \Delta_6(\gamma_1(m))$. $\gamma_1(m) = \{d \in D / \text{age of } f_1 \text{ in } d \text{ equals to } 0\}$ is a very large set of concrete cache states, e.g. the function f_1 can take $(C - s_1)$ different locations.

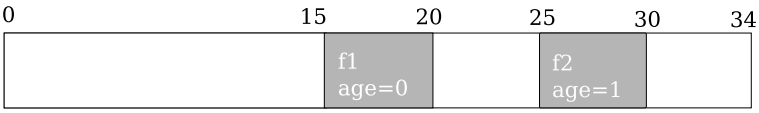


Figure 2.5: A concrete cache state $d \in \gamma_1(m)$

Figure 2.5 presents a concrete cache state $d \in \gamma_1(m)$. The update of this concrete state produces the eviction of f_1. We can no longer say that a reference to f_1 is a hit. Although $m(f_1) = 0$ is far from $NB = 4$, we cannot decide the non-eviction of f_1 because of *external fragmentation*. So, $m' = \widehat{\Delta_6}(m) \wedge m'(f_1) = \omega$, therefore, we have lost the precision very quickly.

The specification $\widehat{\Delta_i} = \alpha_1 \circ \Delta_i \circ \gamma_1$ must be written as an algorithm. If the reference to f_i is classified as a hit (**H**), then there is no problem (it is like the previous abstraction). The problem is when the reference is a miss (or unclassified). In this case, $\forall f_j \in \mathcal{F}, j \neq i \wedge m(f_j) \neq \omega$, we can be in one of the three following states:

- We can predict the **E**viction of f_j (**E**): $\forall d \in \gamma_1(m) : d' = \delta(d, f_i) \wedge f_j$ is not cached in d'.
- We can predict the **N**on-**E**viction of f_j (**NE**): $\forall d \in \gamma_1(m) : d' = \delta(d, f_i) \wedge f_j$ is cached in d'.
- We can**n**ot predict the **N**on-**E**viction of f_j (**NNE**): $\exists d_1, d_2 \in \gamma_1(m) : d'_1 = \delta(d_1, f_i) \wedge d'_2 = \delta(d_2, f_i) \wedge f_j$ is not cached in $d'_1$and cached in d'_2

We need an algorithm which determines the case among those three situations and which avoids enumeration of $\Delta_i(\gamma_1(m))$. In the case of **E** and **NNE**, the age of f_j will be update in $m' = \widehat{\Delta_i}(m)$ to ω. In the case of **NE**, unfortunately, we cannot take $m'(f_j) = m(f_j) + 1$ because if there is a function f_k whose eviction is predicted (**E**), then the previous formula is no longer correct. Therefore, we need to calculate the maximal age of f_j in $\Delta_i \circ \gamma_1(m)$.

As a conclusion of the previous example, the age information is not enough to obtain an efficient analysis; because in most cases, the classification will be **NNE** and not **NE**. The question now is: what we have to do to increase the analysis precision (i.e. increase the number of **NNE**s over **NE**s)?

2.4.2 A Refined but Large Abstract Domain

The cache management system has an LRU replacement policy but without any constraint on the blocks sizes, therefore a function can be placed anywhere. We take $Must = \mathcal{F} \longrightarrow (Age_\omega \times \wp(\mathcal{L}))/\mathcal{L} = \{0, \ldots, C - \min\{s_i, i = 1..n\}\}$. We associate with each function an age and a set of possible locations. So, the abstraction will be:

$$(\wp(D), \subseteq, \bigcup) \underset{\gamma_1}{\overset{\alpha_1}{\rightleftarrows}} (Must, \sqsubseteq_{must}, \bigsqcup_{must})$$

- $m_1, m_2 \in Must : m_1 \sqsubseteq_{must} m_2 \Leftrightarrow \forall f_i \in \mathcal{F} : m_1(f_i).age \leq m_2(f_i).age \wedge m_1(f_i).locations \subseteq m_2(f_i).locations$.
- $m_1, m_2, m_3 \in Must : m_3 = m_1 \sqcup_{must} m_2 / \forall f_i \in \mathcal{F} : m_3(f_i).age = \max(m_1(f_i).age, m_2(f_i).age) \ \wedge \ m_3(f_i).locations = m_1(f_i).locations \cup m_2(f_i).locations$.
- $m_1, m_2, m_3 \in Must : m_3 = m_1 \sqcap_{must} m_2 / \forall f_i \in \mathcal{F} : m_3(f_i).age = \min(m_1(f_i).age, m_2(f_i).age) \ \wedge \ m_3(f_i).locations = m_1(f_i).locations \cap m_2(f_i).locations$.

We define the abstraction function α_1 as:

- $\alpha_1(\phi) = \bot$. We extend $(Must, \sqsubseteq_{must}, \bigsqcup_{must})$ with the least element $\bot$.
- $\forall x \in \wp(D) : \alpha_1(x) = m \in Must / \forall f_i \in \mathcal{F} : m(f_i).age =$the maximal age of the function f_i in all concrete cache states in x, and $m(f_i).locations$ is the set of placement locations of f_i in all the concrete cache states.

We define the concretisation function γ_1 as:

- $\gamma_1(\bot) = \phi$.
- $\forall m \in Must \backslash \{\bot\} : \gamma_1(m) = \{d \in D / \forall f_i \in \mathcal{F} :$ age of f_i in d $\leq m(f_i).age \wedge$ placement location of f_i in $d \in m(f_i).locations\}$.

Is this abstraction a correct approximation? To answer this question, we have to prove that: $\forall x \in \wp(D) : x \subseteq \gamma_1(\alpha_1(x))$.

Proof: We suppose that $\exists x \in \wp(D) : x \not\subseteq \ \gamma_1(\alpha_1(x))$ i.e. $\exists d_0 \in x : d_0 \notin \gamma_1(\alpha_1(x))$.
$d_0 \in x \Rightarrow m = \alpha_1(x) \wedge \forall f_i \in \mathcal{F} :$

$$\begin{cases} m(f_i).age \geq \text{age of } f_i \text{ in } d_0 \\ \text{placement location of } f_i \text{ in } d_0 \in m(f_i).locations \end{cases} \tag{2.1}$$

But, $\gamma_1(m) = \{d \in D / \forall f_i \in \mathcal{F} :$ age of f_i in $d \leq m(f_i).age \wedge$ placement location of f_i in $d \in m(f_i).locations\}$. From (2.1), it is evident that $d_0 \in \gamma_1(m)$. Contradiction □

The gain of the abstraction: This abstraction has increased the precision of the update function. To show the benefit, we take the previous example; but now we have more abstract information than the previous abstract domain. So let, in the previous example, $m(f_1).age = 0$ and $m(f_1).locations = \{30\}$. $\gamma_1(m) = \{d \in D /$age of f_1 in $d = 0 \wedge$ placement location of f_1 in $d = 30\}$. Now, we have more information, so more precision and

we can say that $m'(f_1).age \neq \omega / m' = \widehat{\Delta_6}(m)$.

One can remark easily that this new abstract domain is very huge. Although the termination of the analysis is guaranteed by the theory of abstract interpretation, the fixpoint computation may be very much time-consuming process.

2.4.3 A Possible Solution: Reduced Set of Locations by Binary Buddy

A solution is to reduce the number of placement locations a function can take. To do that, we adopt the binary buddy system approach. There are many classes of buddy systems: binary buddy systems, Fibonacci buddy systems, weighted buddy systems, etc. In [16], Knuth describes the original buddy system (the binary buddy system). The initial memory size has to be a power of 2. If a block of size 2^k is requested, then any available block of size 2^n such that $n \geq k$ (in preference $n = k$) can be split recursively into 2 blocks of the same size, which are called *buddies*, until we obtain the suitable size. When two buddies are again free, they are *coalesced* back into a single block. Only buddies are allowed to be coalesced.

So, the sizes of the functions are rounded up to a size power of 2. The cache size is $C = 2^n$ and the least block size is 2^m (and so $n \geq m$). The number of levels in the system will be $\lambda = n - m$. For a function whose size is rounded up to $2^k/m \leq k \leq n$ (we put $\sigma = k - m$ to indicate that the request needs 2^σ free unit blocks); the possible placement locations will be $\{0, 2^k, 2 \times 2^k, \ldots, (2^{\lambda-\sigma} - 1) \times 2^k\}$ and for abbreviation, we do not take the exact locations but we take $\{0, 2^\sigma, 2 \times 2^\sigma, \ldots, (2^{\lambda-\sigma} - 1) \times 2^\sigma\}$ (Figure 2.6).

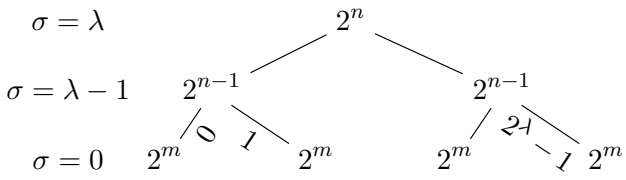
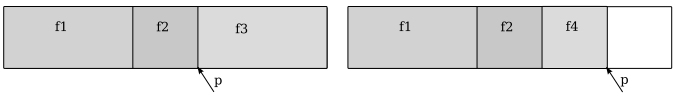


Figure 2.6: A binary buddy system

2.5 BFIFO System

In this cache management system, the cache is considered as a first-in, first-out queue of functions; but the queue does not have a fixed size. Indeed to avoid the internal fragmentation as in the first management system, we choose the non-fixed-size blocks approach. Unlike the LRU policy, the implementation of the FIFO policy is more simple and there is no need to keep information about ages of the functions. We need only a pointer p on the cache which indicate the *current placement location*. For example; the concrete cache state d in Figure 2.7(a) corresponds to the queue $\{f_3, f_1, f_2\}$ such that f_3 is the first-in position. Once the pointer p reaches the cache's right extremity, it will continue from the left one. The update of d when the function f_4 is requested will be d' (Figure 2.7(b)) which corresponds to the queue $\{f_1, f_2, f_4\}$.

- **Advantages:** The implementation is simple and there is neither internal fragmentation nor external one (We suppose that there is no constraint on the sizes of requested blocks).



(a) initial state (b) updated state

Figure 2.7: A FIFO cache

The only external fragmentation is in the case where $(C - p)$ is not enough to hold the requested function, thus the pointer p must jump to the location 0.

- **Drawbacks:** For hardware caches where the blocks have a fixed-size, the performance of LRU policy is better than FIFO. In addition, the WCET analysis of LRU policy is more precise than the one of FIFO and we show why in the following section. However, this comparison between LRU and FIFO needs not be necessarily true in our case where the assumption of blocks sizes is different.

Again, the placement location of a function f_i can be in the interval $[0, C - s_i]$. For an analysis based on abstract interpretation, it is a problem if the abstract domain is large (see §2.4). So, we adopt the binary buddy approach; but the issue with it, is the external fragmentation which can be large. It is worthy to mention that the parameters n and m of the binary buddy are adjustable on the needs of the JIT compiler. In the following, we will explain this new policy and propose a sound analysis.

2.5.1 Binary Buddy FIFO (*BFIFO*) Policy

Conceptually, a BFIFO cache maintains a non-fixed-size queue of functions $\mathcal{F}$. Indeed, the cache can hold at most $2^{\lambda-\sigma}$ $\sigma-$*functions* (a $\sigma-$*function* needs 2^σ free unit blocks, $0 \leq \sigma \leq \lambda$). Therefore, the size of the queue is in $[0, 2^\lambda]$. As we said before, the implementation only needs to keep track on the current placement location (noted as:*cpl*) $p \in \mathcal{L} = \{0, \ldots, 2^\lambda - 1\}$.

Unlike the LRU policy, a cache *hit* in FIFO does not change anything. A cache *miss* inserts the new function f_i at the position p_i and the *cpl* p will be updated to p'. As we have seen in a precedent example, this seems like a shifting of the others functions already in the cache to the right and so the eviction of zero or many functions. More formally:

The update function

Let $pl : D \times \mathcal{F} \longrightarrow \mathcal{L}_\epsilon / \mathcal{L}_\epsilon = \mathcal{L} \cup \{\epsilon\}$ such as $pl(d, f_i)$ is the placement location of f_i in d. If the function f_i is not cached in d, then $pl(d, f_i) = \epsilon$. The cache update function: $\delta : D \times \mathcal{F} \longrightarrow D$ describes the new cache state for a given cache state and a referenced $\sigma-$*function* such that, $\forall d \in D, d' = \delta(d, f_i)$ and:

$$d' = \begin{cases} d & \text{if the access to } f_i \text{ in } d \text{ is a hit} \\ cpl \text{ is } p' \wedge pl(d', f_i) = p' - 2^\sigma & \\ \wedge \forall f_j \in \mathcal{F} : pl(d, f_j) \in [p, p'[\Rightarrow pl(d', f_j) = \epsilon & \text{else} \end{cases}$$

Please note that $[p, p'[$ is not necessary an interval because it may be that $p' < p$ (e.g., $[2^\lambda - 1, 2[$ is equivalent to $\{2^\lambda - 1, 0, 1\}$). Now, we describe how we update p to p':

$$p' = \begin{cases} (p + 2^\sigma) \mod 2^\lambda & \text{if } p \mod 2^\sigma = 0 \\ (p - (p \mod 2^\sigma) + 2^{\sigma+1}) \mod 2^\lambda & \text{else} \end{cases}$$

Remark that the update function is independent from the fragmentation in the concrete cache state. The access function $cl : \wp(D) \times \mathcal{F} \longrightarrow \{H, M, U\}$ is like that of the first management system (LRU with fixed-size blocks).

If we have a sequence of updating $f^* = \{f^1, \ldots, f^n\}$ (a sequence of n functions to be accessed), then we put $\delta^*(d, f^*) = d'/d' = \delta(d_{n-1}, f^n), \ldots, d_2 = \delta(d_1, f^2), d_1 = \delta(d, f^1)$.

Function state (f-state)

The function state of f_i in d is a tuple $fs(d, f_i) = (b, c)/b, c \in \mathcal{L}$ such as b represents the placement location of f_i in the given concrete cache state and c represents the *cpl*. So, the set of f-states will be : $\mathcal{S} = (\mathcal{L} \times \mathcal{L}) \cup \{\omega\}$. The f-state ω means that the concerned function is not cached ($pl(d, f_i) = \epsilon$). $fs : D \times \mathcal{F} \longrightarrow \mathcal{S}$ is a function which associates each function with its f-state in a concrete cache state. When a miss happen, $fs(d, f_i)$ will be updated as follows:

$$fs(d', f_i) = \begin{cases} \omega & \text{if } pl(d', f_i) = \epsilon/d' \text{ is the updated cache state} \\ (b, p') & \text{else} \end{cases}$$

For the new inserted function f_k, $fs(d', f_k) = (pl(d', f_k), p')$.

Intuitively, if a σ-function f_i has $fs(d_1, f_i) = (b, b)$ and $fs(d_2, f_i) = (b, b + 2^\sigma)$, then the updating of d_1 and d_2, when a miss of a $\beta-$function ($\beta < \lambda$) happens, may evict f_i from d_1 but not from d_2. This remark lets us think about constructing a partial order on $\mathcal{S}$.

The partial order $\sqsubseteq_{fs}$: We define $\sqsubseteq_{fs}$ as follows:

- $\forall s \in \mathcal{S} : s \sqsubseteq_{fs} \omega$.
- If $d_1, d_2 \in D$ then $fs(d_1, f_i) \sqsubseteq_{fs} fs(d_2, f_i) \Leftrightarrow \forall f^*$ a sequence of updating (we suppose that the references are misses), $fs(\delta^*(d_1, f^*), f_i) \sqsubseteq_{fs} fs(\delta^*(d_2, f^*), f_i)$

Example: For $\lambda = 2, \mathcal{S}$ is ordered as in the Hasse diagram in Figure 2.8. Two elements in the same line means they are equal (e.g. $(1, 2) =_{fs} (3, 0)$).

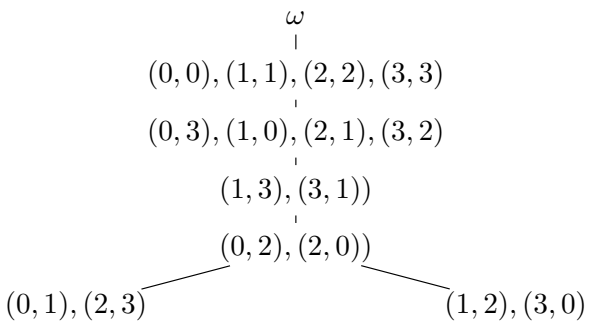


Figure 2.8: An example of $\sqsubseteq_{fs}$ order

$(\mathcal{S}, \sqsubseteq_{fs})$ is a poset but not always a lattice. So, we loose some nice properties but we can always benefit from this partial order because all finite posets verify the ascending chain

condition. Let us show some simple properties of $\sqsubseteq_{fs}$. Let $s = (b, c) \in \mathcal{S}\backslash\{\omega\}$, the number of unit blocks between c and b is the distance:

$$dist(s) = \begin{cases} 0 & \text{if } b = c \\ 2^\lambda - ((c - b + 2^\lambda) \mod 2^\lambda) & \text{else} \end{cases}$$

So, we have the following properties: $\forall b_1, b_2, c_1, c_2 \in \mathcal{L}$,

- $(b_1, b_1) =_{fs} (b_2, b_2)$.
- $(b_1, c_1) \sqsubseteq_{fs} (b_1, c_2) \Leftrightarrow dist((b_1, c_1)) \geq dist((b_1, c_2))$.
- $(b_1, c_1) \sqsubseteq_{fs} (b_2, c_1) \Leftrightarrow dist((b_1, c_1)) \geq dist((b_2, c_1))$.
- $dist(s_1) < dist(s_2) \Rightarrow s_1 \not\sqsubseteq_{fs} s_2$.

A lattice on f-states

Let us take $\mathcal{S}_{must} \subseteq \wp(\mathcal{S})$ such that: $\forall R \in \mathcal{S}_{must}, \forall r \in R, \nexists r' \in R : r' \sqsubseteq_{fs} r$. So, elements of R are incomparable. Such property allow us to get rid of redundant information. Each non-empty element of $\mathcal{S}_{must}$ contains at least one f-state and at most 2^λ f-states.
We extend $\sqsubseteq_{fs}$ on $\mathcal{S}_{must}$ as follow:

$$\forall R_1, R_2 \in \mathcal{S}_{must} : R_1 \sqsubseteq_{fs} R_2 \Leftrightarrow \forall r_1 \in R_1, \exists r_2 \in R_2 : r_1 \sqsubseteq_{fs} r_2$$

It is evident that this relation is reflexive and transitive, but we must prove that it is antisymmetric:$\forall R_1, R_2 \in \mathcal{S}_{must} : R_1 \sqsubseteq_{fs} R_2 \Rightarrow \forall r_1 \in R_1, \exists r_2 \in R_2 : r_1 \sqsubseteq_{fs} r_2$; but if $R_2 \sqsubseteq_{fs} R_1$ then $\exists r_1' \in R_1 : r_2 \sqsubseteq_{fs} r_1'$. So, $r_1 \sqsubseteq_{fs} r_2 \sqsubseteq_{fs} r_1'$; but elements of R_1 are incomparable, therefore $r_1 =_{fs} r_2$. □

To restrict an element of $\wp(\mathcal{S})$, we use the function $rest_{must} : \wp(\mathcal{S}) \longrightarrow \mathcal{S}_{must}$. We put: $\forall R_1, R_2, R_3 \in \mathcal{S}_{must}, R_3 = R_1 \sqcup_{fsmust} R_2 = rest_{must}(R_1 \cup R_2)$. $\sqcup_{fsmust}$ is a binary least upper bound. So, $(\wp(\mathcal{S})_{must}, \sqsubseteq_{fs}, \bigsqcup_{fsmust})$ is complete lattice with $\{\omega\}$ as the top element and ϕ as the bottom one.
In the same way, we construct a complete lattice $(\mathcal{S}_{may}, \sqsupseteq_{fs}, \bigsqcup_{fsmay})$ such that: $\forall R \in \mathcal{S}_{may}, \forall r \in R, \nexists r' \in R : r \sqsubseteq_{fs} r'$.

2.5.2 Must Analysis

In a hardware cache with a FIFO replacement policy, k misses (k is the cache size) must happen to evict newly inserted reference. This rule is no longer true in our case because of the non-fixed-size blocks. So, the FIFO analysis dedicated for hardware caches [11] cannot be applied in this context.
Let us take the abstract domain: $Must = (\mathcal{F} \longrightarrow \mathcal{S}_{must}) \times \wp(\mathcal{L})$. The part $\wp(\mathcal{L})$ is called *pointers*, it is used to express the possible current placement locations. So,

- $m_1, m_2 \in Must : m_1 \sqsubseteq_{must} m_2 \Leftrightarrow \forall f_i \in \mathcal{F} : m_1(f_i) \sqsubseteq_{fs} m_2(f_i) \wedge m_1.pointers \subseteq m_2.pointers$
- $m_1, m_2, m_3 \in Must : m_3 = m_1 \sqcup_{must} m_2 / \forall f_i \in \mathcal{F} : m_3(f_i) = m_1(f_i) \sqcup_{fsmust} m_2(f_i) \wedge m_3.pointers = m_1.pointers \cup m_2.pointers$.

- $m_1, m_2, m_3 \in Must : m_3 = m_1 \sqcap_{must} m_2 / \forall f_i \in \mathcal{F} : m_3(f_i) = m_1(f_i) \sqcap_{fsmust} m_2(f_i) \wedge m_3.pointers = m_1.pointers \cap m_2.pointers$.

$(Must, \sqsubseteq_{must}, \bigsqcup_{must})$ is a complete lattice. The question now is: can we find a Galois connection

$$(\wp(D), \subseteq, \bigcup) \underset{\gamma_1}{\overset{\alpha_1}{\rightleftarrows}} (Must, \sqsubseteq_{must}, \bigsqcup_{must})$$

to be able of using abstract interpretation?
We define the abstraction function α_1 as:

- $\alpha_1(\phi) = \bot$. As usual, we extend *Must* by $\bot$ and we reduce it to just the valid elements.
- $\forall x \in \wp(D) : \alpha_1(x) = m \in Must / m.pointers = \bigcup \{cpl \text{ of } d / d \in x\} \wedge \forall f_i \in \mathcal{F} : m(f_i) = \bigsqcup_{fsmust} \{fs(d, f_i) / d \in x\}$.

and the concretisation function γ_1 as:

- $\gamma_1(\bot) = \phi$.
- $\forall m \in Must : \gamma_1(m) = \{d \in D, cpl \text{ of } d \in m.pointers \wedge \forall f_i \in \mathcal{F}, \exists s \in m(f_i) : fs(d, f_i) \sqsubseteq_{fs} s\}$

This abstraction contains the following information:

- The number of misses a function needs to be evicted. It is somewhat similar to the age information in the first cache management system.
- An f-state does not only abstract the number of misses but it abstracts also the location information.
- To increase the precision of the analysis, we have added a new information: *the current location placement*. This is possible because of the FIFO policy where the placement is independent from the *external fragmentation*.

Remark: We can take the abstract domain as $Must = ((\mathcal{F} \longrightarrow \mathcal{S}) \times \wp(\mathcal{L})) \cup \{\bot\}$ for a fast must-analysis but we will loose some precision.

The abstraction of the access function

As before, the approximation of cl is $\widehat{cl} = cl \circ \gamma_1$. The access type of a reference to f_i (f_i is a $\sigma-function$) is done as follows: Let $m \in Must \backslash \{\bot\}$, if $m(f_i) \neq_{fs} \{\omega\}$ then it is a hit (**H**); else if $(\sum_{f_j \in \mathcal{F} : m(f_j) \neq_{fs} \{\omega\} \wedge f_j \text{ is a } \sigma_j - function} 2^{\sigma_j}) + 2^{\sigma} > 2^{\lambda}$ the reference is a miss (**M**); else it is an unclassified reference (**U**).

The abstraction of the update function

The best approximation of Δ_i is given by the specification $\widehat{\Delta}_i = \alpha_1 \circ \Delta_i \circ \gamma_1$. Let $m, m' \in Must\backslash\{\bot\}, f_i$ a $\sigma - function$ $: m' = \widehat{\Delta}_i(m)/$

$$m' = \begin{cases} m & \text{if } \widehat{cl}(m, f_i) = \mathbf{H} \\ \forall f_j \in \mathcal{F} : m'(f_j) = \bigsqcup_{fsmust}\{s' = \text{the updating of } s \\ \text{ when a } \sigma - function \text{ is loaded}/s \in m(f_j)\} & \text{if } \widehat{cl}(m, f_i) = \mathbf{M} \text{ or } \mathbf{U} \\ m'.pointers = \bigcup\{p' = \text{the updating of } p \text{ when a} \\ \sigma - function \text{ is loaded}/p \in m.pointers\} & \text{if } \widehat{cl}(m, f_i) = \mathbf{M} \\ m'.pointers = \bigcup\{p' = \text{the updating of } p \text{ when a} \\ \sigma - function \text{ is loaded}/p \in m.pointers\} \cup m.pointers & \text{if } \widehat{cl}(m, f_i) = \mathbf{U} \end{cases}$$

If $\widehat{cl}(m, f_i) = \mathbf{U}$, here there is a difference between LRU and FIFO and we explain why the LRU policy is more predictable than the FIFO one [29, 11]. In an LRU cache, if we access a reference f (a hit) then a reference g (a miss), one can predict that f is always cached; but this needs not be necessary true with a standard FIFO policy, specifically when the hit on f was on the right-most position in the queue. So:

$$m' = \begin{cases} m'(f_i) = \bigsqcup_{fsmust}\{(\bar{p}, p')/p \in m.pointers \wedge p' = \text{the updating of } p \\ \text{ when a } \sigma - function \text{ is loaded} \wedge \bar{p} = p' - 2^{\sigma}\} & \text{if } \widehat{cl}(m, f_i) = \mathbf{M} \\ m'(f_i) = \{(0, 0)\} & \text{if } \widehat{cl}(m, f_i) = \mathbf{U} \end{cases}$$

The f-state $(0, 0)$ represents a last chance to f_i to be cached (it is like a right-most position in a standard FIFO) and which is a worst case.

2.5.3 May analysis

Because of the different reaction of $\widehat{\Delta}_i$ towards a reference of type (**M**) and a reference of type (**U**), one can collaborate the may-analysis with the must-analysis to increase the number of (**M**) accesses over the (**U**) accesses. The abstract access function of may-analysis is as follow: let $m \in May\backslash\{\bot\}$, if $m(f_i) = \{\omega\}$ then the access to f_i is a miss else it is unclassified.

2.6 Fixed Layout System

In this approach, the function layout is fixed statically at compilation. Each function is given a unique location. In some ways, this approach is equivalent to a direct-mapped hardware cache, except that blocks do not have a fixed-size. This approach is done in two steps: proposing a must-analysis (the may-analysis is useless) and a method to compute the function layout. First, we start with the analysis.

Let $pl : \mathcal{F} \longrightarrow \mathcal{L}/\mathcal{L} = \{0, \ldots, C - \min\{s_i/i = 1..n\}\}$ a function which associates with each function f_i a unique placement location $pl(f_i)$; and let the set of concrete cache states $D \in \wp(\mathcal{F})/\forall d \in D, \forall f_i, f_j \in d : \nexists c, pl(f_i) \leq c < pl(f_i) + s_i \wedge pl(f_j) \leq c < pl(f_j) + s_j$. In a concrete cache state, two functions never overlap.
We define a must-analysis such that: the abstract domain is $Must = \wp(\mathcal{F})$. So,

- $m_1, m_2 \in Must : m_1 \sqsubseteq_{must} m_2 \Leftrightarrow m_2 \subseteq m_1$.

- $m_1, m_2, m_3 \in Must : m_3 = m_1 \sqcup_{must} m_2 / m_3 = m_1 \cap m_2$.
- $m_1, m_2 \in Must : m_3 = m_1 \sqcap_{must} m_2 / m_3 = m_1 \cup m_2$.

So,$(Must, \sqsubseteq_{must}, \bigsqcup_{must})$ is a complete lattice. We define the abstraction function α_1 as:

- $\alpha_1(\phi) = \bot$. We extend $Must$ by a least element $\bot$.
- $\forall x \in \wp(D) \backslash \{\phi\} : \alpha_1(x) = \bigcap_{d \in x} d$.

We define the concretisation function as:

- $\gamma_1(\bot) = \phi$.
- $\forall m \in Must \backslash \{\bot\} : \gamma_1(x) = \{d \in D, m \subseteq d\}$.

The update function : Let $\delta : D \times \mathcal{F} \longrightarrow D$ the update function such that:

$$d' = \delta(d, f_i) = \begin{cases} d & \text{if } f_i \in d \\ f_i \in d' \wedge \forall f_k \in d, \text{ if } \exists c, pl(f_k) \leq c < pl(f_k) + s_k, & \\ pl(f_i) \leq c < pl(f_i) + s_i, \text{ so } f_k \notin d' & \text{else} \end{cases}$$

The extension of δ is $\forall x \in \wp(D) : \Delta_i(x) = \{d' / \forall d \in x : d' = \delta(d, f_i)\}$. The abstraction of Δ_i is $\widehat{\Delta}_i$ such that: $\forall m \in Must \backslash \{\bot\}$,

$$m' = \widehat{\Delta}_i(m) = \begin{cases} m & \text{if } f_i \in m \\ f_i \in m' \wedge \forall f_k \in m, \text{ if } \exists c, pl(f_k) \leq c < pl(f_k) + s_k, & \\ pl(f_i) \leq c < pl(f_i) + s_i, \text{ so } f_k \notin m' & \text{else} \end{cases}$$

2.6.1 Layout Computation

The method to compute the layout comes in three steps:

- Conflicts costs determination: The goal of this step is to determine for each two functions $f_i, f_j \in \mathcal{F}$ the overhead in execution time causes by f_i overlap with f_j.
- Conflicts determination: The objective is to minimize the overhead caused by all the conflicts.
- Layout determination: For each function f_i, this step determines a suitable placement $pl(f_i)$ w.r.t. the second step.

Conflicts costs determination

This problem has been tackled many times. Code positioning is a classic problem [27] which attempts to reorder the code to reduce some overheads for example instruction cache conflicts. Usually, this process is guided by the execution profile but in our case it is guided by call frequencies on the WCET path because we try to minimize the worst-case and not the average case. Our conflict cost determination is a modified version of the method presented in [10]. The method takes as input the call graph annotated with frequencies (e.g. Figure 2.9). An edge $(f_1 \rightarrow f_3)$ means that f_1 calls f_3 with a frequency $l(f_1, f_3) = 2$. Those frequencies are obtained by a WCET estimation of the program using the IPET technique (see §1.3.1) without

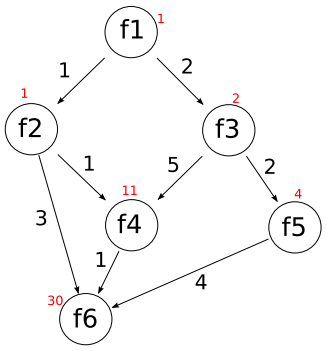


Figure 2.9: An example of a call graph

considering the existance of the JIT compilation. In the call graph, We do not differentiate between call points, e.g. if f_1 calls twice f_3, then the frequency $l(f_1, f_3)$ will be the sum. A first step in the method is to compute the global frequency of each node (e.g. in Figure 2.9: $G(f_6) = 30$):

$$G(f_i) = \begin{cases} 1 & \text{if } f_i \text{ is the entry function} \\ \sum_{f \in predecessor(f_i)} G(f_i) l(f, f_i) & \text{else} \end{cases}$$

The second step is to compute for each node f_i its conflict cost with its neighbors. The neighborhood is the union of two sets:

- **Childhood**: Nodes reachable from f_i following the child relationship in the call graph. The maximum distance is specified by a fixed parameter m_1. e.g. f_3 is a child of f_1 of level 1, and f_6 is a child of level 2 or 3 it depends on the path. So, if we have a feasible path $f_i f_1 f_2 \cdots f_n f_j$, the conflict between f_i and f_j is equal to: $c_{ij} = G(f_i) l(f_i, f_1)(M(f_i) + M(f_j))$; such that $M(f)$ is the overhead caused by a miss of the function f i.e. the worst-case compilation time(WCCT). For example (Figure 2.9), if f_1 overlaps with f_3, then the expansion of the calls/returns series will give $f_1\ f_3\ f_1\ f_3\ f_1$, that generates $2 = G(f_1).l(f_1, f_3)$ evictions of f_1 and 2 evictions of f_3. We must take the contribution of all feasible paths. One important remark is that a conflict has more chance to happen when the level of the child is small. So, the contribution of a path has to be multiplied by $K_1^{level-1}/K_1 \leq 1$.
- **Brotherhood**: Nodes reachable form f_i following the brother relationship in the call graph. The maximum distance is specified by a parameter m_2. e.g. f_2 is a brother of f_3 of level 2, and f_4 is a brother of f_6 of level 4 if we consider that f_4 is the child of f_3 and f_6 is the child of f_2. If we have two feasible paths $f_k f_1 \cdots f_n f_i$ and $f_k f'_1 \cdots f_j$ such that $f_1 \neq f'_1$, then $c_{ij} = G(f_k) \min[l(f_k, f_1), l(f_k, f'_1)](M(f_i) + M(f_j))$. This is an overapproximation value since the exact value depends to the control flow structure (e.g. loops). The contribution of this conflict has to be multiplied by $K_2^{level-2}/K_2 \leq 1$

Conflicts determination

Let us take the set of $0/1-$variables $\{x_{ij}/i, j = 1..n \wedge i < j\}$ such that:

$$x_{ij} = \begin{cases} 1 & \text{if } f_i \text{ overlaps with } f_j \\ 0 & \text{else} \end{cases}$$

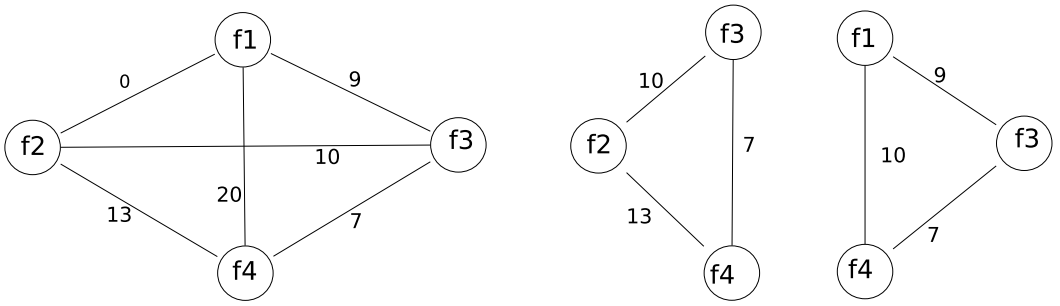
The total overhead is a dynamic property. Statically, we try to minimize $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} c_{ij}.x_{ij}$. We propose here to use an ILP (integer linear programming) formalism to solve the problem when n is not a big number, and an heuristic else.

An ILP solution: The major constraint here is posed by the cache size C. For example, if we have $\mathcal{F}' = \{f_i, f_j, f_k\}$ such that $s_i + s_j + s_k > C$ then there is at least one conflict between two functions in $\mathcal{F}'$. Formally, the problem is:

$$\min Z = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} c_{ij}.x_{ij} \quad \text{subject to } \forall \mathcal{F}' \in \wp(\mathcal{F}) : \sum_{f\in\mathcal{F}'} Size(f) > C \Rightarrow \sum_{f_i,f_j\in\mathcal{F}'\wedge i<j} x_{ij} > 0.$$

An important remark is that if $\mathcal{F}_1 \subseteq \mathcal{F}_2$ such that $\sum_{f\in\mathcal{F}_1} Size(f) > C$ then obviously $\sum_{f\in\mathcal{F}_2} Size(f) > C$. In this case, it is useless to generate two constraints; the constraint for $\mathcal{F}_1$ is enough. Finding all the minimal constraints is like searching for a cut in the lattice $(\wp(\mathcal{F}), \subseteq, \bigcup)$. Therefore, there are at most $\binom{n}{\frac{n}{2}}$ constraints.

An heuristic solution: We model the problem as a graph problem. The functions in $\mathcal{F}$ represent nodes. An edge between two nodes f_i and f_j means that there is no conflict between the two functions. At beginning there is no conflict at all (the graph is complete). each edge is labeled by the conflict cost.
Example: Let $\mathcal{F} = \{f_1, f_2, f_3, f_4\}$ and $s_1 = 15, s_2 = 20, s_3 = 10, s_4 = 5$ and $C = 30$. The conflicts costs are represented in Figure 2.10(a).



(a) The initial state (b) Removing edge $(f_1 - f_2)$

Figure 2.10: An example of the heuristic solution

The size constraint is modeled now as: each maximal clique Q (Q is maximal if there is no clique Q' such that $Q \subset Q'$) in the graph has a size $s = \sum_{f\in Q} Size(f) < C$. At the beginning there is only one clique. We choose to remove the edge $(f_i - f_j)$ with the *minimum* conflict cost. This yield to exactly two maximal cliques ($Q - \{f_i\}$ and $Q - \{f_j\}$, see Figure 2.10(b)). If one of the two cliques does not verify the size constraint, we choose another edge and so on (be careful on shared edge, in Figure 2.10(b) edge $(f_3 - f_4)$ is shared between the two cliques).

Layout determination

After we determine the conflicts, we can do the analysis without computing the exact placements of functions (we do not need this information in the analysis). A simple way to compute

the layout is to use the ILP formalism. We take the set of integer variables $pl(f_i)/f_i \in \mathcal{F}$. So, $\forall f_i, f_j \in \mathcal{F}, i < j$:

$x_{ij} = 1 \Rightarrow pl(f_j) \leq pl(f_i) < pl(f_j) + s_j \quad \vee \quad pl(f_i) \leq pl(f_j) < pl(f_i) + s_i$
$\Rightarrow 0 \leq pl(f_i) - pl(f_j) \leq s_j - 1 \quad \vee \quad 0 \leq pl(f_j) - pl(f_i) \leq s_i - 1$
$\Rightarrow (1 - s_i)(1 - y_{ij}) \leq pl(f_i) - pl(f_j) \leq (s_j - 1)y_{ij}$. Such that y_{ij} is a $0/1$-variable. So, we have two linear constraints.

$x_{ij} = 0 \Rightarrow pl(f_i) + s_i \leq pl(f_j) \quad \vee \quad pl(f_j) + s_j \leq pl(f_i)$
$\Rightarrow s_i \leq pl(f_j) - pl(f_i) \leq C \quad \vee \quad s_j \leq pl(f_i) - pl(f_j) \leq C$
$\Rightarrow s_j y'_{ij} - C(1 - y'_{ij}) \leq pl(f_i) - pl(f_j) \leq C y'_{ij} - s_i(1 - y'_{ij})$. Such that y'_{ij} is a $0/1$-variable. So, we have two linear constraints.

In summary, we have $\frac{n(n-1)}{2}$ conflicts determined by the second step, each conflict generates two constraints. $\forall f_i \in \mathcal{F}$, we generate another constraint $pl(f_i) \leq C - s_i$. So, we have in total n^2 linear constraints. The objective function in this problem is constant.

Chapter 3

Experimental Results

After we have presented different cache management systems in the previous chapter, we present here some results. It is worthy to remember that the JIT compiler can implement more than one cache management system. The programmer decides statically which system he will use after he analyzes his program. The objective of the following results is not necessarily a *strict* comparison between the three systems since each system can be the best in some situations as we will see later. We start by comparing the average case performance of some cache management systems, then discussing the heuristic used to compute the layouts, and finally comparing the predictability of the three systems.

3.1 Average Case Performance

In this section, we present the result of a simulation of two management systems: fixed-size blocks with LRU and BFIFO. The performance of the fixed-layout system is depending to the computed layout. In each simulation, we randomly generated a sequence of $200 * nbf$ accesses to nbf functions and we calculated the number of hits. LRU1 and FIFO1 are obtained for a cache of size $C = 1KB$, LRU2 and FIFO2 for a cache of size $C = 4KB$, and finally LRU3 and FIFO3 for a cache of size $C = 16KB$. Figure 3.1 depicts the results of the simulation. It represents the percentage of hits among the $200*nbf$ accesses in each situation. The sizes of functions are uniformly distributed in the interval $[10, maxF * C]$ such that $0 < maxF \leq 1$. The parameter $maxF$ controls the heterogeneity of the sizes of functions and the number of blocks NB of the LRU system. So, when $maxF$ increases, NB decreases.

We can easily remark that:

- LRU with fixed-size blocks has a bad performance when $maxF$ tends to 1 because the number of blocks tends also to 1.
- The BFIFO system is insensitive to the variation in $maxF$.
  From case (C), if $maxF > 0.01$ ($NB < 100$), then the BFIFO system is better than LRU in terms of average case performance.

We have also remarked that the bigger is λ (the depth of the binary buddy system), the better is the performance of the BFIFO system.

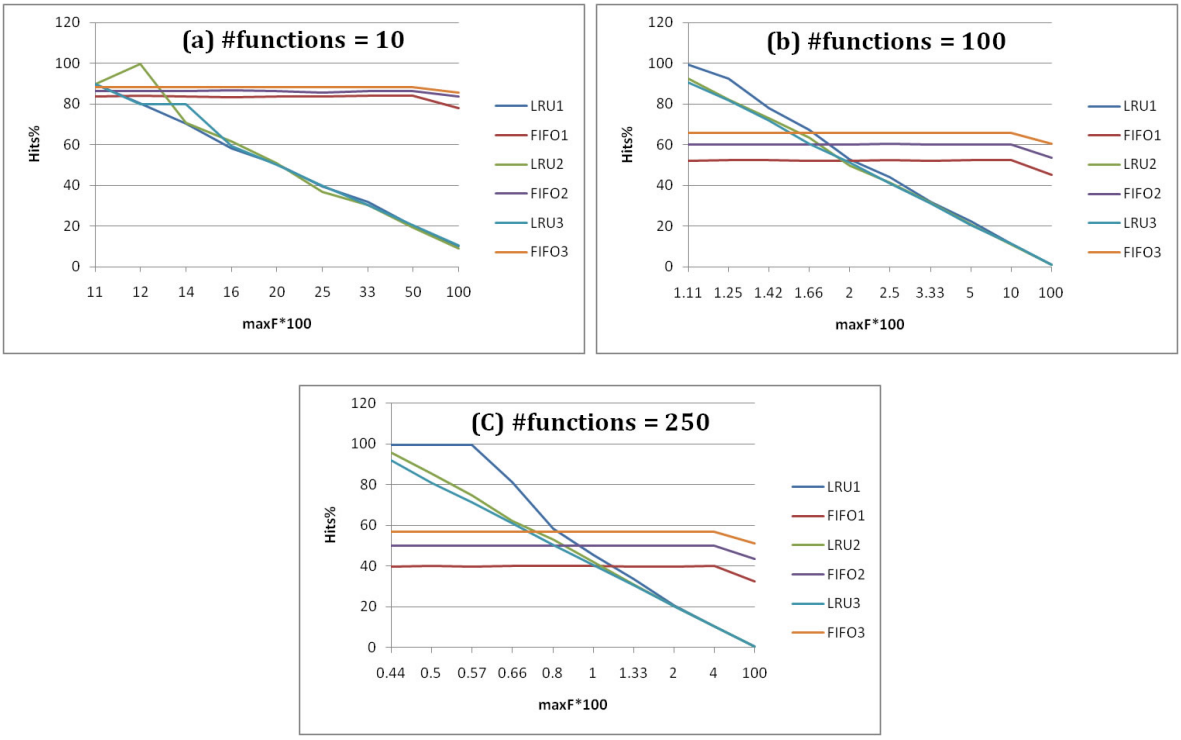


Figure 3.1: Simulation results

3.2 Conflicts Determination

As expected, the ILP solution is very time consuming but provides a precise solution. Because we are interested in programs with many functions, the following results are obtained by the heuristic solution. The heuristic is parameterized by the maximum number of edges ($maxR$) we can remove at each iteration. Edges to remove are chosen from cliques which do not satisfy the size constraint. From each clique, we choose the edge with the minimum conflict cost, then the obtained edges are ordered and the first $maxR$ edges are chosen to be removed at this iteration. If the algorithm does not terminate in a period of ($time$) minutes, it switches to a simple strategy: for each clique already obtained (i.e. a clique which does not satisfy the size constraint), the nodes are packed in sets, such that the size of a set is less than the cache size and there is no conflict among the nodes of this set. For all two nodes in different sets, a conflict will be generated.

Figure 3.2 presents the total conflict cost (as a percentage relative to the maximal total conflict cost when everybody is on conflict with everybody) of the layout of 25 functions for different configurations. We have chosen only 25 functions because it is a case when the heuristic with $maxR$=1 terminates in a reasonable time. The sizes of the functions are uniformly distributed in the interval $[10, maxF * C]$. C is the cache size and it is taken as 16 KB. It is easy to remark that:

- The total conflict cost decreases when $maxF$ decreases and which is expected since the sizes of the functions are decreased (sizes are homogenous) and so the number of conflicts.
- The time to find a solution is shorter when $maxR$ is bigger. The reason is that at each

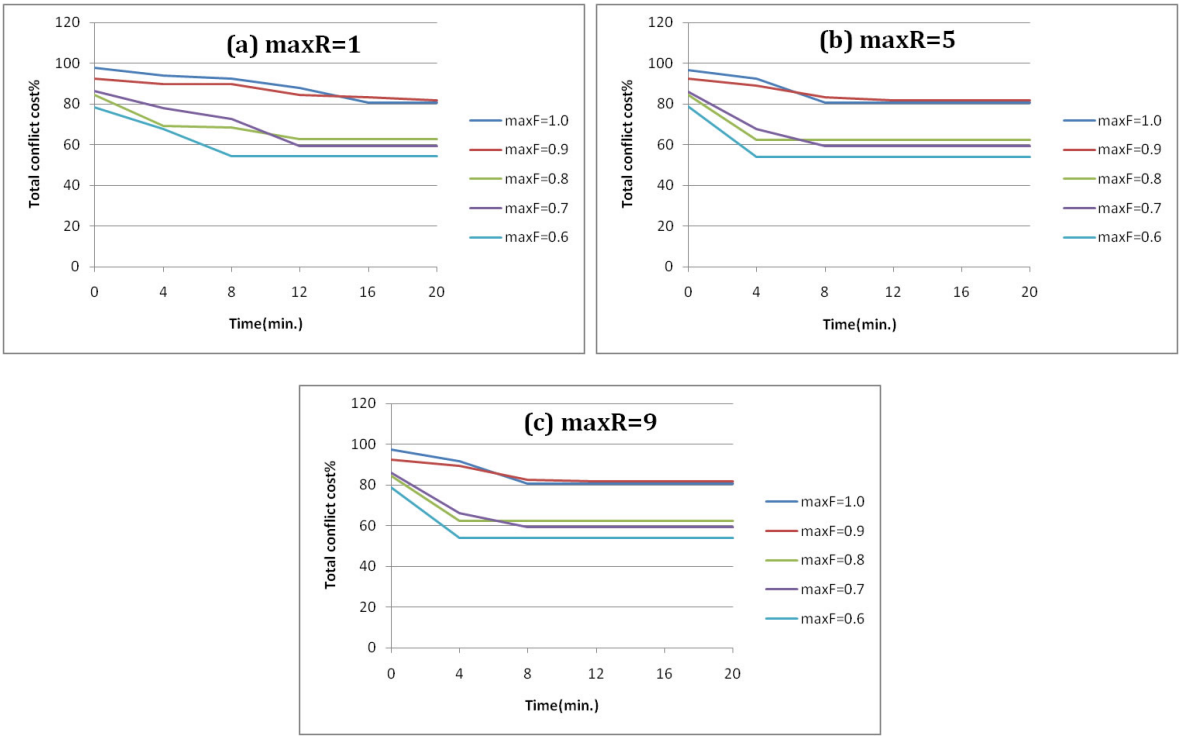


Figure 3.2: Behavior of conflict determination heuristic

iteration, a clique has a bigger chance of satisfying the size constraint after we remove many edges than when we remove only one edge.

- The time to find a solution is shorter when *maxF* is smaller, because the number of cliques which do not verify the size constraint is smaller. So, when *maxF* tends to 1, it is better if we increase *maxR* and decrease *time* to find quickly the solution.

3.3 Cache Analysis

The experiments were conducted on a subset of subtasks of the *debie1* benchmark [1] which is based on the on-board software of the DEBIE instrument for measuring impacts of small space debris or micro-meteoroids. The subtasks are compiled for a MIPS R2000/R3000 architecture with gcc 4.1 with no optimization, and we have used the *Salto* tool [3] to extract the CFGs. Table 3.1 represents the characteristics of the subtasks. *max size* is the maximal size of functions, *avg size* is the average of sizes of functions and *#calls&returns* is the number of call and return nodes in the program for all possible contexts. The average is quite far from the maximal function size.

3.3.1 Results

Figure 3.4 represents the percentage of calls&returns (relative to #calls&returns) which are classified as always hits, and the time of the analysis in minutes. The number of accesses to

[1] debie1 benchmark http://www.mrtc.mdh.se/projects/WCC08/doku.php?id=bench:debie1

Name	#functions	Total size	max size	avg size	#calls&returns
Hit_ISR	71	4053	612	57.08	1452
TC_ISR	76	4167	612	54.82	1492
TM	96	7000	1263	72.91	2294
Monitoring_Task	132	9119	1263	69.08	16170
Su_Self_Test	139	9677	1263	69.62	20336

Table 3.1: Benchmarks characteristics

be classified ranges from 1492 in *Hit_ ISR* to 20336 in *Su_ Self_ Test*. FLRU is the notation of the fixed-size blocks with LRU management system. For the BFIFO system, the results are obtained for $\lambda = 2$, this choice is arbitrary and it is not the best as we will see later. For the fixed-layout system, the conflicts are determined by taking $maxR = 6$ and $time = 10$ minutes. The worst-case compilation time (WCCT) is taken to be linear to the size of the obtained binary code, because the WCCT analysis is quite complicated and it is not done in this first solution. Figure 3.3 represents the percentage of the number of conflicts (relative to the total number of conflicts) in each situation. The number of conflicts decreases when the cache size increases as expected. The worst case is 37.10% and which is actually a low percentage, the reason is that there are many functions with a small size (see Table 3.1).

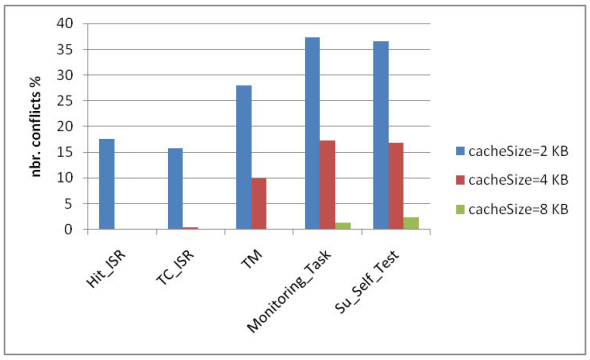


Figure 3.3: The number of conflicts for fixed-layout

From this cache analysis, we can remark:

- Each system of the three management systems can be the best in a given situation, therefore the JIT compiler may implement the three systems.
- The BFIFO analysis with $\lambda = 2$ usually takes more time than the two others analyses, because there are two analyses (must and may-analysis) and the update function is more complicated than ones of the other systems. The analysis time of the fixed-layout system, depicted in Figure 3.4, does not include the time spent in computing the layout.
- The fixed-layout system provides the best results when the number of conflicts tends to 0, else it is bad although when the number of conflicts is only 10% (case of *TM, Monitoring_ Task* and *Su_ Self_ Test*). So, reducing the number of conflicts does not mean necessarily improving in the same rate the predictability of the fixed-layout system.
- When the cache size increases, the number of blocks for the FLRU system increases and the number of conflicts for the fixed-layout system decreases. Therefore the analysis

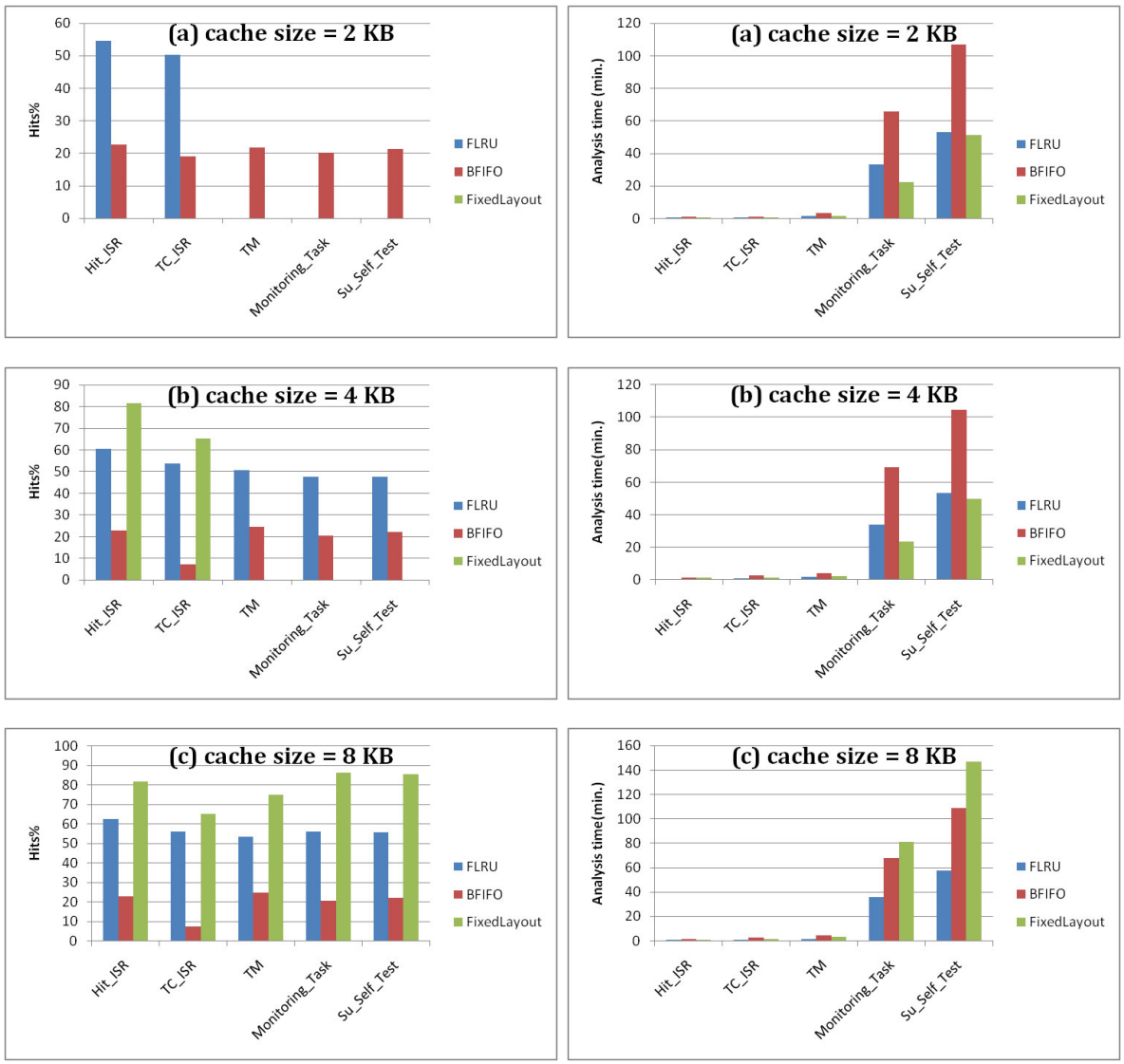


Figure 3.4: Cache WCET analysis results

time increases and the results get better and better. We can remark this difference from cases (a) and (c). For example, in case (a), the number of blocks for the subtask *Monitoring_ Task* is 1 ($hits\% = 0$); but in case (c), NB=6 ($hits\% = 56.14$).

- The BFIFO system is the best when the cache size is equal to 2 KB because for the FLRU system there is only one block and for the fixed-layout there are many conflicts.

3.3.2 Effect of λ

All the previous results were expected. The only surprise for us was when we tried to observe the effect of the variation of λ on the number of always hits and the time of the analysis (see Figure 3.5). $C1$ is the case of TC_ISR with $C = 2$ KB and $C2$ is the case of TM with $C = 2$ KB. We remarked that:

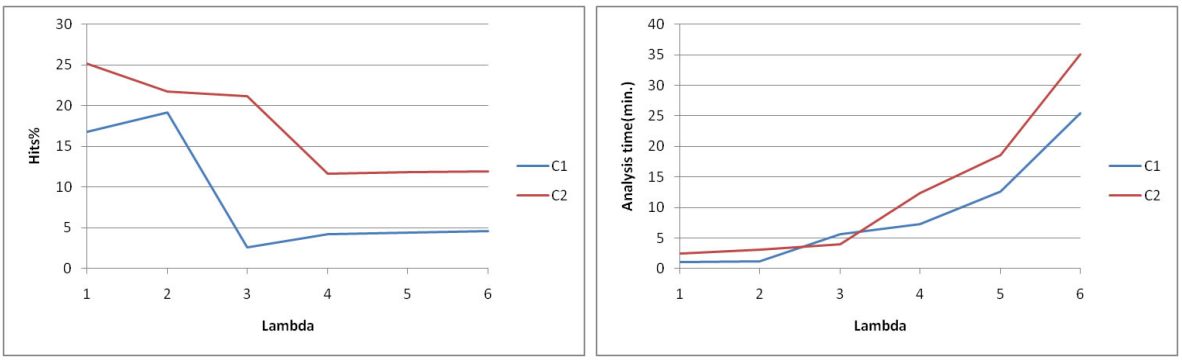


Figure 3.5: The effect of lambda

- Generally, there is no rule, and we cannot say the bigger λ it is, the better the predictability of the BFIFO system will be, as in the average case.
- The time of the analysis increases when λ increases. This is expected since the number of unit blocks increases and so does the number of *f-states*.

3.4 Perspectives

The work done in this internship is a first solution. In this section, we present some perspectives.

WCET computation

The classification is not enough for comparing between the three analyses. Indeed, if an analysis A of a program gives 20 references classified as always hit, and an analysis B of the same program gives only 10; this does not mean necessarily that A is better than B unless if we compute the WCET of the whole task. This step is not yet implemented because of the complexity caused by the JIT compilation for the usual analysis of hardware caches. Indeed, this last analysis does not take in consideration the ability of removing a function from the memory and then invalidating the contents of the instruction cache. We need to modify the existent tools to consider this requirement.

In addition, we have to know the worst compilation time of each method. This step is not easy and it is not done yet.

Loop problem

In the three analyses, if a call node to a function f situated in a loop and $\widehat{cl}(AC_{in}(loop\ entry), f) = U/M$, then at the end of the analysis the classification will be (U). But if we separate the first iteration of the loop from the other $n-1$ iterations, one may predict the presence of the function in the cache for the $n-1$ iterations. To do that without changing the previous analysis we have just to peel the first iteration of each loop in the program's CFG. In hardware cache analysis, this problem is solved by adding a third analysis: a persistence-analysis [33].

CFG reduction

As one can remark, the original CFG of a subtask is huge and many nodes have no effect (ordinary nodes). So, we can reduce the CFG (e.g. a loop without any call node can be reduced to one ordinary node) but we must be careful of keeping the control flow intact. This step may reduce the time of the analyses a lot.

Alternative to the LRU with fixed-size blocks system

If the sizes of the functions are heterogeneous, then there will be a performance degradation due to internal fragmentation. One direct reflection is to divide the cache to m LRU subcaches cache= $\{SC_1, \ldots, SC_m\}$. Each subcache SC_i has NB_i blocks, and each block in this subcache has a fixed size SB_i. A first constraint is : $\sum_{i=1}^{m} NB_i \times SB_i \leq C$ and $C - \sum_{i=1}^{m} NB_i \times SB_i$ is minimal.
The analyses of subcaches are independent from each other because each subcache SC_i has each own subset of functions $\mathcal{F}_i \subseteq \mathcal{F}$. A second constraint is that the subsets form a partition of $\mathcal{F}$ and $\forall f \in \mathcal{F}_i : Size(f) \leq SB_i$.
The objective here is to reduce the overall internal fragmentation that is a dynamic property. If we look for a global static view, we can take this sum: $\sum_{i=1}^{m}(\sum_{f \in \mathcal{F}_i} SB_i - Size(f))$.
Another objective is to increase the predictability of the caches analyses. One idea is that if a function f_1 calls frequently a function f_j, then it is better if the two functions will be assigned to two different subcaches.
As you can remark, too many variables $(m, NB_i, SB_i, \ldots)$ to puzzle out and conflicting objectives, therefore this idea is not further developed.

Conclusion

This document begins with a brief overview of the JIT compilation and methods to perform timing analysis and which is a crucial job in the design of a hard real-time system. The JIT compilation is used in a virtual machine such as the JVM to reduce the overhead caused by the interpretation process.

The JIT compilation is a technique where a method will be compiled only when it is used. So, the compilation time is no longer separated from the execution time. This fact is a serious threat for the traditional timing analysis context. The JIT compilation features many dynamic characteristics and techniques such as adaptive optimization and inlining. Those advanced techniques are used to achieve more performance.

Our work is a first attempt to use the JIT compilation in a hard real-time system. We consider a simple case where the system is mono-task and there is no use of advanced techniques such as adaptive optimization. The problem seems simple but to know statically the overhead caused by the compilation time is not an easy job and we have to predict whether a call to a function launch the JIT compiler or not.

Embedded systems have usually many resources constraints such as a limited amount of memory. So, the compiled code has to be put in a memory zone which acts as a software cache. Accordingly to the cache management system, the timing analysis provides different results. In this document, we have proposed three management systems: LRU with fixed-size blocks, binary buddy FIFO (BFIFO) and fixed-layout management system.

The experimental result show that each system can be the best in some situations, therefore the JIT compiler may implement the three systems. In hardware caches, the predictability of FIFO policy has been proven to be worse than the one of LRU policy. But in our context, we have seen that the BFIFO is the best when the sizes of functions are heterogenous and the number of fixed-size blocks tends to 1 in the LRU system.

We have proposed also a method to compute the layout. The fixed-layout management system has been the best management system when there are only few conflicts; else the LRU with fixed-size blocks system is the most predictable in general.

As a future work, we will try to analyze the predictability of the JIT compilation process for different optimization levels; and also the predictability of the advanced techniques used in a JIT compiler to achieve more performance, etc.

Bibliography

[1] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-level analysis of a portable Java byte code WCET analysis framework. In *RTCSA '00: Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, Washington, DC, USA, 2000. IEEE Computer Society.

[2] G. Birkhoff. *Lattice theory (3rd ed.)*. American mathematical society, Providence, Rhode Island, 1967.

[3] François Bodin, André Seznec, and Erven Rohou. Salto: System for assembly-language transformation and optimization. In *Proc. 6th Workshop on Compilers for Parallel Computers*, pages 261–272, 1996.

[4] Marco Cornero, Roberto Costa, Ricardo Fernández Pascual, Andrea C. Ornstein, and Erven Rohou. An experimental environment validating the suitability of CLI as an effective deployment format for embedded systems. In *HiPEAC'08: Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers*, pages 130–144, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[6] Silviu S. Craciunas, Christoph M. Kirsch, Hannes Payer, Ana Sokolova, Horst Stadler, and Robert Staudinger. A compacting real-time memory management system. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 349–362, Berkeley, CA, USA, 2008. USENIX Association.

[7] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling Java just in time. *IEEE Micro*, 17(3):36–43, 1997.

[8] Evelyn Duesterwald. Dynamic compilation. In *The Compiler Design Handbook*, pages 739–762. 2002.

[9] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Syst.*, 17(2-3):131–181, 1999.

[10] Marco Garatti. FICO: A fast instruction cache optimizer. In *SCOPES*, pages 388–402, 2003.

[11] Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In *SAS '09: Proceedings of the 16th International Symposium on Static Analysis*, pages 120–136, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] Jan Gustafsson. Worst case execution time analysis of object-oriented programs. In *WORDS '02: Proceedings of the the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, Washington, DC, USA, 2002. IEEE Computer Society.

[13] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier Science Inc., New York, NY, USA, 1977.

[14] Erik Yu-Shing Hu, Guillem Bernat, and Andy Wellings. Addressing dynamic dispatching issues in WCET analysis for object-oriented hard real-time systems. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 2002.

[15] John Ban Nang Kam. *Monotone data flow analysis frameworks: a formal theory of global computer program optimization.* PhD thesis, Princeton, NJ, USA, 1976.

[16] Donald E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[17] Chandra Krintz and Brad Calder. Using annotations to reduce dynamic optimization time. *SIGPLAN Not.*, 36(5):156–167, 2001.

[18] Anatole Le and Laurie Hendren. Using inter-procedural side-effect information in JIT optimizations. In *14th International Conference on Compiler Construction (CC). LNCS*, pages 287–304, 2005.

[19] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[20] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, WashingtoAn algorithm with constant execution time for dynamic storage allocationn, DC, USA, 1999. IEEE Computer Society.

[21] Stoodley Mark, Ma Kenneth, and Lut Marius. Real-time Java, part 2: Comparing compilation techniques. http://www.ibm.com/developerworks/java/library/j-rtj2/index.html.

[22] Stoodley Mark, Fulton Mike, Dawson Michael, Sciampacone Ryan, and Kacur John. Real-time Java, part 1: Using java code to program real-time systems. http://www.ibm.com/developerworks/java/library/j-rtj1/index.html.

[23] M. Masmano, I. Ripoll, J. Real, A. Crespo, and A. J. Wellings. Implementation of a constant-time dynamic storage allocator. *Softw. Pract. Exper.*, 38(10):995–1026, 2008.

[24] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[25] T. Ogasawara. An algorithm with constant execution time for dynamic storage allocation. In *RTCSA '95: Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*, Washington, DC, USA, 1995. IEEE Computer Society.

[26] James L. Peterson and Theodore A. Norman. Buddy systems. *Commun. ACM*, 20(6):421–431, 1977.

[27] Karl Pettis, Robert C. Hansen, and Jack W. Davidson. Profile guided code positioning. *SIGPLAN Not.*, 39(4):398–411, 2004.

[28] Isabelle Puaut. Real-time performance of dynamic memory allocation algorithms. In *ECRTS '02: Proceedings of the 14th Euromicro Conference on Real-Time Systems*, Washington, DC, USA, 2002. IEEE Computer Society.

[29] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Syst.*, 37(2):99–122, 2007.

[30] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *JTRES '06: Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 202–211, New York, NY, USA, 2006. ACM.

[31] James E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[32] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. 1955.

[33] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Syst.*, 18(2/3):157–179, 2000.

[34] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

[35] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *IWMM '95: Proceedings of the International Workshop on Memory Management*, pages 1–116, London, UK, 1995. Springer-Verlag.

[36] Ding Yuxin, Mei Jia, and Cheng Hu. Design and implementation of Java just-in-time compiler. *J. Comput. Sci. Technol.*, 15(6):584–590, 2000.