HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Laboratory of Information Processing Science

Mikko Reinikainen

A Framework for Static Program Analysis

Master's Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.

Otaniemi, 15h December 2003

Supervisor: Prof. Eljas Soisalon-Soininen
Instructor: Vesa Hirvisalo, Lic.Sc.(Tech.)

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author:	Mikko Reinikainen	
Title of the thesis:	A Framework for Static Program Analysis	
Date:	15h December 2003	**Number of pages:** 60
Department:	Computer Science and Engineering	
Professorship:	T-106 Software Technology	
Supervisor:	Prof. Eljas Soisalon-Soininen	
Instructor:	Vesa Hirvisalo, Lic.Sc.(Tech.)	

Our environment is full of embedded systems that interact with their surroundings in real time. Static program analysis is one way to evaluate timing behavior of real-time systems. Several tools have been developed for static program analysis, but most existing tools are still under development. There is a demand for better tools that could be used in the development and verification of real-time systems.

Creating a complete analysis tool from scratch is laborious and needs to be facilitated. This thesis proposes a framework for static program analysis. The framework solves typical problems encountered when creating static program analyzers. A prototype implementation of the framework is also presented. The prototype implementation can be used to create static program analyzers in the Java programming language. Three applications were successfully built on the prototype implementation.

The framework is modular and contains well-defined interfaces. It is possible to extend the framework. The prototype implementation provides most of the basic functionality needed in a static program analyzer. The solution focuses on performance evaluation, but is general enough to be used in other fields of static program analysis as well.

Keywords: static analysis, program analysis, software, performance

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Mikko Reinikainen	
Työn nimi:	Työkalurunko staattiseen ohjelma-analyysiin	
Päivämäärä:	15. joulukuuta 2003	**Sivuja:** 60
Osasto:	Tietotekniikka	
Professuuri:	T-106 Ohjelmistotekniikka	
Työn valvoja:	Prof. Eljas Soisalon-Soininen	
Työn ohjaaja:	TkL Vesa Hirvisalo	

Ympäristömme on täynnä sulautettuja järjestelmiä, jotka vuorovaikuttavat ympäristönsä kanssa reaaliajassa. Staattinen ohjelma-analyysi on yksi tapa reaaliaikajärjestelmien aikakäyttäytymisen arviointiin. Staattiseen ohjelma-analyysiin on kehitetty useita työkaluja, mutta useimmat nykyiset työkalut ovat vasta kehitteillä. On tarve paremmille työkaluille, joita voitaisiin käyttää reaaliaikajärjestelmien kehityksessä ja oikeaksitodistamisessa.

Kokonaisen analyysityökalun luominen tyhjästä on työlästä ja sitä täytyy helpottaa. Tässä diplomityössä esitetään työkalurunko staattiseen ohjelma-analyysiin. Työkalurunko ratkaisee tyypillisiä ongelmia, joita kohdataan staattisia ohjelma-analysaattoreita luotaessa. Työssä esitetään myös työkalurungon prototyyppitoteutus, jota voidaan käyttää staattisten ohjelma-analysaattoreiden luomiseen Java-ohjelmointikielellä. Prototyyppitoteutuksen päälle rakennettiin onnistuneesti kolme sovellusta.

Työkalurunko on modulaarinen ja sisältää selkeät rajapinnat. Työkalurungon laajentaminen on mahdollista. Prototyyppitoteutus tarjoaa suurimman osan perustoiminnoista, joita tarvitaan staattisessa ohjelma-analysaattorissa. Ratkaisu keskittyy suorituskyvyn arviointiin, mutta on riittävän yleinen koskemaan muitakin staattisen ohjelma-analyysin osa-alueita.

Avainsanat: staattinen analyysi, ohjelma-analyysi, ohjelmistot, suorituskyky

Contents

Foreword

This thesis was made as part of the PERF project at the Laboratory of Information Processing Science of Helsinki University of Technology. The project was funded by the Academy of Finland under the grant 51509.

I would like to thank manager of the project, Vesa Hirvisalo, for providing the opportunity to learn great deal about scientific research and to make this thesis under competent instruction.

I thank Professor Eljas Soisalon-Soininen for supervising my thesis.

I thank Juha Tukkinen for having numerous conversations about my thesis, and for keeping me company at the office.

Finally, I want to thank my wife Mari and daughters Elli and Emma for being the most important thing in my life. You gave me wonderful support.

Otaniemi, 2nd December 2003

Mikko Reinikainen

Chapter 1

Introduction

Our environment is increasingly being filled with appliances that contain processors and software. Modern mobile phones, vehicles, toys, and household appliances are controlled by *computer programs* that run in a processor embedded within the appliance.

Most embedded systems interact with their surroundings in *real-time* [15]. Timing of this interaction is critical. For example, the anti-lock brake system (ABS) of a car must work exactly at the right time, or it may cause damage or endanger human lives.

The timing of programs can be evaluated by *static program analysis*. This technique extracts knowledge of the run time behavior of a program without executing the program. Static analysis of modern systems has been actively researched in the latest decade (for a review, see, e.g., Ermedahl [17]).

Several *tools* have been developed for static program analysis, but most existing tools are still under development. There is a demand for better tools, that could be used in the design and verification of real-time systems.

1.1 Problem

The main problem addressed in this thesis is, *how to create tools for static program analysis*. The main problem contains the following subproblems, which need to be solved by the designer of a static analysis tool:

- inputting the program to be analyzed,
- representing the program in a format suitable for analysis,
- describing the target machine,
- analyzing the program,
- representing the results of the analysis, and
- outputting the results of the analysis.

Creating a complete analysis tool from scratch is laborious and needs to be facilitated. A tool architecture should be *retargetable* to support new target machines, and *flexible* to support adding new analyses [17]. An *ad hoc* tool is difficult to develop further.

In this thesis, the problem is studied in the context of tools that evaluate *performance*. The problem also exists when creating tools for other purposes (e.g., compilers). However, issues that are not relevant to performance evaluation tools are out of the scope of this thesis.

1.2 Solution

This thesis uses the constructive method. As a solution to the problem, this thesis proposes a *framework* for static program analysis. The framework contains six modules that each solve one of the subproblems stated in the

previous section. The architecture supports adding new input formats, target machines, analyses, and output representations.

Also, a *prototype implementation* of the framework is presented. The prototype implementation consists of Java [49] packages that implement the modules of the framework. The prototype implementation can be used to create static program analyzers in the Java programming language.

The solution focuses on performance evaluation, but is general enough to be extended to other fields of static analysis.

Validity of the solution is shown by the constructive method: Three *applications* were built based on the prototype implementation. The applications demonstrate the capability of adding a new analysis, describing a target machine, inputting the program to the analyzer, and outputting the results of the analysis. Additionally, Tukkinen [53] has successfully created a tool that was based on the prototype implementation.

1.3 Thesis Outline

The following chapter presents context and theoretical background for this thesis. The chapter discusses performance evaluation of modern real-time and embedded systems by static analysis.

The subsequent chapters present the contributions of this thesis. The main contribution, a modular framework for static program analysis, is proposed in Chapter 3. The prototype implementation of the framework is presented in Chapter 4. The three applications built on the prototype implementation are described in Chapter 5.

Finally, results of the work are summarized and evaluated in Chapter 6, which also speculates on possible future work on the subject.

Chapter 2

Background

This chapter gives the contextual and theoretical background that is needed in the rest of this thesis. The reader is first introduced with the principles of performance evaluation. Then, performance is discussed in the light of real-time and embedded systems. After that, modern hardware features that affect performance are presented.

The background for static program analysis and different types of static program analyses are presented. Special aspects of execution time analysis are discussed. Representations that are suitable for program analysis are specified. Static analysis performed by compilers is discussed. Finally, related work is presented.

2.1 Performance Evaluation

Performance of a *software system* is the ability of the system to cope with a certain *workload* [26]. Coping with a workload means that the system fulfills *requirements* specified for the system.

A software system is a combination of *software* and *hardware*. Components

of the system are called *resources*. Resources can be divided into *software resources* and *hardware resources*. Software resources may be, for example, procedures, system calls, logical modules or whole programs. Hardware resources include computational units, memories, input and output devices and other peripherals.

The workload consists of the *input* of the system. The input can be, for example, commands typed by the user or measurement data from a measuring instrument. The execution time of a program often depends on the input.

Performance is a quality factor of the system. Performance of a system can be *engineered* in advance, but it is not a feature that could easily be added on afterwards. In some cases performance issues may be solved by making minor changes to the system. However, improving the performance of a software system may also require redesigning the whole system.

Analyzing the performance of a software system is called *performance evaluation*. Performance evaluation can utilize different *techniques*. Evaluation can be based on different criteria, which are called *metrics*. Producing reliable results with any technique requires choosing a representative workload. Choosing an unrepresentative workload can give false results on the performance of a system [26].

Techniques for performance evaluation of a software system are classically categorized into *measurement*, *simulation* and *analytical modeling* [26]. Each technique has its virtues and weaknesses.

Measurement means directly observing the behavior of the software system. This can be done, for example, by instrumenting the program code or by using some hardware measurement instruments. Measurement can only reveal the behavior of the program with certain inputs. It cannot guarantee the behavior on all possible executions. Also, the instrumentation itself may affect performance of the system.

Both simulation and analytical modeling require a mathematical *model* of the

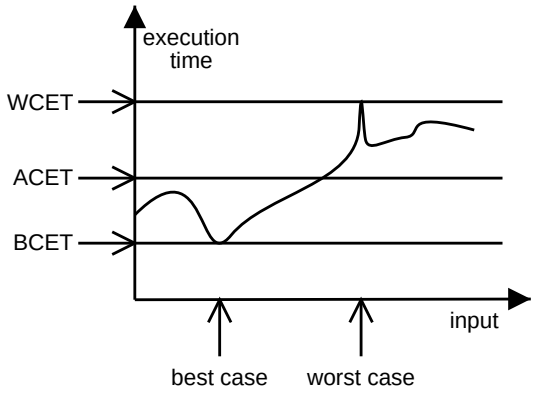

Figure 2.1: Possible execution times of a program with different inputs

system. Besides software and hardware, the input of the system has to be modeled. In simulation the behavior of the system is observed by *executing* the model and in analytical modeling by *analyzing* the model.

Simulating or analyzing a complete model is often undecidable. Therefore, an approximate model or an approximate analysis may be needed. This leads to an inability to observe all properties of the system.

Criteria for performance evaluation include *execution time*, *throughput*, *resource utilization*, *availability* and *reliability*. The following contains definitions of these metrics.

Execution time (also called *response time* or *latency*) tells how long it takes to service a request. *Worst case execution time* (WCET) is the longest possible execution time of the system. *Best case execution time* (BCET) is the shortest possible execution time of the system. *Average case execution time* (ACET) is the average execution time of the system. An example of possible execution times of a program is presented in Figure 2.1.

2.2 Real-Time and Embedded Systems

In a *real-time system*, the correctness of computation depends not only on the result, but on the time at which the result is produced. Execution times of the system must meet given *deadlines*. If a real-time system contains multiple concurrent *tasks* that use the same resources, they need to be *scheduled*. This is usually done by a *real-time operating system* [32]. Real-time systems are divided into *hard real-time systems* and *soft real-time systems*.

In hard real-time systems, producing results at a wrong time makes the result useless. An example of a hard real-time system is the airbag in a car. If during a crash the airbag does not deploy in time, it is useless. However, if the airbag deploys too early, it can be fatal, too [47].

In soft real-time systems, missing a deadline reduces quality of the service, but the system still provides a service. In a soft real-time system the average case execution time of the system is most important. An example is a real-time video player: If the system is not able to decode all of the input stream, it might need to skip a few frames but the video would still be possible to watch.

In real world, the division between hard and soft real-time systems is not always clear. Some examples of hard and soft real-time systems are presented in Figure 2.2. One must not make the assumption that real-time systems should be fast. The significant property is, that timing of a real-time system is *predictable*.

Usually only small part of the program code has real-time requirements [15]. Thus, it would be useful to be able to analyze only the timing-critical part of the software.

Embedded systems are systems that are not primarily computers, but that contain a built-in software system. Mobile phones, vehicles, automation systems, toys and household appliances that have a microprocessor are all em-

Hard real-time	Soft real-time
airbag of a car	video player
anti-lock brake system (ABS) of a car	user interface
flight-control system of an aircraft	telephone switch
medical equipment	
elevator	

Figure 2.2: Examples of hard and soft real-time systems

bedded systems. Many embedded systems need to fulfill real-time requirements. For example, all the hard real-time systems in Figure 2.2 are also embedded systems.

Embedded systems are often built of very simple hardware. Thus, they have limited hardware resources. This leads to other performance issues besides timing. For example, the power consumption of an embedded system may be critical to its use.

2.3 Modern Hardware

Caches, pipelines and other modern hardware features make static performance analysis of a program difficult, because the execution time of a program becomes sensitive to the execution history of the program. The speed of memory accesses depends on the contents of the cache, and the cycle count of a single instruction depends on the previously executed instructions.

The Performance Gap

In 1965 Gordon Moore made his famous observation that the number of transistors per integrated chip grows exponentially [36]. Today this observation still holds true, and therefore the performance of new microprocessors has

increased about 60 percent every year [55]. The speed of memories has also been increasing exponentially, but the increase is much smaller, about 7 percent per year. This has lead to a gap between the performance of processors and memory.

Because of the gap, memory is becoming more and more a bottleneck in software systems. Traditional performance metrics of a program (number of executed lines, number of executed operations) are no longer as significant as they used to be.

Modern hardware architectures have many features that try to diminish the performance gap. These features include *caches*, *pipelines*, *branch prediction*, *delayed branching*, *data forwarding* and *out-of-order execution*. These features naturally have other purposes as well, such as overcoming the maximum clock frequency that limits the performance of the system.

Caches

Caches are fast memories that are smaller than the main memory [48]. There can be several levels of caches between the main memory and the processor. *Instruction caches* are used to store instructions of a program. *Data caches* are used for data accessed by the program. A *unified cache* contains both instructions and data. Caches improve performance by exploiting *temporal* and *spatial locality*.

Temporal locality means that a memory location is repeatedly accessed within a short period of time. Contents of recently used memory locations are held in the cache in hope of repeated accesses to them. For example, instructions of a loop get loaded into the instruction cache on the first iteration of the loop. On consecutive iterations the instructions will be found in the cache.

Spatial locality implies that memory locations close to each other are likely to be accessed contemporaneously. For example, accessing the first byte of

a character string usually implies that the rest of the string will soon be accessed.

A *cache hit* means that the data requested is found in the cache and, therefore, can be quickly retrieved. A *cache miss* means that the data is not found in the cache and a slower access to upper memory level is needed.

The cache consists of blocks of memory called *cache lines* (also called *cache blocks* or *cache slots*). If a cache miss occurs, the whole memory block containing the accessed memory location is loaded in the cache. This is done on the assumption that other memory locations in that block might be accessed in the near future.

Associativity of a cache defines which cache lines can be used to store a certain memory block. In a *fully associative* cache a memory block can reside on any cache line. In an *A-way set associative* cache each memory block can reside on any of a set of A cache lines. In a *direct-mapped* cache one memory block can reside on exactly one cache line. A cache with more associativity is more flexible, but it is harder to implement.

Example 2.1 *The Motorola ColdFire 5307 contains a unified 8 kilobyte four-way set associative cache with 128 sets. Size of one cache line is 16 bytes. The cache uses pseudo round robin replacement policy: if a cache miss occurs and all four lines of a cache set contain valid data, the line indicated by a two bit global replacement counter is replaced and the counter is incremented. Layout of the ColdFire cache is illustrated in Figure 2.3. [18]*

Pipelines

Pipelines improve performance by overlapping the execution of successive instructions [46]. Execution of an instruction is divided into separate steps that are performed by *pipeline stages*. There can be one instruction performing each step simultaneously.

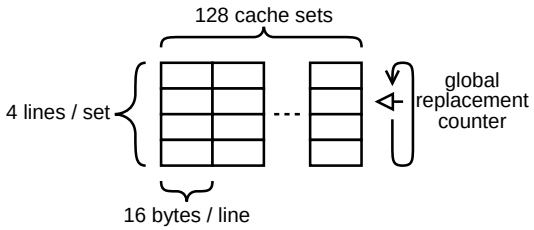

Figure 2.3: Layout of the cache of Motorola ColdFire 5307

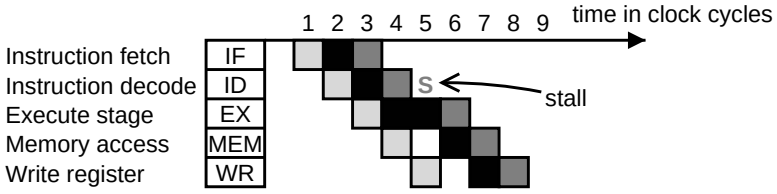

Figure 2.4: Execution of instructions in a pipeline with five stages

A pipeline may *stall* as a result of a *pipeline hazard*. There are three kinds of pipeline hazards. *Structural hazards* are caused by a conflict in the processor hardware, *data hazards* are caused by data dependencies (e.g. a *load-use conflict*), and *control hazards* are caused by changes in control flow (e.g. a *branch instruction*).

Example 2.2 *Execution of instructions in a simple pipeline with five stages is illustrated in Figure 2.4. The second instruction (black) spends two cycles in the execute stage, which causes the third instruction to stall one cycle.*

Other Hardware Features

Superscalar execution means that a processor has a pipeline that can execute multiple instructions in a one clock cycle [46]. *Out-of-order execution* means, that the processor can choose the order in which it executes the instructions. *Speculative execution* means that the processor can execute instructions before they are known to be needed. In *delayed branching* the

instruction sequentially after the branch instruction is executed before the branch actually takes place. All these features make the timing behavior of a program complicated, because there is no longer a sequential dependency between adjacent instructions.

A processor with *instruction prefetch* fetches instructions from the memory into a *prefetch queue* . When a branch, interrupt or a conditional instruction is executed, the prefetch may have fetched the wrong instruction and the queue needs to be flushed. *Branch prediction* tries to guess the correct branch after a conditional instruction, thereby reducing the risk of flushing the prefetch queue. The prediction can be based on *target address* of the branch, *history* of that jump (saved in a *branch history table*) or *decoding* of the instruction.

Architectures that use *virtual memory* have a *translation table* that translates addresses of virtual pages to addresses of physical memory pages [20]. Every memory access needs to be translated from a virtual address to a physical address. Contents of the whole translation table do not fit in the fast processor memory. Solution to this problem is *translation look-aside buffer* (TLB), which is a small cache that contains recently accessed entries of the translation table. Implementation and behavior of virtual memory affect the performance of a program.

2.4 Static Program Analysis

Static program analysis means extracting knowledge of dynamic, i.e. run time, behavior of a program without executing the program. Static program analysis can be used in tools that evaluate performance of a program. Static program analysis is also used in optimizing compilers (see Section 2.8) [3].

Because most of the interesting properties of programs are undecidable (not computable), static analysis needs to make a conservative *approximation.*

Results of the analysis must be *safe*. Safeness means that the results can be depended on. Results should also be as *precise* as possible. Precision of the results indicates how close the approximated result is to the reality.

Since static performance analysis of modern hardware features is difficult, many real-time systems use old and simple hardware without caches and pipelines. These features may also simply be switched off in order to make the system more predictable. An analysis technique that could take these hardware features into account would allow for using more advanced hardware in real-time systems.

Analysis of different programs is not equally difficult. Typical programs for desktop computers perform many arithmetic operations, whereas software for embedded systems performs more logical and bitwise operations [14].

One challenge in constructing a program analyzer is balancing the cost and precision of the analysis. Producing safe but too coarse estimations might be easy, but improving the accuracy may require an infeasible amount of computation.

Approaches to Static Program Analysis

Static program analysis can be performed with several approaches. The following presents the most important approaches.

Abstract interpretation is a well developed theory for constructing static program analyses [13, 39, 12]. Abstract interpretation provides a formal method for deriving an approximation that is based on the *semantics* of the analyzed language. The approximation represents behavior of the program in an *abstract domain* that is an abstraction of the concrete semantics of the language. The behavior of the program is characterized by equations that are usually solved by a method called *fixed-point iteration*. A common way to implement fixed-point iteration is the *worklist* algorithm.

In the *equational approach* the program is analyzed by generating a system of set equations that describes the properties of the program. The set equations can be solved by fixed-point iteration.

The *constraint based approach* is very similar to the equational approach. Instead of equations, the program is described by a system of inequations. For each system of equations there are multiple systems of inequations that have the same minimum solution as the equations.

There are other approaches to static program analysis as well. For example, *type and effect systems* describe the program by two forms of judgments: properties of states before and after executing a statement of the program, and the effect of each statement [39].

2.5 Types of Static Program Analyses

Data flow Analysis

The process of collecting information about the way variables are used during run time is called *data flow analysis* [3]. Different properties of the data flow of the program can be analyzed.

Liveness tells which variables have a defined value, i.e. are *live*, at a program point. *Available expressions* describes which calculated expressions are available at a program point:

> An expression $x \oplus y$ is *available* at a node n in the flow graph if, on every path from the entry node of the graph to node n, $x \oplus y$ is computed at least once *and* there are no definitions of x or y since the most recent occurrence of $x \oplus y$ on that path. [5]

Reaching definitions describes a relation between variable definitions and uses. For each assignment of a value to a variable, it tells the program

points at which the variable still has that value [10]. *Reaching expressions* do roughly the same to expressions as reaching definitions to values. For each calculated expression, it tells the program points at which the expression still has that value.

The data flow of a program can be described with simple set equations called *data flow equations*. The equations can be solved by fixed-point iteration. The set equations usually have multiple safe solutions. The *minimum solution* or *minimum fixed point* describes the solution that is most accurate, i.e. contains least superfluous information.

Control Flow Analysis

Control flow analysis finds the possible execution paths of a program [3]. The control flow defines the order, in which statements of the program can be executed. There are many ways to *represent* the control flow (see Section 2.7) and to execute the analysis (for example *bottom-up*, *top-down*, *up-and-down*).

The control flow graph may contain *loops*, i.e. sequences of instructions that are executed repeatedly. One method for determining the loop structure of a control flow graph is the notion of *dominators*.

Example 2.3 (dominating) *A node d of a flow graph* dominates *node n, written d* ***dom*** *n, if all paths from the initial node of the flow graph to n go through d.*

Domination can be presented as a dominator tree *[3]. In the dominator tree, each node dominates only its descendants in the tree. An example of the dominator tree of a simple control flow graph is presented in Figure 2.5. All the loops of the control flow graph can be found by searching for edges where the control flows from a node to its dominator. These edges are called* back edges.

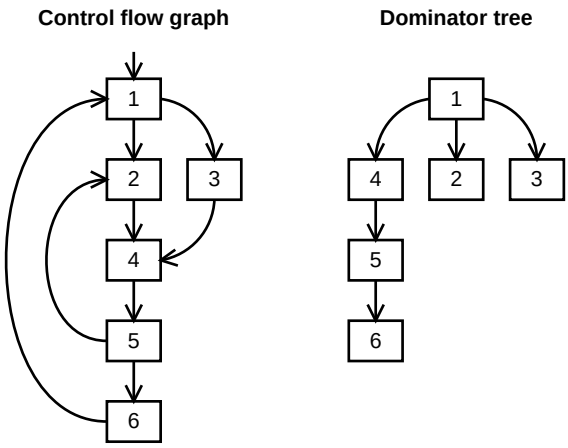

Figure 2.5: Dominator tree of a simple control flow graph

The control flow of a program is *reducible*, if it only contains loops that are either completely nested with each other, or separated [3]. *Irreducible* control flow contains partially overlapping loops. An irreducible control flow graph is more difficult to analyze than reducible.

Structured programs are easier to analyze than *unstructured programs*. Structured programs contain no goto statements nor breaks from inside loops, whereas unstructured programs can contain them. In general, the control flow is more evident in a program written in a *high level* language. *Low level* code is more difficult to analyze. Problems that arise when extracting control flow information from a binary format are discussed in Section 2.7.

Producing the control flow graph from a binary executable for modern architectures is difficult [51]. Problems may arise, for example, in the presence of memory indirections (jump or call tables, procedure variables, etc.), ambiguous usage of machine instructions, very long instruction words (VLIW), interlocked or overlapping procedures and data inside code blocks.

It may be impossible to derive all required control flow information from very complex programs [17]. In many automatic control flow analyzers the analysis is done with the help of *manual annotations* inserted in the code.

The programmer can use the annotations to insert extra flow information within the program code.

Value Analysis

Value analysis computes possible values of variables or machine registers [3]. The possible values of each variable or register are represented as sets or ranges of values. In some cases, value analysis can be used to improve control flow analysis, for example, by detecting infeasible conditional branches.

Alias Analysis

Alias analysis finds aliases, i.e. variables that share the same memory location [5]. The following variables can be aliases: variables passed as *call-by-reference parameters*, variables whose *address* is taken, l-value expressions that dereference *pointers*, l-value expressions that contain *array subscripts* and variables used in *inner-nested procedures*.

Shape Analysis

Shape analysis determines information about the heap-allocated data structures that the program manipulates [33, 45]. It tries to find "shape invariants" for programs that destructively update dynamically allocated storage.

Interprocedural Analysis

Usually the different analyses presented above are *intraprocedural*, i.e. restricted to a local level within a single procedure. *Interprocedural analysis* performs static analysis globally across the procedures of the program.

Interprocedural analysis is more difficult than intraprocedural analysis, because procedure calls often contain call-by-reference parameters, which leads to aliasing. Usually alias analysis is needed before interprocedural value or data flow analysis can be performed.

Another problem in interprocedural analysis is distinguishing the *call context* of procedures. The same procedure may result in different analysis results when it has been called from different parts of the program [44].

Interprocedural analysis can be further divided into *intratask interference* (interference caused by procedures of the same task) and *intertask interference* (interference caused by context switches and other tasks).

2.6 Execution Time Analysis

One way to guarantee that deadlines of a real-time system are met is to calculate the worst case execution time and best case execution time of tasks of the system. It may be difficult to calculate the *actual* execution times (WCET_A and BCET_A). Instead, *estimated* execution times (WCET_E and BCET_E) can be used. The estimated execution times must be *safe*, i.e. $\mathrm{WCET}_E \geq \mathrm{WCET}_A$ and $\mathrm{BCET}_E \leq \mathrm{BCET}_A$. They should also be *tight*, i.e. as close as possible to the actual execution times. Estimating the execution times of a program is illustrated in Figure 2.6.

Note, that calculating the WCET of a system does not tell us, what is the worst case execution, i.e. what input causes the system to spend most time. Respectively, BCET calculation does not reveal the fastest execution, only the duration of the fastest execution.

Static worst case execution time analysis of a program is usually decomposed into *control flow analysis*, *microarchitecture analysis* and *path analysis* [18]. Control flow analysis discovers the control flow of the program. Microarchitecture analysis consists of analysis of various features of the architecture,

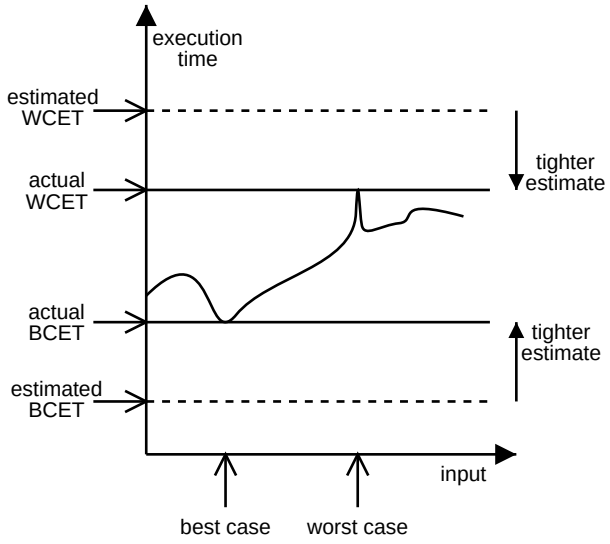

Figure 2.6: Estimating execution times of a program

such as *value analysis* (determining possible register values), *cache analysis* (determining which memory accesses cause cache hits and misses) and *pipeline analysis* (determining which operations cause pipeline stalls). Path analysis estimates the length of the longest execution path based on microarchitecture analysis.

Ermedahl [17] has presented a review of tools that perform flow analysis, cache and pipeline analysis, and path analysis. Developers of real-time systems would find it valuable, if a WCET analyzer could be integrated with other tools of the development environment [4].

Cache Analysis

A cache miss causes a severe performance penalty. Therefore, it is important to know the number of cache hits and cache misses in order to get a precise WCET estimate. For the programmer, it would also be valuable to know, which memory accesses of a program are hits and which are misses.

Cache analysis computes the *cache state* at each program point with the help of an *update function*. The update function describes how a memory access alters the cache state.

Pipeline Analysis

In order to get precise WCET estimates, also the behavior of the pipeline needs to be analyzed. A pipeline contains only a few instructions, whereas caches can contain megabytes of data. Therefore the sets of possible pipeline states are smaller than sets of possible cache states. Also, the pipeline does not have a complicated replacement policy: instructions enter the pipeline from the other end and leave it when they are done.

A pipeline with branch prediction may cause the processor to fetch instructions that will never be executed [18]. This may affect the cache and translation look-aside buffer state, which complicates the analysis.

Path Analysis

After the control flow of the program has been resolved, and the timings of individual instructions have been calculated, *path analysis* calculates the estimated length of the longest execution path of the program. There are several approaches to path analysis.

Implicit path enumeration technique (IPET) is an approach that is based on *integer linear programming* (ILP) or *constraint programming* (CP). In IPET the program is represented as a set of *integer constraints*. Solving the set of constraints reveals the length of the longest path. The path itself is not explicitly known, hence the name implicit path enumeration [17]. A program representation for implicit path enumeration technique is presented in Section 2.7.

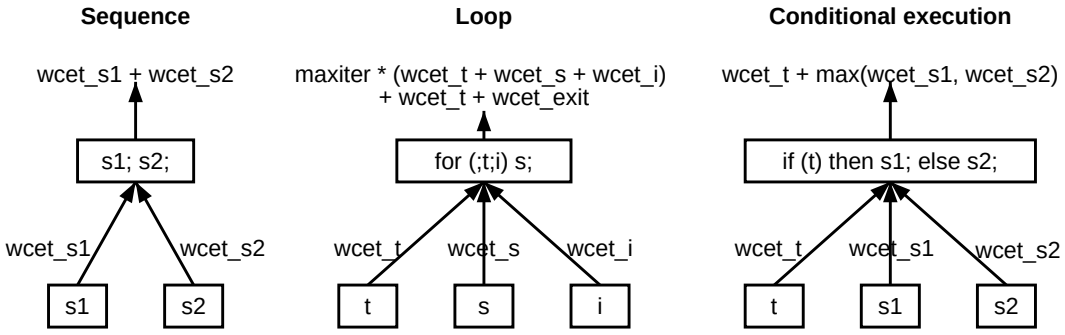

Figure 2.7: WCET calculation in tree based path analysis

In the *tree based approach* the program is represented as a syntax tree. Length of the longest path is calculated by traversing the tree in a bottom-up fashion. WCET of a program part is calculated recursively by using WCET of smaller parts of the program (see Figure 2.7 for simple rules of the calculation). [42, 11]

The main shortcoming of the tree based approach is, that it is limited to structured programs. The tree based approach also assumes that the bounds for execution time of the program can be derived from bounds for execution time of parts of the program. [31]

In the *path based approach*, the execution paths of a program fragment are explicitly explored. This approach handles complex flow constraints somewhat better than the others [31].

2.7 Program Representations

In order to be analyzed, a program needs to have a suitable representation. Not all possible representations are suitable for static program analysis. For example, a program represented in a high level language is more difficult to analyze than the same program represented in a low level language [19].

Programming languages are designed to be used in text files that are readable by human. Object code is suitable for storing the program in a linkable file. A binary executable is a compact format suitable for storing and executing the program. All these formats are cumbersome to analyze. The program to be analyzed is usually translated from these formats into an intermediate representation.

An ideal intermediate representation is

- easy to generate,
- easy to analyze,
- easy to manipulate, and
- easy to translate into target code [3].

Ermedahl [17] has surveyed the different program representations used in WCET analysis. He states that all program flow information is not useful in every type of analyses. Also, not all program representations can represent all possible program flows. Therefore, the program representation should contain a sufficient amount of flow information, but no more.

Different kinds of static analyses impose different requirements on the program representation [8]. The following contains description of common representations used in static program analysis.

Control Flow Graph

Control flow graph (CFG) is a low level program representation [5]. The *nodes* of a control flow graph are *basic blocks*. A basic block is a sequence of individual instructions that is always entered at the beginning and exited at the end. There is an *edge* in the control flow graph from a basic block to each basic block that can be executed right after the former block.

Abstract Syntax Tree

Abstract syntax tree (AST) is a high level program representation can be derived from the *parse tree* of a program by removing most non-terminal symbols. The abstract syntax tree presents the structure of the source program. Because only lexical dependencies between expressions are contained, it is difficult to recognize semantic dependences between expressions in an abstract syntax tree. [37]

Colin and Puaut [11] use a program representation that contains a control flow graph and an abstract syntax tree of the program. They perform tree based path analysis on the abstract syntax tree.

Dependence Graph

Dependence graph is a program representation that is used in some modern compilers [3]. A dependence graph describes how the computations of a program depend on each other. A dependence graph imposes minimal constraints on the order of the nodes. Results of some analyses, for example cache and pipeline analysis, depend on the order of execution of program nodes. For this reason dependence graphs alone are not an adequate representation for those analyses.

Scopes and Flow Facts

Engblom and Ermedahl [16, 17] perform WCET analysis by representing the dynamic behavior of programs with the concepts of *scopes* and *flow facts*. Scopes are individual executing environments of the program, such as function calls and loops, that can be iterated. Scopes form a recursive *scope tree*. Each basic block of a control flow graph belongs to one scope. Flow facts are integer *constraint expressions* that are assigned to specific *contexts*. The

```
function f() {
  g();
  h();
}
function g() {
}
function h() {
  g();
}
```

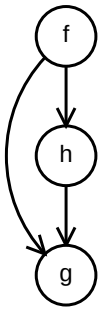

Figure 2.8: The call graph of a simple program

constraint expressions give limits to the possible execution counts of a single scope. A context specifies a set of iterations of a scope.

Call Graph

The procedures of a program can be called from multiple points in the program. From an analysis point of view the same procedure may behave differently when called from different program points. The possible *call contexts* of procedures can be presented as a *call graph*. The call graph has a node for each possible call context of a procedure. Edges of the call graph show the individual calls from procedure to another. An example of a call graph is presented in Figure 2.8.

2.8 Static Analysis in Compilers

Compilers are a major class of tools that perform static program analysis. It is useful to know the basic architecture of a typical compiler. Also, when analyzing a program, it is important to understand, how a compiler may

transform the program flow described in the original source code.

A *compiler* translates a source program written in a programming language into an executable format [3]. A compiler is usually divided into *front end*, which reads and analyzes the *source code*, and *back end*, which produces *target code* for the target architecture. This is the *analysis-synthesis model* of a compiler.

The front end translates the source program into an *intermediate representation* (see Section 2.7) from which the back end generates target code. The intermediate representation is used when static analysis is performed on the program. The representation is usually *machine-independent*, which makes it easier to apply machine-independent *optimizations* to the program, based on the analysis. Also, *retargeting* the compiler for a new target architecture is easier, because only a new back end is required.

Typical optimizations that are done during compilation are *common subexpression elimination*, *copy propagation*, *constant folding*, *unreachable code elimination*, *dead code elimination* and *strength reduction*. Different kinds of static analyses of the program are needed to perform different optimizations.

Common subexpression elimination removes expressions that compute values that are already computed. Copy propagation removes unnecessary assignments, that only rename a variable. Constant folding evaluates program elements that get constant values. Unreachable code elimination removes code that cannot be executed. Dead code elimination removes code that produces values that are not used. Strength reduction replaces expensive operations by equivalent cheaper operations. The effect of these compiler optimizations is illustrated in Figure 2.9.

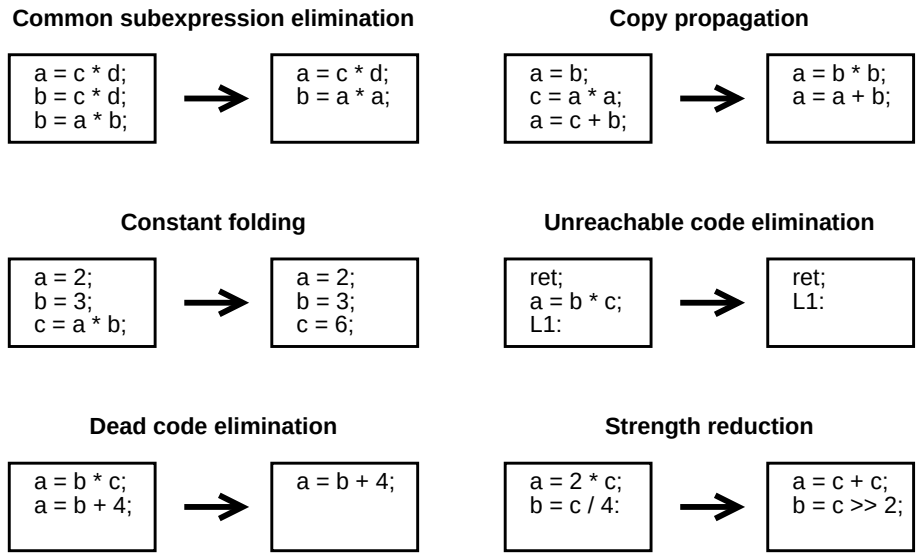

Figure 2.9: Effect of some compiler optimizations

2.9 Related Work

This section presents existing analyzer generators, frameworks and independent tools for static program analysis. Most of the tools are for performance evaluation, since the focus of this work is on performance evaluation. The work presented here works as a basis for the contributions of this thesis.

Analyzer Generators and Frameworks

PAG (Program Analyzer Generator) [34] is a tool for generating static program analyzers. It has been developed at the Universität des Saarlandes, Germany, in which there has been active research on WCET and static program analysis. The `USES`-group [18] has presented a modular system for calculating reliable run-time guarantees. Kästner and Wilhelm [28] have presented a top-down method based on program slicing for generating the control flow graph from assembly code. Theiling [51] has presented a bottom-up approach for generating the control flow graph from binary code.

Ermedahl [17] has presented a modular WCET tool architecture and a prototype implementation that supports analysis of NEC V850E and the ARM9 architectures. The tool uses a program representation based on scopes and flow facts (see Section 2.7) for WCET analysis.

Z1 is a data flow analyzer generator [57]. The user of Z1 provides a *parser* and an *abstract interpreter* of the language to be analyzed. The tool produces an executable analyzer, that maps each program point to an abstract program state. The user of Z1 can set a parameter for a desired cost-accuracy balance. The tool then generates an analyzer which has the specified performance balance. The balancing is achieved by simplifying the domain of analysis sufficiently with *projection expressions* provided by the user.

Mueller [38] has presented a framework that uses the data flow approach for bounding worst-case instruction cache performance for caches with arbitrary levels of associativity. The framework is based on the work of the group at Florida State University [54].

EEL (Executable Editing Library) [29] is a library that supports generating a control flow graph from binary executables. The library is implemented in C++. Executable files can also be modified via the library.

Individual Tools

aiT [1] is a set of commercially available WCET analyzers by AbsInt. aiT is based on PAG and the work at the Universität des Saarlandes. aiT performs the analysis on machine code level and supports several modern hardware features. For example, it can analyze the pipeline and cache of ColdFire, PowerPC and ARM7 processors. The aiT tools were originally designed in the DAEDALUS project [40] for validating the timing behavior of avionics software.

Bound-T [23] is a commercially available WCET analyzer for simple binary

code. The tool constructs a control flow graph and a call graph from the binary executable [24]. Then it analyzes loop bodies in order to find loop counters. The tool uses the Omega system [41] to implement the loop analysis in Presburger arithmetic. Integer linear programming is used to find the worst execution path. The user of Bound-T can provide the tool with assertions (for example loop bounds) that help tighten the time estimate. The tool does not support indirect memory accesses, indirect branching, loops without explicit counters nor programs with irreducible control flows.

Heptane [2] is a tree based WCET analyzer that is available under the GPL license. The tool can analyze instruction cache, pipeline, and branch prediction of Intel Pentium and MIPS processors. Analysis is done on programs written in ANSI C. A retargetable assembly transformation tool is used to obtain hardware dependent information.

Gromit [21] is a tool for static timing analysis of superscalar processors. It supports two processors of the PowerPC family. The tool is implemented in the Java language.

Cinderella [56] is a prototype tool for bounding worst case execution times of binary executables on the Intel i960KB and Motorola MC68000 processors. The tool models cache behavior by using linear constraints that are solved by an method based on IPET. Cinderella supports retargeting by separating the target dependent parts from target independent parts of the tool.

Chapter 3

A Framework for Static Program Analysis

This chapter presents the main contribution of this thesis: a framework for static program analysis. The framework is intentionally defined on a general level, which allows for constructing very different static program analyzers.

3.1 Design Principles

The basic architecture of the framework resembles the analysis-synthesis model of compilers. Interchangeable front-ends input the program into the analyzer. Then, analyses are performed on the intermediate representation. Instead of target code, the back end outputs results of the analyses.

The framework is divided into modules. Each module solves a specific problem in creating a static analyzer. Each module also has a well defined interface. Therefore it is possible to develop modules of the framework individually.

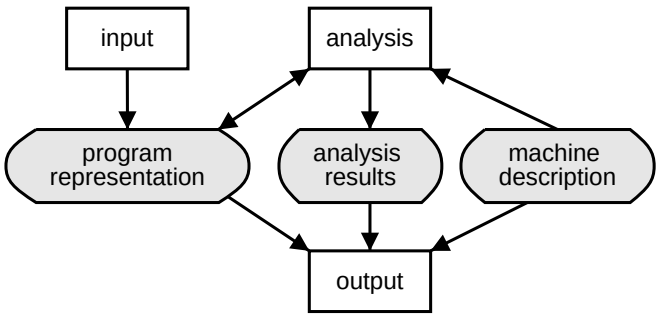

Figure 3.1: Interaction between modules of the framework

3.2 Modules of the Framework

The framework contains six modules with specific purposes:

Module	**Purpose**
Input	Inputting the program to be analyzed.
Program	Representing the program.
Machine	Describing the target machine.
Analysis	Analyzing the program.
Results	Representing the results of the analysis.
Output	Outputting the results of the analysis.

The interaction between the modules is illustrated in Figure 3.1. *Input* reads the program to be analyzed and stores it in the format specified by *program* representation. *Machine* description contains information about the target machine needed by the *analysis*. Analysis uses the program representation and machine description and produces analysis *results*, also possibly altering the program representation. *Output* presents the results of the analysis, potentially utilizing the program representation and machine description.

Input

The Input module is responsible for lexical analysis and parsing of the program to be analyzed. It must extract the control flow of the program. The Input module uses the Program module to build the analyzable representation of the program.

Depending on the implementation of the Input module, the input program can be, for example, high-level source code, assembler code, object code or a binary executable.

The Input module stores, from which input file and which line or position of the file each expression of the input program is from. If the relation between program expressions and original source code is also required in the analysis, Input module is responsible for extracting that relation as well.

Example 3.1 *Usually, a compiler produces several assembler instructions to implement a single high level language expression. A way for determining the relation between assembler instructions and original high level expressions is from debug data produced by the compiler. One format suitable for determining the relation is the "stabs" debug format [35].*

There is one input instance for each input file of the program to be analyzed. This allows, for example, the analysis of *dynamically linked* programs, that use several input files in order to be executed.

When analyzing the control flow of assembly or machine language code, it is worth noting that conditional branches and function returns are the only instructions after which the control can flow to several directions.

Program

The Program module defines a representation for the program to be analyzed. A program is represented as a simple *control flow graph*, because it is a representation that can be utilized in most analyses.

It was decided, that the flow graph contains individual instructions instead of basic blocks as nodes of the control flow graph. This makes the representation and analyses simpler (separate local analysis of the contents of the basic blocks is not required), while no information is lost. The set of possible instructions is described by the Machine module.

In addition to the control flow graph, the program representation also contains *symbol tables* that map labels to addresses and addresses to nodes of the control flow graph. These can be used to input program written high level languages or symbolic assembler.

The Program module also stores information about procedures of the program. The Input module calculates and stores in the control flow graph for each node which procedure it belongs to. Each procedure has an *entry node* and several possible *exit nodes*.

Information equivalent to a call graph (Section 2.7) is stored implicitly within the procedures. Each procedure knows, which procedures it calls. This allows presenting the call contexts within infinitely recursive and mutually recursive procedures.

If a procedure needs to be analyzed differently in different call contexts, the subgraph which represents the procedure can be duplicated. In the VIVU approach [52] the same method is also used to distinguish the first iteration of a loop from the rest.

Machine

The Machine module provides means for representing the *target machine* and its properties, such as number of registers. This information may be used in analyzing the program.

The module also defines the *instruction set* of the machine. The major properties of an instruction that affect many analyses are, whether the instruction modifies control or not, and whether the instruction accesses memory or not.

The module contains an abstraction for representing *machine resources*, such as machine registers. A machine resource is divided into cells, e.g. individual registers.

The following properties of the target machine may be relevant to the analysis: calling conventions, binary and assembler representations of instructions, semantics of instructions, power consumption, code size, cycle counts, pipeline implementations, and memory hierarchy [43].

Analysis

The Analysis module is very simple in order to support all kinds of static program analyses. Basically it just provides an interface for adding new analyses that can communicate with the program representation and machine description, and produce analysis results.

The previous chapters present the general theory and worked examples of static program analysis. Each analysis has its own special considerations. The implementation of each analysis is left open to the user of the framework.

Results

Analysis results can be *intermediate* or *final*. Intermediate results are used during the computation of the analysis. Final results are presented in the Output module.

Analyses can also produce *global* or *local results*. Global results, which describe the program as a whole, can include execution time of the program and throughput of the software as per a certain service. There can be also some statistics about the program that are global.

Local results are analysis results that are associated with certain nodes or edges of the control flow graph. These include program state at a program point, and cache or pipeline state.

Nodes and edges of the control flow graph can store information about the state of the program. This will be used to store intermediate and final results of the analyses.

Output

The best way to present the results of the analysis depends on the type of the analysis. Global analysis results may be represented as straightforward text or tables.

Graph visualizing provides a natural format for presenting local analysis results that are tied to nodes and edges of the control flow graph. However, the visualized control flow graph of a program becomes easily very large, and the results of the analysis may get lost in the vast of nodes and edges.

A very useful output method would be to present the analysis results with the original source code of the program. For example, the memory references that cause a cache miss could be marked in the original source code.

3.3 Conclusions

The presented framework has a clear, modular architecture. Each module is interchangeable and fulfills a specific purpose. New features can be added to the modules as long as the current interfaces are preserved.

The Input module supports creating front ends for all kinds of program representations.

According to Engblom [14] a complete WCET analysis tool must be able to handle the following program features: recursion, unstructured flow graphs, function pointers and function pointer calls, data pointers, deeply nested loops, multiple loop exits, deeply nested decision nests, and non-terminating loops and function. The Program module supports representing all these features.

The Machine module is general enough to encompass all required machine descriptions, and provides an useful abstraction for representing machine resources.

The interface provided by the Analysis module is intentionally left very general, so that it is possible to plug in all present and future analyses. More specific communication between analyses and the rest of the framework need to be specified by the user of the framework.

Results of the analysis can be stored either globally, or locally into the control flow graph. The Output module allows a wide range of outputs to be generated from the analysis results.

Chapter 4

Prototype Implementation: JSPAF

This chapter presents Java Static Program Analysis Framework (JSPAF), which is a prototype implementation of the presented framework. The implementation contains Java packages for creating static program analyzers. JSPAF is not an analyzer generator. Instead, it facilitates creating an analyzer in a general purpose programming language.

The user of JSPAF is a programmer who wants to construct a static program analyzer in the Java programming language. This chapter describes how the framework works and how it can be used to build static program analyzers.

4.1 Architecture

In JSPAF the problem of creating a static analyzer is divided into several subproblems, which relate to the modules presented in the previous chapter. The modules implemented and subproblems solved by the Java packages are presented in Table 4.1.

Subproblem	Module	Package
Inputting the program to be analyzed.	Input	`input`
Representing the program.	Program	`program`, `graph`
Representing the target machine.	Machine	`instr`, `machine`, `resources`
Analyzing the program.	Analysis	`analysis`
Representing the results of the analysis.	Results	`analysis`
Outputting the results of the analysis.	Output	`output`
Reporting and logging progress of analysis.	-	`log`
Interfacing analyzer with the user.	-	`main`

Table 4.1: Subproblems solved by packages of JSPAF

The four interfaces Interfaces provided by the JSPAF framework to the user are illustrated in Figure 4.1. `input` allows adding new front ends that reads the program to be analyzed. `machine` contains the target machine description. `analysis` performs a specific static analysis. `output` presents the results of the analyses.

JSPAF was designed to be easily extended and modified. Many packages provide interfaces that will be implemented by user code. The packages and classes of the framework are illustrated in Figure 4.2.

The sizes of the packages of JSPAF are presented in Table 4.1. Next, a more detailed description of each package is given.

`input`

Package `input` implements the Input module of the framework. The package provides an interface for inputting the program to be analyzed. The input analyzes the syntax of the program, parses it, and generates the program representation. It stores for each instruction, from which file and position

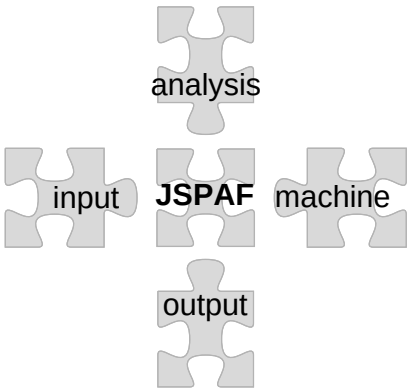

Figure 4.1: Interfaces provided by the JSPAF framework

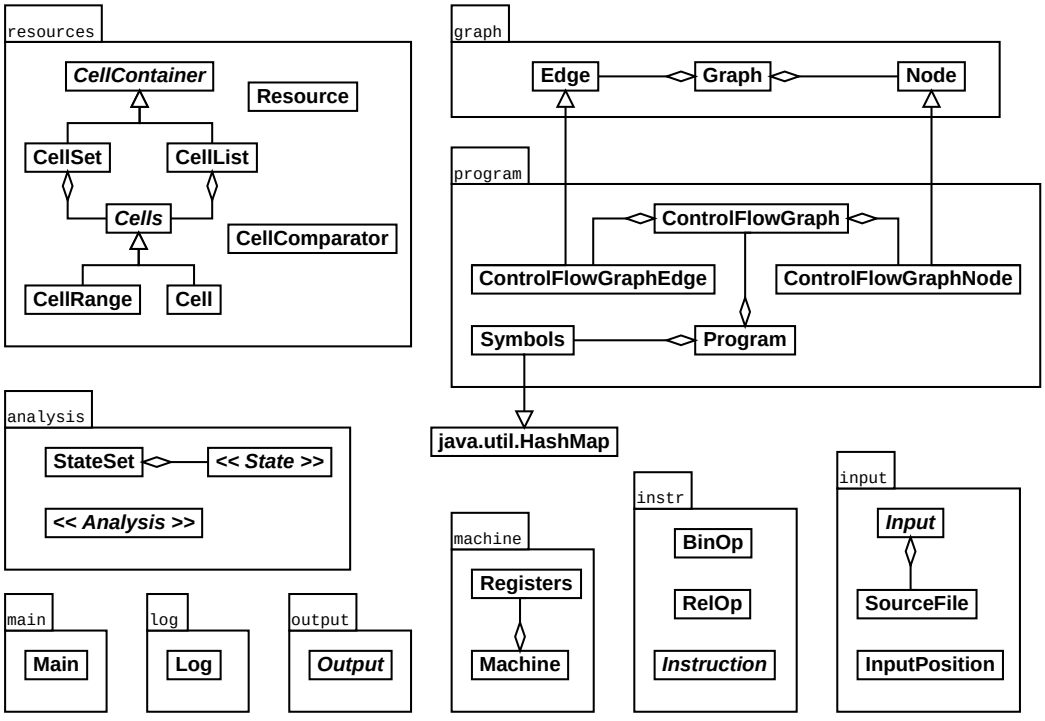

Figure 4.2: Packages and classes of the JSPAF framework

package	lines of code	relative size
`input`	196	6,4 %
`program`	748	24 %
`graph`	344	11 %
`instr`	216	7,0 %
`machine`	98	3,1 %
`resources`	912	30 %
`analysis`	215	7,0 %
`output`	48	1,6 %
`log`	50	1,6 %
`main`	249	8,1 %
total	3076	100 %

Table 4.2: Absolute and relative sizes of the packages of JSPAF

the instruction has been read.

The `input` package was designed so, that a *lexer and parser generator* can be used for lexical analysis and parsing of the input files. A lexer generator produces the program code for lexical analysis based on an input file, which describes the syntax of the input. A parser generator produces the code for parsing the input according to an input file, which describes the grammar of the language. Popular parser generators for Java include JavaCC [50], JavaCUP [25] and Grammatica [9].

`program`

Package `program` implements the Program module of the framework. It provides the data structures used by the Input module to construct the program representation. The program representation consists of *a list of inputs* used to construct the representation, the *control flow graph* of the program, *pro-*

cedures of the program, and *symbol tables* used to store labels used in the program.

The representation for procedures contains the name, entry node, and exit nodes of the procedure. Also, each procedure contains the set of procedures called by the procedure, which is used to implicitly represent the call context information. If a traditional call graph is required by an analysis, it is straightforward to implement one with the help of the general graph implementation in the `graph` package.

The symbol tables are implemented as hash maps from a label to an address, or from an address to a control flow graph node.

`graph`

Package `graph` supplements the Program module by providing general data structures and methods for representing and manipulating directed graphs. Nodes and edges of the graph can contain data.

`instr`

The `instr` package is the first of three packages used to implement the Machine module. It provides an interface for describing the instruction set of the target machine. The data stored about an instruction is: original code (for example the assembly code of the instruction), address, byte size, arguments, resource cells used by the instruction, and resource cells defined by the instruction.

Regular types of binary and relational operations are readily provided for easy implementation of such instructions.

machine

The `machine` package also contributes implementing the Machine module. It provides an interface for defining general properties of the target machine. A set of machine registers is readily defined. The number and type of registers are parametrized.

resources

The `resources` package is the part of the Machine module that is used to implement all machine resources that can be divided into discrete cells. Examples of such machine resources are the register file of the processor, the pipeline of the processor, a cache memory or a translation look-aside buffer.

The package provides data structures and methods for constructing, manipulating, and comparing sets and lists of resource cells. In addition to describing the target machine, this package is useful in implementing analyses that consider the machine resources.

analysis

The package `analysis` implements module Analysis of the framework. It provides two aids for defining analyses. The first aid is an analysis interface that has access to the program and machine representations.

The second aid is an abstraction of an analysis state, which is used to represent the analysis results, either within the control flow graph, or in separate global data structures. Analysis states can be manipulated and compared with each other.

`output`

The `output` package implements module Output of the framework. It defines an interface for outputting results of the analysis. The interface has access to the program representation, analysis results and the machine description.

`log`

The `log` package helps in reporting the progress of the analysis. It provides means for outputting messages on the terminal and optionally to a file.

`main`

The `main` package contains the main program that runs the analyzer. It provides means for parsing the command line, and running the inputs, analyses and outputs.

4.2 Conclusions

JSPAF is a collection of Java packages that implements the modules of the framework presented in Chapter 3. It provides interfaces, data structures and methods for creating static program analyzers in the Java language.

JSPAF provides most of the basic functionality needed in a static program analyzer. With the help of JSPAF, designers of a static program analyzers do not need to start from scratch. Instead, they can add individual modules, such as inputs, analyses, and outputs, on top of the framework.

Chapter 5

Practical experience

The author collected practical experience about the framework by building three applications on the JSPAF framework. The framework was still refined during implementation of the applications. This chapter describes how the applications were created.

A pipeline timing analysis tool that was built by Tukkinen [53] on top of the JSPAF framework is shortly presented at the end of this chapter.

5.1 Liveness

The first application with the framework was to implement a simple liveness analysis of machine registers. The analysis calculates, which machine registers are live at each program point. It uses the simple iterative liveness analysis algorithm presented in Figure 5.1.

Liveness analysis was easy to implement because the framework provided representation of the program as a control flow graph and means to manipulate sets of registers. The analyzer was at first tested by hand generated control flow graphs, since no inputs (front ends) were implemented.

```
repeat
    for each edge in edges do begin
        edge.live := edge.endNode.live;
    end
    for each node in nodes do begin
        node.live := {};
        for each edge in node.outgoingEdges do
            node.live := node.live ∨ edge.live;
        node.live := node.live \ node.defs;
        node.live := node.live ∨ node.uses;
    end
until (no live set changes);
```

Figure 5.1: Algorithm for liveness analysis

The whole liveness analysis package contains about 330 lines of well Javadoc commented Java code, which are distributed evenly across the three classes in the package. The package contains the following classes:

LivenessAnalysis Implementation of the liveness analysis algorithm.

LivenessState The set of live registers at a program point. Contains methods for manipulating the set.

LivenessOutput An output that writes contents of the control flow graph (including liveness information) to a file that can be rendered by the graphviz [6] graph drawing program.

5.2 Simple Machine

After implementing the first application there was a need to construct control flow graphs from real programs. A front end was built for an abstract processor called Simple Machine (SM). Simple Machine is an abstract register machine with a simple instruction set [22].

Package `sm` implements assembler input, machine description, and instruction set of Simple Machine. The package contains about 800 lines of properly Javadoc commented Java code, of which about 450 lines contain description of the instruction set. In addition there is a 85 line input file for the lexical analysis generator and a 300 line input file for the parser generator.

Input

Input of Simple Machine assembler code is implemented by the class SMInput, which extends class input.Input. Input consists of lexical analysis, parsing and flow analysis. Flow analysis includes converting jump target labels to actual jump offsets, and analyzing control flow of procedure calls and returns from procedures.

The assembler code is translated into tokens by a lexical analyzer generated with the JLex[7] lexical analyzer generator. Tokens of Simple Machine assembler are specified in `SM.lex`, the input file for JLex.

The tokenized input is parsed by a parser generated with the JavaCUP[25] parser generator. Grammar of Simple Machine assembler is specified in `Grm.cup`, the input file for JavaCUP. The parser stores the program as an incomplete control flow graph. Each instruction is stored in one node of the control flow graph. Control flow of jumps, function calls and function returns is still missing from the control flow graph. The parser adds all labels to symbol tables, which map labels to addresses and addresses to control flow graph nodes. After the first pass the symbol table contains all labels of the

```
procedure analyzeNode(proc, node)
begin
    if (node.procedure ≠ null) then return;
    node.procedure := proc;
    if (node is a call) then begin
        proc.addCall(node.calledProcedure);
        analyzeNode(node.calledProcedure, node.calledNode);
        analyzeNode(proc, node.nextNode);
    end else if (node is a ret) then
        proc.addExitNode(node);
    else for each successor in node.successors do
        analyzeNode(proc, successor);
end
```

Figure 5.2: Algorithm for analyzing procedures of the program

program.

In the second pass control flow of jumps to labels is calculated. This means that from each control flow graph node that contains a jump instruction an edge is added to the target node of the jump. Target nodes are resolved by using the symbol table. The integer offset of the jump is also stored in the jump instruction.

The third pass analyzes procedures of the program. The algorithm is presented in pseudo code in Figure 5.2. The control flow graph is traversed by going through all nodes reachable from the start node of the program. Each node is marked as belonging to a procedure. Entry and exit nodes of each procedure are stored in a data structure in class Program. The node labeled `main` is interpreted as the start node. A move to register `r3` (return address) followed by an unconditional jump is interpreted as a procedure call. All program points targeted by a procedure call are interpreted as procedure entries. A return instruction is interpreted as an exit node of a procedure.

```
procedure retTargets()
begin
    for each node in controlFlowGraph do begin
        if (node is a call) then begin
            nextNode := node.nextNode;
            proc := node.calledProcedure;
            for each exitNode in proc.ExitNodes do
                controlFlowGraph.addEdge(exitNode → nextNode);
        end
    end
end
```

Figure 5.3: Algorithm for analyzing returns from procedure calls

The fourth and final pass analyzes control flow of returns from procedure calls. The algorithm is presented in pseudo code in Figure 5.3. For each control flow graph node that is a call, a control flow edge is added from the exit nodes of the called procedure to the instruction right after the call instruction.

After the fourth pass all procedures and procedure calls of the program are known. Also, the control flow graph contains all structurally possible execution paths.

Machine description

The Simple Machine architecture is described by the class `SMMachine`, which extends class `machine.Machine`. It contains the following parameters of the architecture: *number of registers*, *size of memory in words*, and *size of instruction in bytes*.

Instruction set

Instruction set of the Simple Machine is described by classes that inherit class `SMInstruction`, which extends `instr.Instruction`. These classes are: `BinOp`, `Jmp`, `Jmp` and its subclass `JmpCond`, `Loadw`, `Storew`, `Loadb`, `Storeb`, `Mov`, `Nop`, `Ret`, and `Not`. The most interesting is `Jmp`, which stores the target label, target node, target address offset and sequential successor of a jump instruction.

5.3 Basic Block

Package `basicblocks` implements a simple basic block analysis. The package contains about 500 lines of properly Javadoc commented Java code. About 200 lines belong to class `BasicBlockAnalysisanalysis` which performs the analysis, about 150 lines belong to class `BasicBlock` which is used to represent results of the analysis, and about 150 lines belong to class `BasicBlockOutput`, which is used to output the results of the analysis.

The analysis combines nodes of a control flow graph into basic blocks. If a node has only one outgoing edge and the node at the end of the edge has only one incoming edge the two nodes are grouped together in a basic block. See the algorithm in Figure 5.4.

The control flow inside a basic block is sequential. The basic block can only be entered through the entry node and left through the exit node. There can be no conditional branches inside the basic block. However, a basic block may contain unconditional jumps, function calls and returns.

```
procedure analyzeBasicBlock(node)
begin
    block := new basicBlock();
    block.add(node);
    outgoing := node.outgoingEdges;
    while (size(outgoing) = 1) do begin
        next := head(outgoing).endNode;
        if (size(next.incomingEdges) = 1) do begin
            block.add(node);
            outgoing = next.outgoingEdges;
        end else
            break;
    end
    for each edge in outgoing do
        analyzeBasicBlock(edge.endNode);
end
```

Figure 5.4: Algorithm for basic block analysis

5.4 Pipeline Timing Analysis Tool

Tukkinen [53] has implemented in his master's thesis a prototype tool for pipeline analysis. The tool was built on the JSPAF framework presented in this thesis.

The target machines of the tool are ARM7TDMI and ARM9TDMI. The first has a 3-stage pipeline and the latter a 5-stage pipeline. The tool contains a front end that inputs assembly code generated by the GCC compiler. The analysis part of the tool computes cycle counts for individual instructions. Pipeline effects are accounted for in the analysis.

5.5 Conclusions

Three applications were successfully built on the prototype implementation of the presented framework. Each application utilizes several services provided by the JSPAF framework.

The ease of adding new analyses was demonstrated by creating an application that performs liveness analysis. Integrating the framework with a lexer generator and a parser generator was possible, and provided successfully a front end for a new input language. The basic block analysis and representation show the flexibility of the framework by generating a new view to the program representation.

The work done by Tukkinen was also a successful test of the implemented framework.

Chapter 6

Conclusions

Static program analysis tools are capable of evaluating programs with all possible inputs and all possible execution flows. This thesis presented a framework, that facilitates creating static program analysis tools. This chapter discusses the results of this thesis and speculates on possible future work on the subject.

6.1 Results

The framework is *modular* and contains *well-defined interfaces*. Therefore, it is easy to add support for new analyses, target architectures, inputs that read the program, and outputs that show the results of the analysis. Modularity also facilitates fast prototyping with naïve code that can later be reworked.

The framework provides a *program representation* that is based on control flow graphs. The representation is suitable to be used in most common static analyses [8, 18, 28, 15, 17, 46].

The *input interface* of the framework can be used to construct the program representation in bottom-up [51] or top-down [28] fashion.

The framework provides an *analysis interface* that enables describing the analyses in the common Java programming language instead of some proprietary analysis language.

The interface for the actual analyses is very general and supports creation of different analyses. However, it does not provide a ready solution to how the analysis should be internally implemented and how different analyses should communicate with each other.

A weakness of the framework is that it analyzes a single program as a whole. In order to better support dynamically loaded libraries, operating system code and intertask analysis, it should be possible to analyze only parts of the program code. It would also be useful, if one could specify by hand, which parts of a large program need be analyzed.

The practical experience gained from constructing the applications shows, that the prototype framework is useful when constructing static program analysis tools.

A static program analyzer constructed with the prototype framework does not need to be a stand-alone program. It can also be integrated with another Java program. This way it is easier to add program analysis features to a tool than by interfacing the tool with an external static program analyzer or an analyzer generator. Also, the other tool can easier provide the analyzer with important information about the program that is being analyzed.

6.2 Future Work

The presented approach of supporting construction of static program analyzers in a general purpose programming language could be further evaluated. The approach could be compared to other approaches that either use a proprietary programming language (e.g., PAG [34]) or a formal description language (e.g., [7, 50]).

Also, the presented framework and prototype implementation have potential for further development. The framework could be extended with interfaces for specific analyses. For example, to better support execution time analysis, interfaces for a *timing model* and *path analysis* could be defined.

One useful application of the framework would be to integrate a static program analyzer built on the framework with an *integrated development environment* (IDE). A developer who edits the source code of a program in an IDE could be presented information that was discovered by static analysis of the program.

Current performance evaluation tools have mostly concentrated on calculating worst and best case execution times of programs. A performance criterion that is significant in mobile embedded systems is *power consumption* of the system. The execution of the program affects power consumption of the system, and can be analyzed with methods of static program analysis [27, 30].

This thesis has considered creating static program analysis tools that are biased towards performance evaluation. However, the framework could be applied to other purposes, too. For example, the focus of the framework could evolve from performance evaluation into validating, model checking, type checking, partial evaluation, compiling, analyzer generation, and simulation.

Bibliography

[1] AbsInt Angewandte Informatik GmbH. aiT: Worst-case execution time analyzers. Web page. Referenced in November 2003. `http://www.absint.com/ait/`.

[2] ACES Software, IRISA. Heptane (Hades Embedded Processor Timing ANalyzEr) static WCET analyzer. Web page. Referenced in November 2003. `http://www.irisa.fr/aces/work/heptane-demo/heptane.html`.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[4] Jan Gustafsson Andreas Ermedahl. Realtidsindustrins syn på verktyg för exekveringstidsanalys. Technical Report 97/06, ASTEC (Advanced Software TEChnology), Box 337, SE-751 05 Uppsala, SWEDEN, 1997.

[5] Andrew W. Appel. *Modern compiler implementation in Java.* Press Syndicate of the University of Cambridge, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, United Kingdom, 1998.

[6] AT&T Labs Research. Graphviz - open source graph drawing software. Web page. Referenced in November 2003. `http://www.research.att.com/sw/tools/graphviz/`.

[7] Elliot Joel Berk and C. Scott Ananian. Jlex: A lexical analyzer generator for Java(TM). Web page. Referenced in August 2003. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

[8] Johann Blieberger. Data-flow frameworks for worst-case execution time analysis. *Real-Time Systems*, 22(3):183–227, May 2002.

[9] Per Cederberg. Grammatica: C♯ and Java parser generator. Web page. Referenced in November 2003. `http://www.nongnu.org/grammatica/`.

[10] Jean-François Collard and Jens Knoop. A comparative study of reaching definitions analyses. Technical report, PRiSM, University of Versailles, 1998.

[11] Antoine Colin and Isabelle Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proc of the 13th Euromicro Conference on Real-Time Systems*, Delft, The Netherlands, June 2001.

[12] Patrick Cousot. Abstract interpretation based formal methods and future challenges, invited paper. In R. Wilhelm, editor, *Informatics — 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[14] Jakob Engblom. Static properties of commercial embedded real-time programs, and their implication for worst-case execution time analysis. In *Proc. Fifth IEEE Real-Time Technology and Applications Symposium*, June 1999.

[15] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis.* PhD thesis, Uppsala University, April 2002.

[16] Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21th IEEE Real-Time Systems Symposium (RTSS'00)*, November 2000.

[17] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis.* PhD thesis, Institutionen för informationsteknologi, June 2003.

[18] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Rheinhard Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Embedded Software Workshop*, pages 469–485, Lake Tahoe, USA, October 2001.

[19] Jonathan C. Hardwick and Jay Sipelstein. Java as an intermediate language. Technical Report CMU-CS-96-161, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, August 1996.

[20] John L. Hennessey and David A. Patterson. *Computer Organization and Design: The Hardware/Software Interface; 2nd edition.* Morgan Kaufmann, 1998.

[21] A. Hergenhan and W. Rosenstiel. Static timing analysis of embedded software on advanced processor architectures. In *Proceedings of Design, Automation and Test in Europe (DATE '00)*, pages 552–559, March 2000.

[22] Vesa Hirvisalo, Mikko Reinikainen, and Juha Tukkinen. Simple Machine specifications. Technical Report Perf-10-GEN, Helsinki University of Technology, Laboratory of Information Processing Science, September 2003.

[23] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Bound-T execution time analyzer. Web page. Referenced in November 2003. `http://www.bound-t.com/`.

[24] Niklas Holsti and Sami Saarinen. Status of the Bound-T WCET tool. In *Proc. 2nd International Workshop on Worst-Case Execution Time Analysis (WCET2002)*, 2002.

[25] Scott Hudson, Frank Flannery, and C. Scott Ananian. CUP parser generator for Java. Web page. Referenced in August 2003. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[26] Raj Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, Inc., 1991.

[27] Chandra Krintz, Ye Wen, and Rich Wolski. Predicting program power consumption. Technical Report 2002-20, UCSB, July 2002.

[28] Daniel Kästner and Stephan Wilhelm. Generic control flow reconstruction from assembly code. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 46–55. ACM Press, 2002.

[29] James R. Larus and Eric Schnarr. EEL: machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 291–300. ACM Press, 1995.

[30] Sheayun Lee, Andreas Ermedahl, Sang Lyul Min, and Naehyuck Chang. An accurate instruction-level energy consumption model for embedded RISC processors. In *LCTES/OM*, pages 1–10, 2001.

[31] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-97/2003-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April 2003.

[32] Jane W. S. Liu. *Real-Time Systems.* Prentice-Hall, first edition, 2000.

[33] Roman Manevich. Data structures and algorithms for efficient shape analysis. Master's thesis, Tel-Aviv University, School of Computer Science, January 2003.

[34] Florian Martin. PAG - the Program Analyzer Generator. Web page. Referenced in September 2003. `http://rw4.cs.uni-sb.de/~martin/pag/`.

[35] Julia Menapace, Jim Kingdon, and David MacKenzie. *The "stabs" debug format.* Free Software Foundation, Inc. Contributed by Cygnus Support, 1993.

[36] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[37] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, 1997.

[38] Frank Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.

[39] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer Verlag, 1999.

[40] Information Society Technologies Programme of EU's Fifth Framework Programme. DAEDALUS (validation of critical software by static analysis and abstract testing). Web page. Referenced in November 2003. `http://www.di.ens.fr/~cousot/projects/DAEDALUS/index.shtml`.

[41] William Pugh, Evan Rosser, Wayne Kelly, Bill Pugh, Dave Wonnacott, and Tatiana Shpeisman. The omega project: Frameworks and algorithms for the analysis and transformation of scientific programs. Web page. Referenced in November 2003. `http://www.cs.umd.edu/projects/omega/`.

[42] Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1:159–176, 1989.

[43] Norman Ramsey and Jack W. Davidson. Machine descriptions to build tools for embedded systems. *Lecture Notes in Computer Science*, 1474:176–192, 1998.

[44] Thomas Reps, Mooly Sagiv, and Susan Horwitz. Interprocedural dataflow analysis via graph reachability. Technical Report 94/14, University of Copenhagen, 1994.

[45] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[46] Jorn Schneider and Christian Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 35–44, 1999.

[47] Maria Segui-Gomez. Driver airbag effectiveness by severity of the crash. *American Journal of Public Health*, 9:1575–1581, October 2000.

[48] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[49] Sun Microsystems, Inc. Java 2 Platform, Standard Edition. Web page. Referenced in November 2003. `http://java.sun.com/j2se/`.

[50] Sun Microsystems, Inc. Java Compiler Compiler (JavaCC) - the Java parser generator. Web page. Referenced in November 2003. `http://javacc.dev.java.net/`.

[51] Henrik Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing*

Systems and Applications, pages 23–30, Cheju-do, South Korea, December 2000.

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

[53] Juha Tukkinen. Static timing analysis for the pipelines of embedded processors. Master's thesis, Laboratory of Information Processing Science, Helsinki University of Technology, December 2003.

[54] Randall T. White, Frank Mueller, Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *Proc. of the IEEE Real-Time Technology and Applications Symposium*, pages 192–202, Montreal, Canada, June 1997.

[55] William A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[56] Sharad Malik Yau-Tsun Steven Li and Andrew Wolfe. Cinderella 3.0 home page. Web page. Referenced in November 2003. `http://www.ee.princeton.edu/~yauli/cinderella-3.0/`.

[57] Kwangkeun Yi and Luddy Harrison. Z1: A data flow analyzer generator.