Traineeship:

Estimating memory requirements of applications at the source level

Student: A. H. M de Koning
Student id: 0513849

Supervisors: Dr. Ir. A. A. Basten
Ir. S. Stuijk
M.Sc. S. V. Gheorghita

Date: August 23, 2005

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
1. Introduction

Moore’s law states that the compute power of integrated circuits doubles every 18 months. The operating frequencies of integrated circuits have become higher and higher in the past three decades to conform to this law. The high clock speeds lead to high energy consumption. Energy consumption is nowadays the bottleneck in chip design for two reasons. Firstly, heat produced in an integrated circuit is caused by energy consumption. Chips malfunction or are destroyed after a threshold temperature is reached. Secondly, high energy consumption leads to a shorter battery life for mobile devices. The energy storage capacity of batteries has not increased much over the past three decades and is not expected to do so in the next few years. Therefore, energy consumption is particularly important for embedded systems. Current developments in integrated circuit design show a shift from high clock speed single processor systems towards on-chip multi-processors that operate on low clock frequencies. The low clock frequency allows chips to be powered by a low input voltage, which leads to a lower energy consumption, as the energy consumption has a linear relation with the operating frequency and a quadratic relation with the input voltage. For instance, a system containing a single 100 MHz processor operating on voltage $U_1$ will consume more energy than two 50 MHz processor operating on voltage $U_2$, if $U_2 < U_1$. The performance of the two systems is the same if the available parallelism can be fully exploited. On-chip multiprocessors have a high performance/energy consumption ratio, are scalable, flexible and are therefore a well-suited solution for mobile applications.

Current developments show that chips are increasingly getting memory dominated. For 2010 it is expected that 90% of the chip area will consist out of memory and may even go further in the future for two reasons. Firstly, logic scales faster with chip technology than memory. Secondly, current multimedia applications require increasingly more memory. The electronic design community expects future electronic systems to be multiprocessors systems in which platforms are re-used that integrate many IP blocks. Therefore, it is no longer affordable for each different sub-system to have a separated, large memory, as adding on-chip memory has a huge effect on chip area. This suggests a high level of memory re-use. Until now, dedicated memories were close to the computational logic and had small access times. However, future chips memories will be at a larger distance from the computational logic and are potentially shared among many computational resources.

In single processing systems, designers had little freedom in mapping the application on the available memory resources. This changes with the upcoming of multiprocessor systems in which many memories are available with different access times, which are potentially shared among different processors. Information is needed on the memory requirements of an application in order to map it onto a multiprocessor. A designer must know when different data structures are used in the application and what the memory requirements of these data structures are. This report provides a means to estimate the memory requirements applications at the source level. Chapter 2 defines the problems that occur in estimating the memory requirements of an application. Chapter 3 presents the approach used to tackle these problems. Next, Chapter 4 describes how a solution for these problems was implemented. Chapter 5 shows a case study for a multimedia application. And finally, Chapter 6 presents conclusions and recommendations for further work.
2. Problem definition

2.1 Architecture template

A general template of a multiprocessor architecture [1] is depicted in Figure 2.1. A multiprocessor architecture consists of tiles, connected to an interconnection network by means of an interface. Tiles have a communication assist available, which handles communication and decouples communication from computation. Two types of tiles exist: tiles with and without processors. The memory available in tiles without processors can be accessed by multiple tiles and is therefore known as remote memory. The memory in tiles with processor(s) is known as local memory. Remote memory will generally be larger than local memory.

Figure 2.1: Multiprocessor architecture template

2.2 Design space exploration

The Y-chart in Figure 2.2 is a general approach to explore the design space of electronic systems and can be applied to design a multiprocessor. A multiprocessor system is a combination of an application, an architecture and a mapping that determines which part of the application runs on which part of the architecture. Usually, the designed system is analyzed, providing feedback to the designer. Next, the designer can adjust the application instance, the architecture or the mapping in order to come to a system that conforms to the system performance requirements, for instance compute power, energy consumption, chip area, etc. The application is usually split in different parallel tasks, which are mapped onto the different processor tiles of the architecture. The data dependencies or communication between tasks are mapped onto connections of the underlying communication infrastructure. The mapping also decides how and which data structures are stored in which memory resource. To come up with a good mapping technique it must be known which are the memory requirements of the application and where and when the data structures are used in the application. Finding means of estimating the memory requirements of an application and detecting where data structures are used in the application is the main problem addressed in this report.

Figure 2.2 Y-chart
2.3 Memory organization

The memory organization seen by an application typically consists out of a memory space called the stack, a memory space called the heap and a global data section. An application consists out of functions. Data structures that are declared outside functions are global variables, which are accessible in all the functions from the application. Global variables are stored in the global data memory. Local variables are data structures that are declared inside a function and are only accessible inside that function. Local variables are stored on the stack and are all removed from the stack when the function returns. The data structures that are passed between functions are stored on the stack. If a function calls another function, then the calling function pushes the data structures that are passed to called function on the stack. The function that was called can use the parameters by accessing them on the stack. The heap is used for data structures for which memory is dynamically (on runtime) allocated.

2.4 Abstraction level

The abstraction level of the application under evaluation should match the abstraction level used by designers. The programming language C is the dominant programming language in embedded systems and it is therefore the best choice as abstraction level. An advantage of applications written in C is the relative independence of the hardware backend (Trimedia, ARM, MIPS, PowerPC, x86, etc). The basic building blocks in a C program are functions. The memory requirement of an application is formed by the maximum application stack size plus the size of global variables. Dynamic memory allocation is rarely used in embedded systems and therefore it is not considered in this work. When evaluating applications written in C, the goal is to give a good estimation of the memory requirements and not the precise amount. This is, because the exact required amount of memory depends on the C-compiler. Compiler optimizations that remove variables lead to an overestimation of the memory requirements. Underestimations are possible, as certain optimizations for execution speed (e.g. loop unrolling) can introduce extra variables, which increase the stack size. However, the estimated memory requirement is exact if all compiler optimizations are switched off.

2.5 Calculating stack size requirements

A small example on how to calculate the stack size of a function is given in Figure 2.3. In the code snippet one global variable and six local variables are declared on various positions in the code. In Figure 2.3b the lifetime of the variables is plotted vertically and the memory requirements are plotted horizontally. Figure 2.3b shows that the required amount of memory differs in the various positions in the code. The memory requirement of an application is determined by the maximum used stack size plus the size of global variables that are used. Calculating the size of the global data structures is straightforward, so the remainder of this subsection focuses on calculating the stack size.

A scope tree is a means for determining the stack size of an application and is shown for the example in Figure 2.3c. A scope is a code block, which is a segment that starts with an opening brace '{' and ends with a closing brace '}'. A code block that appears inside another block is a nested code block and constitutes a separate
A local variable is alive in the scope in which it was declared and inside nested scopes. The nodes in a scope tree represent the code blocks of a function. Nested blocks of code are the child nodes that connect with an edge to a parent node. A node is a leaf of the scope tree, if it has no connected children. Variables that are alive in a parent node are also alive in its children. The required stack size of a function is determined by traversing the scope tree. The maximum over the children of a parent plus the memory size of the variables declared inside the parent is taken in each step through the scope tree. In this example the required stack size is 20 bytes (assuming a 32-bit processor architecture, i.e. 4-byte integers).

![Code snippet and Scope tree](image)

Figure 2.3 Calculating memory requirements. (a) Code snippet of an application; (b) Life time of variables versus required memory size; (c) Scope tree of example.

Usually function calls occur in applications. Function calls introduce an edge from the scope where the function call is, to the root of the scope tree of the called function. An example of a code snippet that includes a function call is shown in Figure 2.4a. The corresponding scope tree is shown in figure 2.4b.

![Code snippet and Scope tree](image)

Figure 2.4 (a) Code snippet (b) Scope tree
Problems can occur when calculating the memory requirements as outlined in Figure 2.3 when function calls are present in the application. Not every function call present in the application will always be executed in practice, though the scope tree of the called function will be evaluated when the function call is detected while traversing the scope tree. Consider for example the code in Figure 2.5a in which a recursive function call is used. The corresponding scope tree is shown in Figure 2.5b. Calculating the memory requirements in the same way as for the example in Figure 2.3 would require traversing a tree with an infinite number of nodes, which forms a problem. The calculated required amount of memory for this application would then be infinite, which is impossible in practice. In reality, recursive functions do not iterate infinitely many times and they have therefore bounded memory requirements.

Recursive functions are an example of the fact that not every function call present in the application is always executed in practice. The real problem in calculating the memory requirements of an application by traversing a scope tree is the fact that control and data flow are not taken into account, while these flows determine whether or not a function call is executed. The control flow determines the structure of the application, for instance if-then-else statements and loops. The data flow determines for instance, if the condition inside the braces of an if(condition) statement is true. In order to have a good estimate of the memory requirements, a means must be found to model control flow and data flow in the application and these modeled flows must be taken into account while traversing the scope tree.
2.6 Problem statement

The focus of this report is on estimating the memory requirements of an application written in C and detecting the data structures that are used. The memory requirements of an application consist of the memory required to store the stack and the global data section; dynamic memory allocation is not considered. A technique to calculate the stack size requirements of an application is by using the scope tree. Control flow and data flow are not taken into account in this technique, which leads to the following problem:

- A function call in a scope is always counted in the mentioned scope tree technique for estimating required memory size, while it is not always executed in practice. This problem can lead to an incorrect estimated memory size.

This problem is handled in this work.
3. Approach

This section introduces the approach that was used to tackle the problem of recursion and the lack of control- and data flow information when calculating the memory requirements of an application using a scope tree.

3.1 Strings

To tackle the problem that each function call in a node of the scope tree is always executed, first some notations and concepts are introduced. Alphabet $F$ is the alphabet of all the names of the functions in the application. A string over alphabet $F$ is defined as a sequence of symbols of $F$. A finite string is a finite sequence of symbols. An example of a finite string is $\sigma = main, foo_3$. The set of all finite strings of arbitrary length over alphabet $F$ is written $F^*$. An infinite string over alphabet $F$ is an infinite sequence of symbols. The set of all infinite strings over alphabet $F$ is denoted $F^\omega$. The set of all strings (finite plus infinite) over alphabet $F$, denoted $F^{*\omega}$ is the union of $F^*$ and $F^\omega$. Function $\ell$ from strings to natural numbers yields for each string its length.

The concatenation of two strings $\sigma, \pi \in F^\omega$, denoted as $\sigma\pi$ is the sequence of length $\ell(\sigma) + \ell(\pi)$ defined as follows: For any natural number $i$ with $0 \leq i < \ell(\sigma)$, $\sigma\pi(i) = \sigma(i)$ and, for any $i$ with $\ell(\sigma) \leq i < \ell(\sigma) + \ell(\pi)$, $\sigma\pi(i) = \pi(i - \ell(\sigma))$.

A string $\sigma \in F^\omega$ is a prefix of some string $\tau \in F^\omega$, denoted $\sigma \preceq \tau$, if and only if there is a string $\pi \in F^\omega$ such that $\tau = \sigma\pi$.

A (recursive) cycle can now be defined. With $\pi \in F^\omega$ and natural number $n$, recursive cycle $\sigma \in F^{\omega n}$ is defined as string $\sigma = \underbrace{\pi \ldots \pi}_n$, which are $n$ strings $\pi$ concatenated.

3.2 Feasible execution paths

The problem in case of recursion is that an automated means for calculating the memory requirements of an application cannot deal with infinite sequences of function calls. A sequence of function calls is called (execution) path, and can be modeled with a string. A finite path can in practice always capture a sequence of function calls in an application. An infinite amount of memory resources is required if this would not be the case, which is impossible. An automated solution can only construct the sequence of function calls it sees in the scopes. In practice the sequence of function calls can depend on control flow and data flow of the application. Some paths, for instance, can be constructed based on the function calls in the scopes, but can never occur in practice.

An abstract representation of this problem is sketched in Figure 3.1. An automated memory requirements calculator uses the function calls in the scopes to limit the set of all possible execution paths that can appear in the application (observed space). However, this space still contains paths that cannot appear in practice, due to control and data flow. The set of paths that can appear in the application is the feasible region and is part of the observed space. A path that is part of the feasible region is called a feasible (execution) path. The user has to limit the observed space in such a way that it only contains the feasible paths.
Therefore, the user needs means for specifying the paths that occur in the application. The paths may become very long in case recursion cycles appear in the function call graph. It is inconvenient for the user to completely specify these paths. A shorthand notation is introduced for describing recursive cycles.

### 3.3 Regular expressions

This section introduces a short hand notation style for specifying regularity in paths. The notation style is based on regular expressions [2]. Let \( k \) be a natural number and \( \sigma, \pi \in F^{-\infty} \). Then \( \sigma = (\pi_1, \pi_2, ..., \pi_k) = (\pi_1, \pi_2, ..., \pi_{k-2}, (\pi_{k-1}))^2 = (\pi_{k-1})^{k-1} = (\pi_k) \).

The power \( k \) defines how many times the string \( \pi \) is concatenated in the recursion cycle. If \( k = 0 \) then \( \sigma \) contains no symbols. In regular expressions usually an iteration \( * \) is used, which means zero or more. In this context the \( * \) is made an explicit integer \( k \). This makes sense, as the strings are meant to limit the observed space of execution paths. There are no restrictions on the string \( \pi \). This holds that string \( \pi \) can contain recursive cycles as well, which can again be captured in a regular expression. A recursive cycle within a recursive cycle is called a nested cycle.

We now argue that arbitrary feasible execution path can be described using regular expressions. Any feasible function call sequence is bounded. Hence, any set of feasible execution paths can be described by a Finite Automaton (FA). Any set of concrete traversals of the scope tree of an application translates in a straightforward way to a finite automaton. Any FA can be described by a regular expression [2]. An example of a finite automaton showing the relation between function calls is shown in Figure 3.2. If there is a function call in a scope of a function, then it is shown as a label on the arc between the two states. Inside a state the function that was called is executed.
The automaton in Figure 3.2 corresponds to the following regular expression:

\[ \text{foo}_1 \cdot ( \text{foo}_1^* \cdot ( \text{foo}_2 \cdot \text{foo}_2^* \cdot \text{foo}_1 \cdot \text{foo}_1^* )^*) \]

This regular expression and the automaton show that function \( \text{foo}_1 \) can call itself and can call function \( \text{foo}_2 \). Function \( \text{foo}_2 \) can also call itself and function \( \text{foo}_1 \). The set of all strings that comply with this regular expression form the observed space in Figure 3.1. Since in practice any iteration is bounded, arbitrary iteration \( \ast \) can be replaced by concrete powers when describing feasible execution paths. In regular expressions, next to \( \ast \), also concatenation \( (\cdot) \) and or \( (\oplus) \) are defined. Operator concatenation was already defined in Chapter 3.1. Operator or is implicitly defined; for instance if a feasible set is described by regular expression \( \varphi \cdot (\alpha \oplus \beta) \), then this corresponds to the set containing \( \varphi \cdot \alpha \) and \( \varphi \cdot \beta \), with \( \alpha, \beta, \varphi \in F^\omega \). The operators defined in this chapter thus implement regular expressions. Therefore, strings and the operators derived from regular expressions do not put constraints or limitations on the kind of applications that an automated memory requirements estimator can evaluate.

### 3.4 Applying strings to applications

With the concept of feasible execution paths and regular expressions a user can specify the set of feasible sequences of function calls to an automated means of estimating the memory requirements of an application. The user describes the function call behavior of an application in a number of feasible paths, which are the sequences of function calls that appear in the application. All feasible paths in the application need to be specified to the automated memory requirements calculator to get a good estimate of the memory requirements. A path consists out of functions, but to keep notations short, a path may also contain strings and the regular expression operator power \( k \). To clarify how a user could specify strings for an application, consider the code snippet in Figure 3.3.
When calculating the memory requirements of the application starting from function \textit{main}, let's assume the following sequences of function calls appear in practice.

(1) \textit{main, foo_1, foo_1, foo_1, foo_1, foo_2, foo_2, foo_2}

(2) \textit{main, foo_1, foo_1, foo_2, foo_2, foo_2, foo_2, foo_1, foo_2, foo_1}

(3) \textit{main, foo_1, foo_2, foo_1}

The automaton in Figure 3.2 accepts these sequences of function calls. However, an automated solution for calculating the memory requirements of the application knows that these are the only feasible sequences of function calls that need to be evaluated. Therefore, the user specifies paths. In this case, one could first specify the following strings, which can be used to specify common parts of execution paths.

\[
\text{string}_1 = \textit{main, foo}_1
\]

\[
\text{string}_2 = (\textit{foo}_2)^3
\]

Now the three feasible sequences of function calls can be captured in 3 short paths by using these strings and the power operator:

\[
\text{path}_1 = \text{string}_1, (\textit{foo}_1)^3, \text{string}_2
\]

\[
\text{path}_2 = \text{string}_1, \textit{foo}_1, \text{string}_2, (\textit{foo}_2, \textit{foo}_1)^2
\]

\[
\text{path}_3 = \text{string}_1, \textit{foo}_2, \textit{foo}_1
\]

An automated memory requirements calculator now knows that these are the only valid paths that occur in the application.

\section*{3.5 Optimizations}

Optimizations can be applied when calculating the stack size requirements of an application in case recursions are present in the function call graph. If a function is part of a recursive cycle, then an automated means of estimating the memory requirements of an application evaluates the same segment of that function for many times, without gaining any new information, see for instance the example in Figure 2.5. Therefore the following optimization can be applied:

Let \( k \) be a natural number, \( \pi \in F^\omega \) and \( \sigma \in F^\omega \) be a recursion cycle, with \( \sigma = (\pi_1, \pi_2, \ldots, \pi_k) = (\pi)^k \). Then the stack size of sequence \( \sigma \) equals \( k \) times the stack size of sequence \( \pi \). Now, string \( \pi \) does not have to be evaluated \( k \) times, but only one time. For example, the stack size of sequence \((\text{foo}_1, \text{foo}_2)^{10000}\) equals 10000 times stack size of sequence \((\text{foo}_1, \text{foo}_2)\).
4. Implementation

This chapter describes the implemented solutions to the problems that were introduced in the previous chapters. The tool *statesize* calculates the memory requirements of an application. *Statesize* returns the required stack size, plus the size of used global structures for all functions in an application. *Statesize* essentially traverses the scope tree of an application. The user has means to pass the feasible paths in a XML file to *statesize*. The tool uses the paths to detect whether a function call present in the scope tree is indeed allowed, given the sequence of functions preceding the call. The tool shows per function, the function calls in that function and what the global variables are, that are used in the functions.

### 4.1 SUIF Compiler system

An automated means of calculating the memory requirements of an application is implemented in the Linux tool *statesize*. The tool is written in C++ and is build upon the open source SUIF Compiler System [3], which is developed by the Stanford Compiler Group of Stanford University. The SUIF system is organized as a set of compiler passes built on top of a kernel that defines an intermediate file format. The passes are implemented as separate programs that link with the kernel contained in the SUIF library. Each pass typically performs a single analysis or transformation and then writes the results out to a file. The tool *statesize* evaluates the memory requirements of applications in the intermediate SUIF format. The transformation from applications written in C, to the SUIF format is automated; a compiler (scc) and linker (linksuif) are present in the SUIF system to perform this task.

### 4.2 Calculating memory requirements

As stated in the problem definition, the memory requirements of an application are formed by the stack size requirements of the functions plus the used global variables. A scope tree needs to be constructed in order to calculate the required stack size of a function. Whether or not a function call present in a scope is executed, depends on the user specified paths. For reusability reasons, the application is split in two main components: the Stack Size Calculator (SSC) and the Function Call Validator (FCV), see Figure 4.1. The stack size calculator traverses the scope tree of the input application. A function call is valid if this function call plus the preceding function calls is a prefix on any of the user specified allowed paths. The SCC requests the FCV to determine if a function call is valid, given the previous function calls.

![Figure 4.1: Block scheme statesize](image)

---

13
The concept of a feasible path was introduced in the previous chapter as a sequence of successive function calls that occur in the application. Paths are implemented as lists in `statesize`. The FCV determines the feasible paths in the application from a user specified input XML file and stores them in a series of lists. The Stack Size Calculator also constructs a list for a path with all successive function calls it encounters in the application. The SCC adds a function call to this path, if it was judged valid by the FCV. The FCV makes this decision based on the feasible paths the user provided.

Note that `statesize` dynamically checks if a function call is valid; it does not statically execute the user specified paths. This is done to allow the split of the implementation into two separate entities, which can be modified independent of each other. `Statesize` can therefore easily be reused and extended in the future. An example of a possible extension is to allow the user to specify infeasible paths instead of feasible paths.

### 4.2.1 Implementation of execution paths

Execution paths are implemented in `statesize` as lists. A list is comprised of elements of the class `CFunctionNode`. `CFunctionNode` objects form a list, in which each element represents a function. The class `CFunctionNode` is used to store information about the function it represents. The class provides means for:

- Management of the list, e.g. adding/removing functions to/from the list.
- Retrieving a function if it is present in the list.
- Storing the names and sizes of global variables.
- Storing the names of nested function calls.
- Printing output results to console and ASCII text file.

A block scheme showing the relations between `CFunctionNode` objects is shown in Figure 4.2.

![Figure 4.2: Block scheme of class CFunctionNode](image-url)

The class `CFunctionNode` is heavily used in the implementation of `statesize`. The Function Call Validator constructs for each user specified path a list of
CFunctionNode objects. These lists represent the feasible execution paths of the application. The Stack Size Calculator maintains two CFunctionNode lists. One list (named pPath) represents the sequence of successive function calls that were judged valid by the Function Call Validator. This list of names is constructed from the path from the function at the root of the scope tree, till the function that is currently evaluated by the SCC. The FCV compares its lists with pPath plus a function call, to judge if the function call is valid. The other list maintained in the SCC (named gpReferencedFunctions) contains all the functions that have been evaluated by the SCC. All the calculated results (the memory requirements/used global variables and called functions) are stored in the corresponding nodes in this list and are printed to the console/file when all calculations are done.

### 4.2.2 Stack Size Calculator

The basic algorithm for calculating the memory requirements as it is implemented in statesize is given in pseudo-code in Figure 4.3. The algorithm is recursive and iterates from the root of the scope tree to the leaves in a post order way. See Figure 2.3c in Chapter 2 for an example of a scope tree. The root of the scope tree is the block of code that forms the body of a function; it starts with the first opening brace in the function and ends with the closing brace. For each node in the tree, the required stack size consists out of the maximum of the memory size of the children of the node (which are nested scopes), plus the memory requirements of a called function and the size of the non-static variables declared in this scope. The memory requirements of a function call are only taken into account if the path from the root of the tree to the parent node of the call plus the called function is valid according to the Function Call Validator. If a function call is valid, then the function is added to the path. After the memory requirements calculation of the called function is complete, the function is removed again from the path. The Stack Size Calculator applies this algorithm starting from the entry function of the application, which is specified by the user.

```c
for all input files
  for all functions in file
    if(function == 'main')
      computesize( root scope of function)

int computesize( scope )
{
  size = 0;
  for all children nodes of scope
  |
    sizechild = computesize( child node)
    size = max(size, sizechild)
  } // end for all children

  if function calls in this scope
  | { // do all function calls
    if ( FunctionCallValidator: pPath, function is valid)
    |   { // if call is valid
      pPath = pPath->AddNode(function);
      sizecall = stack size function call
      size = max(size, sizecall)
      pPath = pPath->RemoveNode();
    }
    size += memory size of non-static variables declared in this scope
  }
  return size;
}
```

*Figure 4.3: Pseudo code of stack size calculation algorithm*
4.2.3 Function Call Validator

The function call validator checks whether a requested function call is valid, given the preceding function calls in a path. The basic algorithm is given in pseudo code in Figure 4.4. First, in the initialization phase all user-specified paths are parsed from the input XML file. Now a function call is judged 'valid' by the FCV if the requested path plus function is a prefix of a user specified path.

```c
void Initialize(char * XMLPathFile)
{
    array m_pPath[] = XMLParseAllPaths( XMLPathFile)
    m_NumPaths = number of parsed paths
}

bool IsFunctionCallValid( path, function)
{
    for i=0; i<m_NumPaths; i++)
    {
        if( path + function is prefix of m_pPath[i])
            return true;
    }
    return false;
}
```

*Figure 4.4: Pseudo code of Function Call Validator*

Obviously, a user can make a mistake in specifying the feasible execution paths of an application. In that case, two situations can occur: the specified path can be inside the observed space (Figure 3.1) or it can be outside the observed space. If the path lies inside the observed space, then this can lead to an overestimation of the memory requirements. If the specified path lies outside the space that can be observed by *statesize*, then this path does not influence the calculated memory requirements.

4.2.4 Global memory

There are three types of data structures that are stored in the global data section of an application: static variables, global variables and temporary data structures introduced by the SUIF compiler. A string between the quotes of a *printf* function is an example of such a temporary data structure. A variable that is declared static is only accessible within the scope it was declared; however, the lifetime of a static variable is no longer confined to this scope, but it will be global. In other words, a static variable is a global variable, which is only accessible within its local definition scope. Therefore, local static variables are stored in a global memory section and not on the stack. *Statesize* detects static variables when they are declared in a scope. Global variables are not declared inside scopes and are therefore detected differently; *statesize* detects if an instruction in a scope uses a variable that is declared global. If a global variable is detected to be used in a function, then its memory size is calculated and the name and size of the *directly* used global variables are stored for that function. Global variables that are used in nested functions are stored as *indirect* used global variables. Compiler introduced temporary global data structures are handled the same way as global variables. Note that a user can force *statesize* to ignore the temporary structures by specifying a command line parameter (see Chapter 4.3.2).
The stack size calculator holds a list (called `gpReferencedFunctions`) of `CFunctionNode` objects. Each node in the list represents a function that has ever been evaluated by the stack size calculator. The name and size of detected global variables and temporary data structures are stored in the corresponding node in this list. `Statesize` prints out the information stored in this list to the console and to a file.

`Statesize` calculates the size of all the data structures that are declared in the application: the total program data size. This number is the size of all global- and static variables and the size of all compiler introduced temporary data structures. If all data structures that are declared in an application are used at some point, then the total program data size corresponds to the size of the used global variables (which includes the temporary data structures) plus the size of the static variables used directly or indirectly by the main entry function of the application (typically ‘main’).

### 4.3 User interaction

#### 4.3.1 The XML Path File

The user specifies all allowed sequences of function calls in an XML file. A general template of the XML input file is given in Figure 4.5. All valid paths are placed in the `<paths>` section. Each path has a name and optionally a “repeat” property with a value, which corresponds to the number of times the content of the path is repeated. A path contains `<function>` and `<string>` elements, which must have a name property and optionally a “repeat” property. Strings that are used in paths must be specified in the strings section. A string may include other strings, with optionally a “repeat” property. The order of the elements within the paths and strings is important; two paths with the same elements but in different order define two different paths.
4.3.2 Command line parameters

Statesize is operated via command line parameters. This section describes what the command line parameters are and how they interact with statesize. Options with default values only have to be specified if the user does not agree with this value.

Tool usage: statesize OPTIONS input_file1 input_file2 ...

input_file1 input_file2 ...

Input files of the application in the SUIF intermediate format

OPTIONS:

-h or --h

Shows the command line options help

-warning

Sets the warnings on
-ignoretmp
  Ignore temporary data structures introduced by the SUIF compiler.

-functionstacksize _value_
  Sets the size of the stack used to save program state for a function call. For instance, the size to save the CPU registers, base stack pointer and instruction pointer.

-showfunctioncalls
  Show the sequences of function calls

-output _file.txt_
  The calculated results are written to the file specified by this option. Default 'results.txt' is used.

-main _mainfunction_
  Sets the entry point of the application; function main is used by default.

-paths _pathfile.xml_
  Specifies the XML file that contains all valid sequences of function calls. A sequence of function calls encountered in the application is only executed if it is part of a path in the <paths/> section of the _pathfile.xml_. See Chapter 4.3.1 for details on the XML file. The memory requirements are only calculated if this option is set. Note that this option is ignored if -writepaths is set.

-writepaths _pathfile.xml_
  This option is a helping hand for the user on how to specify the paths and recursions in the application. Statesize writes all the sequences of functions it encounters in the application to the <paths/> section of the XML file. Recursive cycles present in the application are detected and are written to the <strings/> section. By default, recursions are limited to have a repeat property of 2. Note that the detected paths correspond with the observed space of Figure 3.1, with the restriction that recursion cycles have a repeat factor of 2. A written <path/> does not contain <strings/>; recursion cycles are decomposed in <functions/>. The user must manually verify and edit the paths in _pathfile.xml_ with the behavior of the application, before using it with the -paths _pathfile.xml_ option.
5. Case Study

This section presents the results of a case study conducted with statesize on a typical multimedia application: MPEG 1 – Layer 3 (MP3) decoding. MP3 is part of the MPEG-1 Audio standard. The MPEG-1 Audio coding algorithm [6] is structured in three layers, each one offering a specific trade of bit rate, complexity, delay and subjective quality. Layer 3 has the highest complexity, but offers the best subjective quality audio for low bit rates. MP3 has become the audio standard of the Internet and is applied nowadays in many mobile devices.

In this case study the memory requirements of an implementation of an MP3 decoder were estimated with statesize. The results are compared with the memory usage of the mp3 decoder that was measured while running on an x86 machine.

5.1 Experiment setup

5.1.1 SUIF Intermediate format

The MP3 decoder is comprised of 10 C-files. Statesize can only analyze applications in the SUIF intermediate format. The application is transformed into this format by the scc compiler, which resulted in <filename>.spd files (1). Next, the tool linksuif is applied to link the .spd files (2). And finally mergesuif is applied on the linked files to merge the files into one file: mp3decode.spd1 (3). The reason for this is that all the input files for statesize have to be specified on the command line. The merging of input files is convenient, but not mandatory. A summary of the commands is given below.

1) scc -V --sp <file1.c> <file2.c> ... <file10.c>

2) linksuif <file1.c.spd> <file1.c.spd1> ... 

3) mergesuif <file1.c.spd1> ... mp3decode.spd1

5.1.2 Specifying the feasible paths

The feasible paths of the application have to be specified to statesize in order to estimate the memory requirements. The MP3 decoder contains over 50 functions and manually specifying all the feasible paths is a time consuming task. Therefore, statesize is used to detect recursion cycles and write out paths to the XML file mp3paths.xml, with the command:

./statesize –writepaths mp3paths.xml mp3decode.spd1

Statesize detected and wrote 174 different paths and 1 recursion cycle to mp3paths.xml. This took 6 seconds on the test machine (see section 5.1.3 for specifications). All the paths were manually validated with the source code of the application. The detected recursion cycle only contains one function: MPG_MDCT. By default statesize–writepaths sets the repeat property of detected recursion cycles to 2. However, the source code shows that function MPG_MDCT calls itself for 5 times. This has been manually corrected in mp3paths.xml. Two infeasible paths were removed. MPG_MDCT can only call function MPG_DCT_2pt after 5 iterations;
therefore the two paths in which MPG_MDCT_2pt was preceded by less iterations of MPG_MDCT were removed. No other paths had to be modified.

5.1.3 Experiment system

All experiments are conducted on a 1 GHz Intel Pentium III processor with 4 GByte of memory with the Redhat 9 Linux operating system kernel 2.4.20-24.9smp. The gcc compiler version 2.95 was used with optimizations switched off (level 0) was used to compile applications. An application requires more memory if it runs on a system with an operating system. In order to get insight of the overhead caused by the OS, the memory requirements of a very small application are measured with the real-time analysis tool top [7].

```
int main(void)
{
    while(1);
    return 1;
}
```

Field 5.1: Test application

<table>
<thead>
<tr>
<th>kbytes</th>
<th>TSIZE: Code size</th>
<th>SIZE: Code size + Stack size + data size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>228</td>
</tr>
</tbody>
</table>

Top measures the amount of memory required to store the code of the application, (TSIZE) and measures the amount of memory to store code size + stack size + data (SIZE). The rounded results for the test application of Figure 5.1 are shown in Table 5.1. The results from top show that 228 kBytes of memory are added to the memory usage of an application due to the operating system. Statesize cannot calculate the overhead caused by the OS, as it only can calculate the memory requirements of applications that are in the SUIF intermediate format. Statesize reported that 0 bytes of memory are required for the test application. The overhead due to the OS and the code size of the application should be taken into account when comparing the results of top and statesize. The results of statesize have to be compared with the effective size, which is the measured stack size plus data size, minus the OS overhead.

Though it is possible to take the OS overhead into account, it is very hard to compare the results of statesize with the results of top. There is always a difference between the results produced by statesize and the results of top. This difference is caused by several factors:

- An application already uses 228 kb of memory due to the operating system, which is not measured by statesize.
- The code of the application is stored in memory. Statesize does not take this into account.
- Statesize can only analyze functions that are in the SUIF intermediate format. Library functions cannot be analyzed by statesize, though these functions can have a huge effect on memory usage.
- The granularity of memory storage as it is measured by top and as it is calculated by statesize are very different. The memory management unit of the test machine can only store data in blocks of kilo bytes; it cannot store separate bytes, or words.

It is possible to correct for the first two factors as explained above. The granularity of storages makes it particularly difficult to compare the results of statesize with top. Unfortunately, at the time of the experiment, no other measurement tools were available.
5.2 Results

5.2.1 Estimated memory requirements

The memory requirements of the MP3 decoder are estimated with statesize, with the command:

\texttt{statesize -paths mp3paths.xml mp3decode.spd1}

The entry point of the application is \texttt{main}. This means that the calculated stack size for function \texttt{main} equals the estimated required stack size for the MP3 decoder. The memory required to store the global data structures of the application is also determined by function \texttt{main}. All the results are written to the file \texttt{results.txt}. A snippet of this file is shown in Figure 5.2 and gives the calculated results for function \texttt{main}. The snippet shows which functions are called in \texttt{main} and if the called function could be analyzed. A function can only be analyzed if the source code in SUIF intermediate format is available. The snippet further shows which global variables are used in the application and their size. It also shows the size of the used static data structures. Statesize creates an overview as shown in Figure 5.2 for all analyzed functions.

```
****
****    FUNCTION:       main()
****
****    STACKSIZE:
****       - without function calls : 12 bytes
****       - with function calls : 6068 bytes
****
****    CALLS FUNCTIONS:        ANALYSED
****       - initMainMain() [x]  
****       - print() [ ]          
****       - exit() [ ]           
****       - strcpy() [ ]         
****       - MPG_Get_Filespos() [x]  
****       - MPG_Read_Frame() [x]  
****       - @print() [ ]         
****       - MPG_decodes_L2() [x]  
****       - remote_control() [x]  
****       - usleep() [ ]         
****    GLOBAL VARIABLES:
****       - used in this function : 310 bytes
****       - __tmp_string_4_4 11 "  
****       - __tmp_string_1_5 2 "  
****       - __filename 256 "  
****       - __tmp_string_2_2 37 "  
****       - __tmp_string_1_1 2 "  
****       - others, from function calls : 38799 bytes
****       - __tmp_string_6_2 400 "  
****       - @huffman_table_1 30 "  
****       - @huffman_table 2 60 "  
****
****    STATIC VARIABLES:
****       - SIZE static this function : 0 bytes
****       - Statics from function calls : 66464 bytes
```

Figure 5.2: Snippet of the results for function main in results.txt

Figure 5.2 shows that statesize has calculated that 6068 bytes (6 kB) of memory is required to store the stack of function \texttt{main}, if function calls are taken into account. As \texttt{main} is the entry function of the application, the estimated required stack size of the mp3 decoder is 6 kB. Note that Figure 5.2 shows that statesize could not analyze
the source of each function that was called in the MP3 decoder, which negatively
influences the quality of the estimation. Figure 5.2 further shows how much memory
is required to store the global and static data structures that are used directly and
indirectly in function main. The memory required for the global variables is
34.2 kB (318 + 34709 bytes) and for static variables is 45.4 kB (46464 bytes). In total
79.6 kB (34.2 + 45.4) of memory is required for storing the used global plus static
variables. The total required memory is 85.6 kB (79.6 + 6). A summary of the
estimated memory requirements of the MP3 decoder is given in table 5.3.

<table>
<thead>
<tr>
<th>Stack size</th>
<th>Global variables</th>
<th>Static variables</th>
<th>Total size</th>
</tr>
</thead>
<tbody>
<tr>
<td>size kB</td>
<td>6</td>
<td>34.2</td>
<td>45.4</td>
</tr>
</tbody>
</table>

Table 5.3: Estimated memory requirements

5.2.2 Statistics

Statesize produces statistics about the application that was analyzed. These
statistics can be used to judge the quality of the calculated memory requirements.
The statistics show how many function calls are detected and how many function
calls are analyzed. If there is a large difference between these two numbers, then the
quality of the calculated memory requirements can be poor. This makes sense, as
statesize cannot reason about functions it cannot analyze due to absence of source
code. Figure 5.3 shows the statistics produced after analyzing the MP3 decoder.

Statistics:
Number of functions in the application : 56
Number of function calls : 554
Number of function calls without source : 310
Number of analysed scopes : 237

Total Program Data Size : 96386 bytes

Figure 5.3: Statistics of MP3 Decoder

According to Figure 5.3, the MP3 decoder is comprised of 56 functions and 554
function calls are detected. The source code for 310 function calls (56%) are calls to
functions of which the source code is not available for statesize. In this case, all these
functions are library functions; for example: fprintf, getc, ioctl, socket
and sendto. The quality of the estimation could be poor, as 56% of the function calls
could not be analyzed.

The amount of data required to hold the global data structures of the MP3 decoder is
determined by the size of the used global plus static structures of function main.
Statesize also calculates the size of all declared global data structures, known as the
total program data size (see Chapter 4.2.4). Figure 5.3 shows that the total program
data size of the MP3 decoder is 96 kB (98386 bytes). This does not correspond to
the size of global data structures for main, which is 79.6 kB. This difference is
explained by the fact that there are global variables declared in the MP3 decoder that
are never used in the application. If the user is sure that the difference is caused by
global variables that are never used, including functions of which the source is not
available, then the user can take the size of the global data structures of main as size
of the global data section of the application. Else, the total program data size equals
the required memory size to store the global data structures of the application.
The required CPU time for detecting the paths in the MP3 decoder for estimating the memory requirements with *statesize* is summarized below:

- **CPU time:**
  - Detecting recursion cycles and paths: 6 seconds
  - Calculating memory requirements: 3 seconds

### 5.2.3 Measured memory requirements

The memory requirements of the MP3 decoder are measured with *top* while decoding an mp3 file. The measured results are shown in table 5.4.

<table>
<thead>
<tr>
<th>size (kb)</th>
<th>TSIZE: (Code size)</th>
<th>SIZE: (Code size + Stack size + data size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td></td>
<td>556</td>
</tr>
</tbody>
</table>

*Table 5.4: Measured memory requirements of MP3 decoder*

The MP3 decoder requires 556 KB of memory on the test machine. The amount of memory required to store the stack and data without OS overhead is less; 299 KB (556 – 228 – 29). There is a large difference between the memory usage that was measured with *top* (299 KB) and *statesize* (79.6 KB). It could be expected that *top* and *statesize* produce different results, as *statesize* could not analyze all function calls and both tools measure memory with different granularity, see Chapter 5.1.3. It is unfortunately not possible to quantify the impact of each of these two affects separately.

The application code of the mp3 decoder does not contain any dynamic memory allocation. The measured required amount of memory is invariant of the input file. The same amount of used memory was measured while decoding mp3 files with different bit rates.

### 5.3 Conclusions

#### 5.3.1 Feasible paths

The concept of feasible execution paths is successfully implemented in *statesize* and can accurately capture the sequence of function calls in the application. Specifying the feasible paths to *statesize* seems a time consuming and error prone task at first sight. However, *statesize* can take a lot of work out of the hands of the user with the option `-writepaths _paths.xml`. It is a powerful and accurate means for specifying almost all the paths in the application. The user only has to add paths to this file if recursion cycles are present in the application that repeat more than 2 times. The only other task for the user is removing the infeasible paths from the XML file. *Statesize* writes the detected paths in a few seconds to an XML file, while it would take a lot more time for a user, especially for applications with many functions.

#### 5.3.2 Detecting global data structures

*Statesize* detects all global and static data structures that are used in the application. The user is informed which function calls are directly used per function, and which global variables are used indirectly, via function calls. The global data size only
corresponds to the size of directly and indirectly used global plus static data structures, if all declared global data structures are used in the application.

5.3.3 Estimating memory requirements

There is a significant difference (466.4 kb) between the estimated memory requirements by statesize (89.6 kb) and the measured memory usage (556 kb). It was expected before the experiment was conducted that the measured memory usage with top on the test machine would result in a higher value than was calculated with statesize. The main reasons for this are the difference in granularity of memory storage and the fact that in this case study 59% of the function calls were to functions of which the source code was not available to statesize. The accuracy of the estimation suffers from this lack of information. It is concluded that top cannot be used to verify the results of statesize. Therefore, the functional correctness of statesize still needs be verified experimentally.

Note that usually in embedded systems the memory requirements of library functions are known. Also note that the implementation of the MP3 decoder is a general implementation that complies with the MPEG-1 Audio standard. It is not a ‘lightweight’ implementation targeted for embedded systems. The same holds for the RedHat 9 Linux operating system. Embedded systems typically run operating systems with very low memory usage, or use no operating system at all. In that case, the estimated memory requirements would be closer to the measured memory requirements; however it would still not be accurate if the complete source code of all the used functions is not available.
6. Conclusions and future work

6.1 Conclusions
With the upcoming of multiprocessor systems designers have a lot of freedom in mapping an application on an architecture. Part of the mapping problem is the mapping of the data structures on the available memories. In order to do this, a designer must know what the memory requirements of the applications are and what the data structures are, that are used in the application. This work has presented a technique for estimating the memory requirements of an application, based on traversing the scope tree of an application. Applications are written in C and are assumed not to use dynamic memory allocation. The concept of feasible execution paths has been introduced to solve the problem that control flow and data flow are lacking in a scope tree. A feasible execution path is a sequence of successive function calls that can occur in the application. The tool \textit{statesize} is created which calculates the memory requirements of an application. A user specifies the feasible execution paths of the application to \textit{statesize}. This compensates for the lack of control and data flow in the scope tree; \textit{statesize} uses the specified execution paths to decide whether or not to analyze a function call present in a scope. It can be concluded that:

- The concept of feasible execution paths is successfully implemented in \textit{statesize}.
- A user can easily specify these feasible paths in the application.
- Used global data structures are successfully detected.

Furthermore, based on the case study that was conducted it can also be concluded that:

- The calculated memory requirements of \textit{statesize} are an underestimation if the source code of all functions is not available.
- The results of \textit{statesize} are expected to be exact if compiler optimizations are switched off, the OS does not cause overhead and the memory management unit has the same granularity of memory storage as \textit{statesize}.
- The correctness of \textit{statesize} remains to be verified experimentally.
6.2 Future Work

6.2.1 Experimental verification
The functional correctness of statesize is not successfully experimentally verified and left as further work. In this experiment the measured memory usage should not be influenced by an OS or memory management unit and the source of all the functions in the test application should be available. A way of conducting such an experiment is by using an instruction set simulator that runs the application. The stack size required for the application can be determined by monitoring the stack pointer. The highest value of the stack pointer equals the required stack size of the application.

6.2.2 Function calls without source
In many cases the memory requirements have to be estimated of applications that use functions that are part of libraries. In that case, statesize cannot accurately estimate the memory requirements, as these functions cannot be evaluated. In that case, statesize should be informed with the memory requirements of those functions. It is left as future work to add this to implementation of statesize. A way of implementing this feature, is by adding the XML properties “stacksize” and “datasize” to functions that cannot be analyzed, but are part of a feasible path. If main, printf is a feasible path, but the source of printf is not available, then the following path could be specified:

```xml
<path name = "example">
  <function name = "main" />
  <function name = "printf" stacksize="value1" datasize="value2" />
</path>
```

If the Stack Size Calculator requests the Function Call Validator to verify the path main, printf, then the FCV should return the values of the “stacksize” and “datasize” properties to the SCC. Therefore, the FCV should be modified to collect the stack size and global data size properties from the user specified XML file. The SCC should be modified to take the values of the returned “stacksize” and “datasize” into account.

6.2.3 Infeasible paths
In the current implementation of statesize the user can only specify the paths that are feasible. The tool can be adjusted to allow the user to specify the infeasible paths in the XML file. This can sometimes be more convenient than specifying the feasible paths. This is left as future work.

6.2.4 Optimizations
Optimizations can be applied when calculating the stack size when recursive cycles are present in the application, see Chapter 3.5. These optimizations are not implemented in statesize and are left as future work.
References