Fully automatic benchmarking of real-time operating systems

(HS-IDA-EA-98-115)

Anders Larsson (a95andla@ida.his.se)

Department of computer science
Högskolan Skövde, Box 408
S-54128 Skövde, SWEDEN

Final year project in computer science, spring 1998.

Supervisor: Jonas Mellin

Fully automatic benchmarking of real-time operating systems

Submitted by Anders Larsson to Högskolan Skövde as a dissertation for the degree of B.Sc., in the Department of Computer Science.

1998-06-18

I certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has previously been conferred on me.

Signed: ____

Fully automatic benchmarking of real-time operating systems

Anders Larsson (a95andla@ida.his.se)

Key words: Benchmark, performance test, performance evaluation, real-time operating system, operating system, real-time system, software configuration, automatic configuration

Abstract

Testing and evaluating the performance of different software solutions is important in order to compare them with each other. Measuring, or benchmark, software is not a trivial task and conducting tests in a real-time environment implicates it further. Still, measuring is the only way to provide useful information, for example, which real-time operating system is best suitable for a specific hardware configuration.

The purpose of this project is to design a benchmark support system, which automatically performs benchmarks of a real-time operating system in a host-target environment. The benchmarks are conducted according to a user defined specification and the support system also allows a developer to create configurable benchmarks.

The benchmark support system described also allows parameters to increase monotonically within a specified interval during benchmark execution. This is an important feature in order to detect unpredictable behavior of the real-time system.

Table of contents

1. Introduction

1.1 Why measuring software?

The ability to evaluate computing performance is important, e.g., when comparing different hardware and software solutions in order to achieve good performance to a low price. Different ways to measure, or benchmark, software performance are required to be able to evaluate different software solutions or to validate the specification of a software component, e.g., that certain time constraints are satisfied. Making correct software measurements can be very difficult. The measurements must, apart from being correct, also be representative in the sense that they are meaningful for a purpose. Moreover, the measured data must be comparable to data from other measurements obtained under the same controlled situation.

1.2 Purpose of this work

This work is part of the real-time operating system evaluation project performed at Högskolan Skövde in conjunction with Ericsson UAB. It is a continuation of Rydgrens work [Ryd97] which resulted in a benchmarking method for real-time operating systems, based on results by McRae [McR96]. This benchmarking method has limitations in the sense that it requires lot of manual work before, and between, benchmarks; that also decreased the number of benchmarks possible to make during a period of time. Moreover, it also lacked the ability to do *automatic range parameter iterations*, i.e., iterating the benchmark over a sequence of monotonically increasing values. For example, we would like to specify that a certain parameter should increase from 10 to 10 000 in steps by 100 to see the how this affects the result. Range iterating parameters is an important tool in helping the user to discover unanticipated anomalies. A need for a more configurable benchmark system was discovered.

The purpose of this work is to design a benchmark support system to aid the user in configuration of a benchmark, or series of benchmarks. The focus is on the benchmark support and the interface between the benchmark support and the benchmark program (see Figure 1).

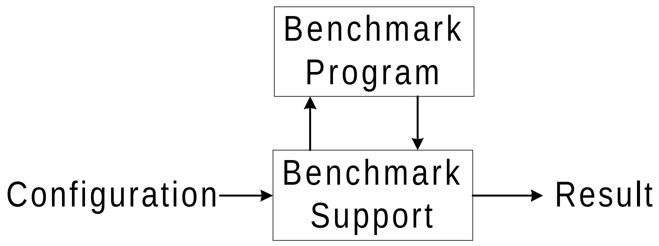

Figure 1 Benchmark support system

The user must be able to specify how the measurement is to be performed. In order to achieve the possibility to configure the benchmark, the system must be parameterized, i.e., the system must be expressed in form of changeable parameters that can be associated to names and corresponding values. The parameters can be initialized in different phases, which complicates the configuration because the lack of support for value binding in all phases. Support for binding must be realized in all phases, at a low preparation effort. A developer must be able to introduce configurable parameters using the same interface, disregarding low-level details such as binding phase, etc.

It is desirable to have a high throughput of benchmarks, i.e., returning to late phases when conducting consecutive benchmarks. But, high throughput of benchmarks results in low control over factors that affects the measurement.

Moreover, the benchmarking system is designed for use in telecommunication environments, where the host-target approach is common. Host and target environments are physically separated and the target lacks important features, e.g., developing- and debugging tools, a user-friendly interface, etc. These are the problems addressed in this work.

The results are that a benchmark support system can be realized by using macro substitution together with a name space copy to support configuration in late phases.

The *background* (Ch. 2) provides the reader with necessary information about topics discussed more thoroughly in following chapters. Topics discussed are: measurement of software and the measurement method used in this project; the host target approach used in telecommunication systems; host-target development; and ways to configure software. In the *problem definition* (Ch. 3), the problem domain is further examined and problems related to the project are identified and discussed. In the *approach* (Ch. 4), different ways of solving the problems evolved in the problem definition are discussed. The *results* from the approach are refined and presented in Ch. 5, together with a high level design of the system. Finally, conclusions are presented in the *summary* (Ch. 6); were all problems solved and what was the main contribution to benchmark science? Future work is also identified in this chapter.

2. Background

2.1 Measurement (of software)

Fenton & Pfleeger [FP96] states a formal definition of a measurement and a measure:

> "Formally, we define measurement as a mapping from the empirical world to the formal, relational world. Consequently, a measure is the number or symbol assigned to an entity by this mapping in order to characterize an attribute."

Correct and reliable measurements require clear definitions of what is intended to be measured. Attributes of objects are measured, rather than the objects themselves. Attributes can be measured directly or indirectly. Direct measurement can be applied when no other attribute or entity has to be measured. For example, measuring the throughput of a database only involves the activity of counting the amount of transactions performed per second. Indirect measurement requires measurement of one or more other attributes. For example, software quality can be viewed as a complex combination of attributes; measuring a single attribute, such as lines of code, rarely gives useful results.

When conducting a scientific investigation there are three different techniques to choose from: surveys, case studies, and formal experiments. Surveys and case studies are mainly applicable when dealing with humans, whereas a formal experiment is a controlled investigation of an activity.

Fenton & Pfleeger [FP96] claim that to produce useful results when conducting a formal experiment, such as a benchmark, three key aspects must be considered:

- Replication: The measurement must, given the same influencing factors, produce the same result over and over again.
- Representativity: The measurement data generated must reveal something meaningful about the tested object; it must be representative. The least degree of representativity accepted must be enough to make reliable comparisons between two tested objects.
- Local control: All factors that influence the tested object must be controlled, else their effects must be eliminated. Elimination of uncontrollable factors can be achieved using normalization (see section 2.1.2) or randomization to hide or skew their influence on the measurement.

2.1.1 Benchmarking

A more specific approach of formal experiments towards measuring software is benchmarking. According to Grace[Gra96], a benchmark, in the computer domain, is any program that is used to measure performance, e.g., context switching, interrupt latency, scheduling latency, interprocess communications, or I/O. Grace also claims that benchmarks are considered to be a political minefield, since they, at best, are an inexact science. Benchmarks are often done to enable comparisons between different software- and hardware configurations.

Rydgren [Ryd97] states that, apart from considering the three principles of a formal experiment, the benchmarking method used, should have the following properties:

- Portability: Measures from some software- and hardware configuration must be able to perform on another configuration of software and hardware in order to be able to compare the results.
- Simplicity: The benchmark should be easy to implement and use. The measured data should be easy to analyze.

2.1.2 Benchmark program

According to McRae [McR96], a *benchmark program* (see Figure 2), that measures the software, consists of the following parts: initialization code, reference code, measurement code and the actual benchmarked code. The initialization code is performed before the measurement loop is entered, and its main purpose is to set controllable factors to desired values. The reference code is one part of the artificial load and is also needed to provide desired conditions for each measurement, e.g., setting caches to a desired state. The other part of the artificial load is the interrupt frequency. Interrupts prevent the benchmark program to entirely reside in cache memory that would produce an incorrect result. It is possible to add further kinds of loads, provided that they are controllable, but this has been demarcated. Rydgren [Ryd97] states two desired properties of the reference code: i) stability, the code must always execute within the same amount of clock cycles; ii) non optimizability, the code should not be subject to compiler optimization.

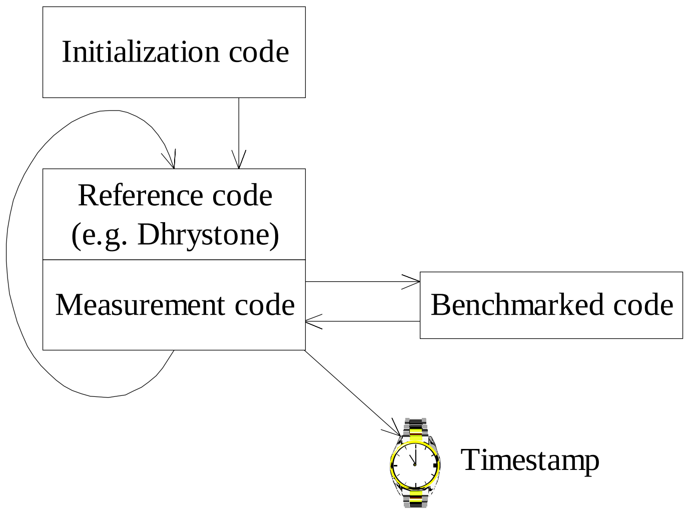

Figure 2 Benchmark program template

The measurement code first calls the benchmarked code that performs the work that is going to be measured, e.g., send- and receive activities when measuring interprocess communication. Secondly, the measurement code generates a timestamp, i.e., a pulse, and control is returned to the reference code. When iterated, the sequence of pulses generated (see Figure 3) provide the useful information from the measurement. The timestamp information is transferred to the host (batch) for further analyze. When executed for the first time the benchmarked code is not called and the measurement data provide a baseline (see Figure 3) for future measurements using the same configuration. The baseline is needed as a reference to normalize subsequent measurements in order to minimize the influence from uncontrollable factors, e.g., the hardware.

For example, the baseline is obtained during the first execution of a benchmark (see lower graph of Figure 3). The y-axis represents the duration of the performed work (µs) and the x-axis represents the number of iterations through the benchmarked program.

The benchmark program is iterated 30 times. The measurements are obtained during the second execution (see upper graph of Figure 3).

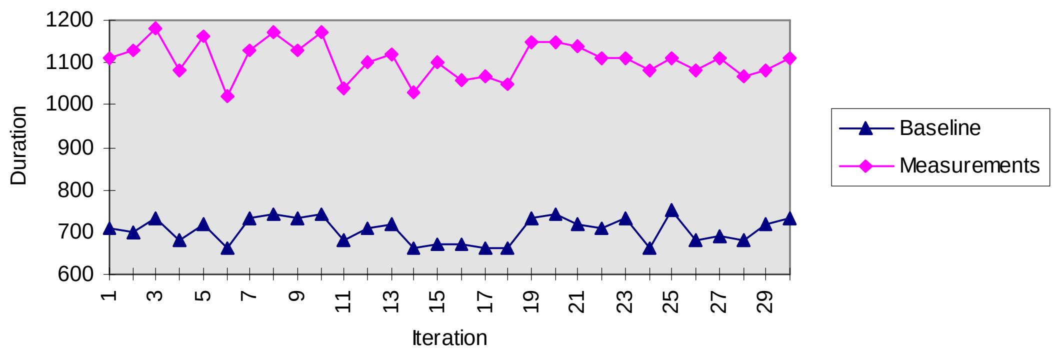

Figure 3 Example of unnormalized measurements and baseline

The normalization is performed by taking the measurements and subtracting the average of the associated baseline obtained using the same configuration (see Figure 4).

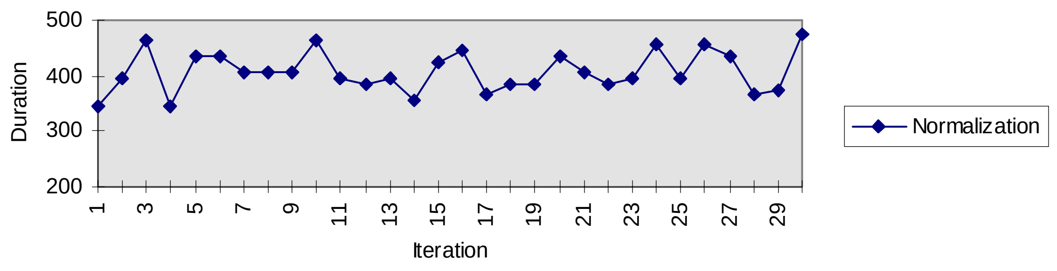

Figure 4 Normalized measurement

2.2 Telecommunication systems

Configuring a benchmarking process for a real-time operating system in a telecommunication environment is a complex task. A telecommunication system is a real-time system application that has high requirements on safety, predictability and availability; it may be disastrous if deadlines are missed.

According to Rydgren [Ryd97], telecommunication systems are also:

- Main-memory based. Time critical programs are locked in physical memory. Virtual memory is only used for maintenance programs.
- Using little or no secondary storage I/O.
- Handling a lot of communication, both internally and externally.
- Parallel in their nature - they rely heavily on multitasking and must handle a high rate of interrupts.

2.2.1 Host-target environment

Telecommunication systems are often developed in a heterogeneous environment where host and target computer are separated (see section 2.3). The target environment basically consists of a processor, memory, a bus and communication peripherals. Communication between host and target environments can, e.g. be serial- or Ethernet communication; facilities for supporting this can be provided from the real-time operating system and must also be supported by the hardware monitor (see section 3.4.1).

2.2.2 Real-time operating systems

A real-time operating system helps the programmer to control the target hardware. Traditional operating systems and real-time operating systems are quite similar in the services that they provide; multitasking, interrupt processing, I/O and memory management. According to Tuulari and Päivike [TP95], real-time operating systems are time critical and respond to external events in a finite period of time; time constraints cannot be guaranteed in a non real time operating system. Therefore real-time operating systems are suitable for control of hardware in embedded real-time systems.

2.3 Host-target development

According to Schütz [Sch93], real-time systems tend to be embedded in larger systems that they are supposed to control. This environment might even be dangerous for humans, which is one reason for using separate computers; the host- and the target system. Other reasons for using separate computers are: space, weight, and cost requirements.

The host is usually equipped with a complete software engineering environment used to develop real-time software, e.g., cross-compilers to generate executable code for the target. In contrast, the target, which is the execution environment, lacks most of these features and is optimized for space, weight, efficiency, and cost. The target is often created for a specific task or application. Thus, no developing tools are available. Schütz also states that the target operating system is functionally optimized for execution of hard real-time tasks, task interaction, and synchronization. In order to interact with the environment, special I/O devices must be supported, e.g., analogue to digital converters.

Testing in a host-target environment is not trivial. Schütz promotes the use of unit testing and preliminary integration testing on the host. Whereas, completion of the integration tests and system tests should be performed on the target. Testing on the host is easier due to the range of available tools; there is usually very little test support on the target. Although, testing target software on the host require ways to simulate the target environment. According to Schütz, this can be achieved through instruction-level simulators and environment simulators. But, in order to render meaningful results, tests to evaluate performance and temporal behavior must be performed on the target system.

2.4 Configuration concepts

One of the purposes of this work is letting the user specify the configuration of the benchmark system, i.e., how measurements are carried out and how the measured data is treated. This requires that the system is *parameterized*, i.e., the system is divided into configurable components. These components, i.e., *parameters*, can then be set to perform the benchmark according to the parameters' value specification. For example, the value associated to a parameter named `loops` may specify the number of iterations through the benchmark program.

A bitpattern, or a value, could be used to identify a certain value, but using human sensible names is much easier.

2.4.1 Naming in general

According to Needham [Nee93] the purpose of names is to identify values. Needham distinguishes between pure and impure names. Pure names are bitpatterns used for unique identification. Impure names can be divided into two subcategories; names intended for internal use, and human sensible names (see Figure 5). The impure names implicate commitments that must be honored if the names are to stay valid. For example, the validity of the directory structure "Projects/Programming/Reports" requires the existence of a directory called "Programming".

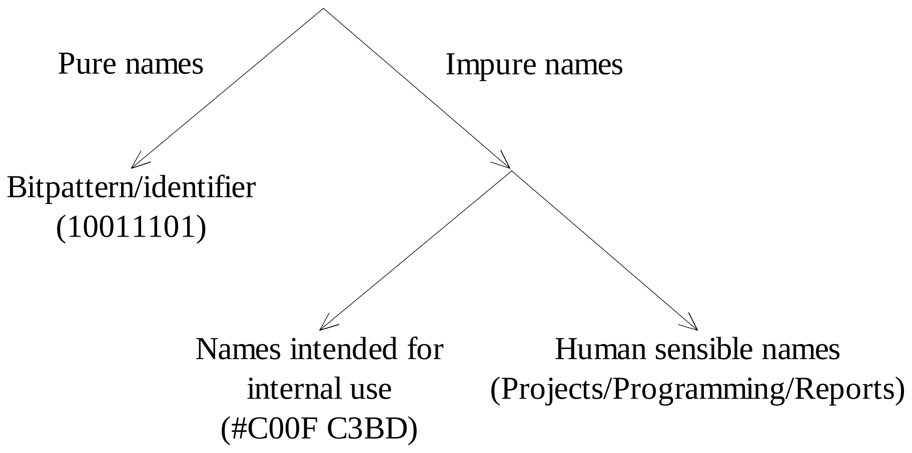

Figure 5 Naming in general

2.4.2 Name to value mapping and binding

The *name to value mapping* can be viewed as a name space database where names are associated to values. The association can be *static* or *dynamic*. Static association means that the association between name and value is fixed and the associated value cannot be changed, whereas dynamic association means that it can be changed. The *binding* is the process where the value is bound to the name in the program, i.e., the program gets aware of the value. The binding can also be static or dynamic.

When developing a computer program, such as a benchmark program, the binding can be realized in different phases, using four different mechanisms (see Figure 6).

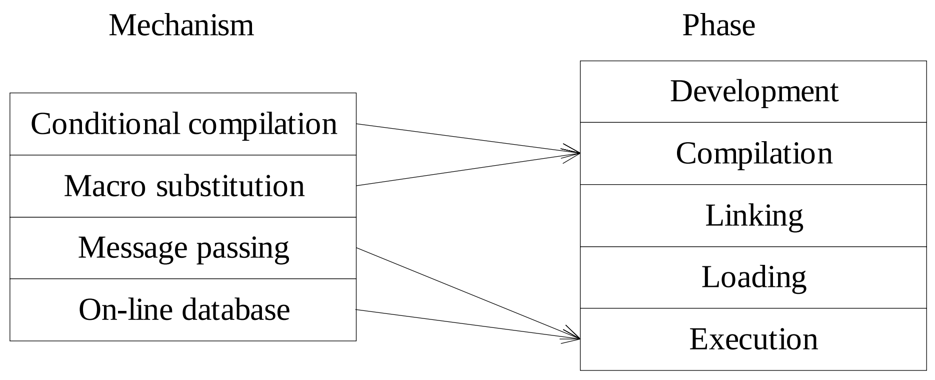

Figure 6 Name-to-value mapping and policies

The *conditional compilation* and the *macro substitution* are invoked in the compilation phase. Conditional compilation allows the user to compile selected sections of the source code. Macro substitution is used to substitute predefined text regions to corresponding values. *Message passing* is a mechanism that allows processes to send and receive messages during execution. Values could also be put into the system by reading them from an *on-line database*.

3. Problem definition

3.1 Benchmark support system: Motivation and goal

Performing benchmarks on a real-time operating system requires sophisticated methods not addressed in most existing measurement techniques. The measurement technique discussed by McRae [McR96] focuses on the benchmark program and does not address the issue of configuration of the benchmark. This may lead to a benchmarking method that requires a lot of manual work to configure the benchmark, which was a result of Rydgrens benchmarking method [Ryd97]. As a consequence, it also lacked the ability to perform *automatic range parameter iterations*. A range parameter iteration is defined as iterating the benchmark program over a sequence of monotonically increasing values. For example, increasing the frequency of interrupts during the benchmark execution.

Performing range parameter iterations is considered very useful since they may reveal information regarding unanticipated anomalies which otherwise may not be revealed. These anomalies may have a severe impact on real-time systems, because each task must be completed within its temporal scope. In contrast, a non real-time system may swallow the anomalies without dire consequences.

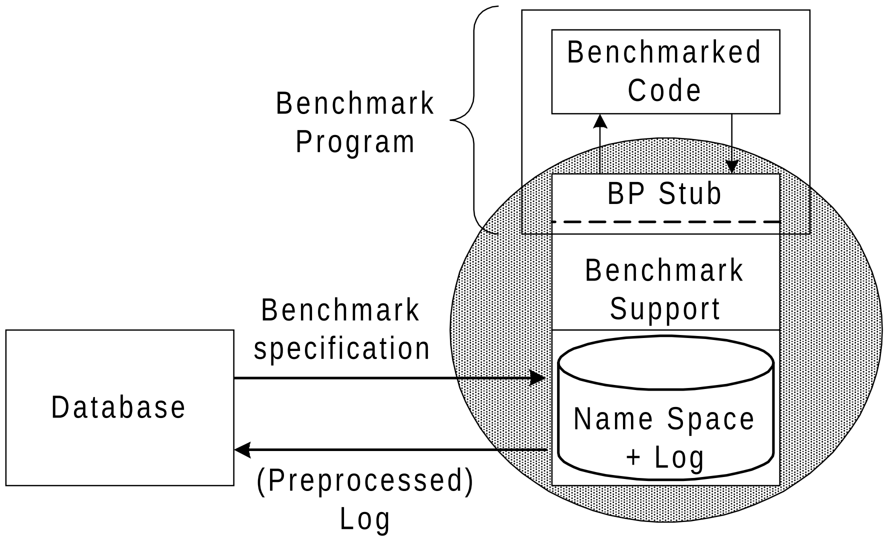

Figure 7 Benchmark support system

The purpose of the project is to design a *benchmark support system* (see Figure 7), where the user is able to specify how the measurement is to be performed. This specification may be read from a multiple user database. The benchmark support system should allow range iterating parameters and perform the measurements automatically.

Focus is put on the benchmark support and those parts of the benchmark support that need to be distributed to the target environment (BP Stub) in order to access the name space. Focus is also put on the benchmark developer interface that allows the developer to introduce developer-defined parameters to his benchmarked code.

3.1.1 Definitions

In order for the reader to fully acquire forthcoming sections, the benchmark organization is introduced. The benchmark organization has been identified and is built up from four components:

- Benchmark program - One specific implementation to benchmark something, e.g., interrupt latency.
- Benchmark - Execution of one specific configuration of a benchmark program (see Figure 2).
- Benchmark sweep: A sequence of configurations of a benchmark program. For example, the current value setting of a range iterating parameter is one specific configuration of a benchmark program.
- Benchmark suite - A set of benchmarks measuring the same thing. For example, if there are X ways to measure interprocess communication, all X ways would constitute the suite. One of the X ways would be defined as a benchmark sweep.

3.2 Software configuration switch management: Overall problem

To simplify configuration of the benchmark, *software configuration switches* must be introduced to the benchmark support. A software configuration switch is an abstraction of a mechanism that helps the developer to configure the benchmark program. The configuration switch provides the ability to control the program without having to redesign it. For example, an analogy in the real world can be a water tap that allows the user to control the water temperature and flow. The benchmark support has to provide information to the configuration switch, regarding:

- Mechanism usage: It is desired that the developer can introduce configuration switches during development without having to specify low level details about how the switches are implemented. Information about the implementation may be which configuration mechanism is to be used and in which phase it is to be invoked. For example, information that specifies the use of macro substitution in the compilation phase. The main purpose of the configuration switch is that information regarding the implementation of the configuration switch can be altered without having to redesign the developed program.
- Name to value mapping: The name to value mapping is the name space where the name of the parameter is associated to a value of a specified type. The name to value mapping must be dynamic to perform range parameter iterations.
- Binding: Information from the name to value mapping is used to associate correct name to corresponding value. The value must then be bound in the program. The binding is static in the sense that binding type cannot be changed during execution of the benchmark program. For example, there is no way to change an integer to a floating value during execution. The value is bound in the program in a specified phase (see section 3.3.1)

3.3 Configurable parameters

Defining *configurable parameters*, i.e., deciding what can be specified in the benchmark support system, is an important part of this project. All factors that affect the measurement must be identified and parameterized. There should also be support for user-defined parameters.

Example of interesting parameters might be:

- Benchmark control parameters. The user should be able to specify benchmark control parameters that affect the measurement, e.g., that the measurement is performed with an interrupt frequency of X.
- Range iterating parameters. One important feature, which has been mentioned earlier, is the ability to perform range parameter iterations. This implies specification of which parameter to change, its type, parameter start value, parameter end value, and increase interval.

3.3.1 Binding phases of configurable parameters

The benchmark program passes through certain phases in order to perform the measurement. They are described briefly in Table 1. Configuration of the benchmark program is not possible in all phases.

Phase	**Explanation**
Specification of configuration	Specification of the benchmark program using the benchmark support system.
Compilation	Compilation of benchmarked code.
Link editing	Linking and memory configuration of benchmark program object code.
Hardware Reboot	Setting target hardware to initial state.
Load	Transferring and loading of benchmark program from host to target computer.
Initialization	Initialization of benchmark program at target.
Execution	Execution of benchmark program at target.
Termination	Termination of benchmark program at target.
Analysis	Analysis of measured data.

Table 1 Benchmark phases

The most interesting phases, in views of configuration, are: specification of configuration, compilation, and execution. One problem that arises is in what phase the actual binding is to occur. When performing consecutive benchmarks, returning to an early phase (e.g. compilation) between benchmarks result in a low *throughput of benchmarks as a result of binding*, i.e., few benchmarks per hour (see Table 2). In contrast, returning to an early phase implies low *effort to introduce configuration switches*. For example, introducing configuration switches in the compilation phase is easy compared to introducing them in the execution phase, because there is no specific support for configuration switches in the execution phase. However, introducing configuration switches for use in late phases does not solve all problems. In late phases, the *control over influencing factors* is, potentially, low (see Table 2). To achieve a representative measurement it may be necessary to return to an early phase. For example, caches that cannot be flushed properly during execution affect the measurement and need to be reinitialized to produce correct results.

Phase	**Preparation effort**[1]	**Throughput of benchmarks**[2]	**Local control**
Specification of configuration	low	low	high
Compilation	medium	medium	high
Execution	high	high	potentially low

Table 2 Configuration phases

In order to achieve a highly configurable benchmark support system, the effort to introduce configuration switches, for use in all phases, must be kept low. It is desirable to have a high throughput of benchmarks, but a high throughput may be achieved at the expense of the quality of the measurement. Yet, the user should be able to specify which phase to return to, since the result may contain enough information for a comparison on a relative scale.

For example, binding parameters in the execution phase might not reveal how good an operating system is on an absolute scale, since there may be low control over factors that affect the measurement. Still, the result might reveal enough information for a comparison on a relative scale. For example, that an operating system behaves better than another benchmarked operating system, provided that the same configuration is used.

3.4 Benchmark developer interface

Letting the user specify binding phase is problematic due to the fact that all configuration policies are not available in all phases. For example, conditional compilation cannot be used in the execution phase.

A developer-friendly benchmark support system requires the use of a *transparent interface*. The interface is transparent in the sense that the developer can introduce configurable parameters without specifying details about the mechanism used to bind the value to the program. The same interface is used, regardless of configuration mechanism and binding phase. This requires a low effort to introduce configuration switches for use in all phases.

3.4.1 Measuring software in an host-target environment

Benchmarking in a host-target environment can be complicated. The host and target cooperate in a heterogeneous environment where they are physically separated. Communication problems are often the result of non standardized communication protocols. Furthermore, in order to allow configuration of the benchmark program in

[1] Effort to introduce configuration switches.

[2] Throughput of benchmarks as a result of binding.

all phases, e.g., during execution, parts of the benchmark support, the BP Stub, must be distributed to the target (see Figure 8). An example of target benchmark support is a message passing mechanism supplying the benchmark program with parameters.

Like all measurement activities, there is a big risk that the process of measuring affects the result. Our measurement tool is the benchmark program, which would not be present during the normal operation of the target, and therefore will affect the result. However, the effect is reduced by normalization (see section 2.1.2).

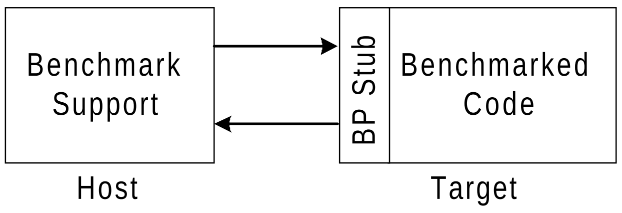

Figure 8 Benchmark host-target environment

As explained in section 2.3, the host provides the developing environment with a good set of developing tools and debugging utilities. In contrast, the target hardware is often used for a specific program and therefore provides very limited resources. In order to realize a configurable benchmark support system, it is important to identify the necessary properties or features of the target. This is not the focus of this project, and the target environment is assumed to be equipped with:

- Communication facilities: The target environment is often equipped with either serial-, or serial- and Ethernet communication facilities to communicate with the host. Serial communication is slow and affects the throughput of benchmarks if the result is logged on the target and transferred to the host.
- Hardware Monitor: The hardware monitor is a small operating system, allowing a reduced set of operations, e.g. loading and execution of programs. It can be used to load an extensive operating system, if available.
- Memory: The memory size of the target hardware is always limited. This also affects the benchmark since there is a limited space for result logs.
- Reboot facilities: To get the same influencing factors for the benchmark, the target needs to be reinitialized. This reinitialization must be achieved through software. Pressing a button to reboot the target can hardly be considered as a fully automatic benchmarking method.
- Hardware timers: McRaes benchmarking method [McR96] uses an external device to generate pulses that provide the useful information from the benchmark. Rydgren uses the target hardware timers [Ryd97].

3.5 The essence of the problem definition

The main problem is to achieve a transparent interface between the developer and the benchmark program. The user must be able to specify parameter(s) and configuration phase, without specifying configuration method. In order to achieve this, two major issues must be addressed: i) all types of configurable parameters must be identified; ii) software configuration switches must be introduced to reach a low preparation effort in all phases (see Table 2). These are the two major problems addressed in this work.

4. Approach

The design of the benchmark support can be realized in many ways. Every design must address: i) benchmark parameters and parameter types; what parameters are desirable to configure? ii) transparent benchmark developer interface; the developer is only interested in defining parameters for usage regardless of low-level details such as parameter type and binding phase.

4.1 Parameter types

It is desirable to be able to configure all parameters that are significant for the benchmark. First, definitions must be made regarding the areas were parameters are expected to be found in order to narrow the search.

According to our definition, parameters that affects the measurement, or how the measurement is being performed, are found within the following areas: design, benchmark execution, hardware, and the target operating system. These areas must be thoroughly analyzed in order to identify desirable parameters.

Design parameters are parameters specifying the behavior of the benchmark support, e.g., how the results are formatted and how they are stored. The benchmarking method used (see section 2.1.2) affects the measurement and must be analyzed in to identify specific parameters. This is done according to McRaes [McR96] definition of the benchmark method. The target hardware and operating system also affects the measurement and need to be further analyzed. The hardware will be analyzed, but influence from the operating system was analyzed by Rydgren [Ryd97] so this is demarcated. All desirable parameters are found within these areas, since this is the definition of the benchmark.

In addition to identifying parameters that affect the measurement, there is also a need to organize the parameters (see section 4.1.1) in a way that corresponds to the definitions in section 3.1.1.

4.1.1 Benchmark organization

In order to get well-structured benchmarks, the parameters must be organized in a way that conforms to the definition of the benchmark organization (see section 3.1.1). The relations between the components in the benchmark organization have been identified (see Figure 9).

Parameters for controlling the benchmark organization are essential in order to make the benchmark automatic and to enable different measurement specifications. For example, measuring interprocess communication may require several different configurations (benchmark sweeps) of benchmark programs. The reason for this is that interprocess communication can be measured in different ways. A complete measurement of the target behavior may comprise measurements of more operations than interprocess communication, e.g., interrupt latency, context switching, etc. These operations are encapsulated into the benchmark suite. The benchmark organization parameters are necessary to preserve the structure of the current specification in the result. For example, the result file must contain information about which benchmarked code that was used, etc.

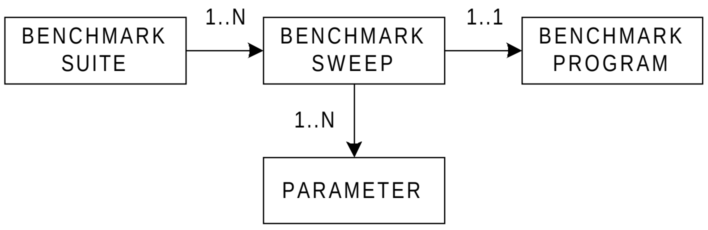

Figure 9 Benchmark organization

The best way to illustrate how the benchmarks can be organized, is by giving an example. This example is one way to let the user specify the behavior of the benchmarks, by creating a parameter specification file:

```
begin suite S1
     loops=500                    ;default parameter
     begin sweep 1                ;S1.1
          begin sweep iteration
               interrupts=0..10000,100:exe
               XPAR2=10..500,10:exe
          end sweep iteration
          code=BC1
          results=RES1
          loops=200
          XPAR1 int 20:comp
     end sweep 1
     begin sweep 2                ;S1.2
          code=BC2
          results=RES2
          interrupts=100
     end sweep 2
end suite S1
```

There is only one suite, `S1`, in this example. First, the number of iterations through the benchmarked code, `loops,` is defined to be 500. Since this parameter is specified outside a sweep, it is used as a default parameter for all sweeps that do not contain the `loops` parameter. Next, `sweep 1` begins by defining a `sweep iteration` that consists of two range iterating parameters; `interrupts` and `XPAR2`. `XPAR2` is a user defined parameter. `XPAR2` is the inner iteration and counts from 10-500 using steps by 10. Every time `XPAR2` reaches 500, `interrupts` is increased by 100. At the time it reaches 10000 the iteration is aborted. Both `XPAR2` and `interrupts` are increased in the execution phase.

`Code` illustrates which benchmarked code that should be used; in this case `BC1`. `Results` specifies where the results from the measurement will be stored. The `loops` parameter specified within `sweep 1` has precedence over the default parameter; `loops` will be associated to 200 during execution of `sweep 1`.

`XPAR1` is another user defined parameter and the type is an integer. The specification states that the value should be bound during compilation. `Sweep 2` is defined in a similar way.

4.1.2 Design parameters

The only identified design parameter is how the result file will be formatted and where it will be stored. A text based format is considered to the best format for the result file. The result file must contain information about: i) benchmarked code and its configuration; ii) corresponding benchmark sweep and suite information and; iii) time stamp information.

One of the advantages with a text based solution is that the format easily can be transformed to any other desired format.

4.1.3 Benchmark execution parameters

According to McRae [McR96], the benchmark program (see section 2.1.2) is built up from initialization code, reference code, measurement code, and benchmarked code. Desirable attributes to configure for benchmark execution are:

- Loops (see Figure 10): The number of iterations through the benchmark program.
- Benchmarked code (see Figure 10): The benchmarked code performs the actual work that is being measured. This is subject to configuration. The benchmark code developer should also be able to create user-defined parameters of the types: int, double, string, and range iterating parameters. A range iterating parameter is a monotonically increasing integer with preset boundaries. The range iterating parameter is increased between benchmark program executions in order to have the possibility to analyze its' influence on the measurement. In Figure 10, range iterating parameters are illustrated as sweeps.
- Reference code (see Figure 10): The reference code is part of the artificial load and is also subject to configuration.

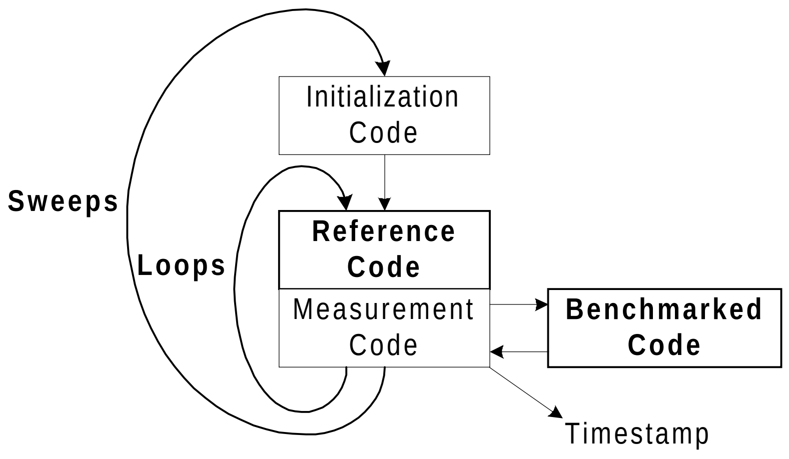

Figure 10 Benchmark execution parameters

4.1.4 Hardware parameters

Some properties of the target hardware also affect the benchmark. A simplified view of the target hardware is illustrated in Figure 11. The CPU controls all other parts and fetches instructions from the memory and decodes them. According to Motorola [Mot97], the CPU fetches recently accessed data from the cache memory. The cache memory may be internal or external to the CPU. I/O circuits handle communication between the CPU and external circuits (timers etc.). I/O circuits have the ability to interrupt the CPU.

Parameters that are subject to configuration are:

- I/O: The interrupt frequency together with the reference code form the artificial load. When an interrupt is generated, it destroys certain cache information. It is necessary to have the ability to control the interrupt frequency if caches cannot be turned on and off.
- CPU/Cache memory: Caches make the system behave unpredictable and it is desirable to have the ability to turn them on and off. Motorola states that pipelining is a technique to break operations (instruction processing or bus transactions) into smaller stages. This also makes the hardware behave unpredictable and it is desirable to turn pipelines on or off. Unfortunately, this is not always possible.

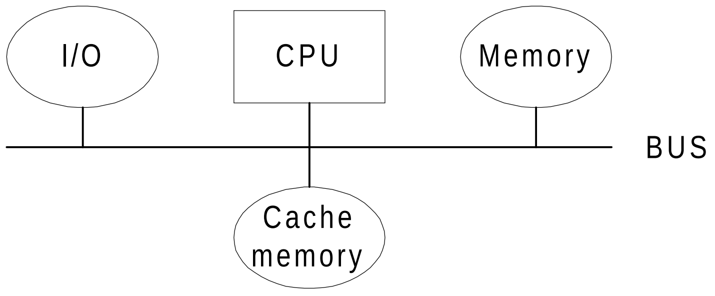

Figure 11 Target hardware

4.2 Achieving transparent benchmark developer interface

In order to achieve a transparent benchmark developer interface, the developer must be able to introduce parameters by using configuration switches. The configuration switch can be viewed as an access to the name space. The name space contains specification about the value associated to the parameter and in which phase it is going to be bound to the program. By using the configuration switch, the developer can introduce parameters to the program without having to care about existing mechanisms; altering the binding phase must not require a change in the developers' program source code.

This requires that the configuration switch can be implemented using different mechanisms whether the value binding is to occur during compilation or execution. If the value is bound during compilation the best way is to statically bind it to the program; this has the least influence on the benchmark. In contrast, value binding during execution requires name space access at the target.

Invoking different mechanisms with respect to binding phase can be achieved in two ways, using two different mechanisms (see section 4.2.2): macro substitution or conditional compilation. Any mechanism used must then be further combined with a value passing mechanism (see section 4.2.3) to read a value from the name space in order to bind it to the system.

4.2.1 Name space distribution

If a parameters' value is to be bound to the benchmark program during execution, the name space must be accessible from the target. This can be achieved in two ways; supplying a copy of the name space at the target, or remote access to the name space from the target.

Name space copy: Copying the name space (see Figure 12) means that there is a local copy of the name space at the target. If there are a lot of parameters, the name space can be very large. Since the information necessary is that related to the specific benchmark sweep in the execution phase, all information concerning the compilation phase does not have to be copied.

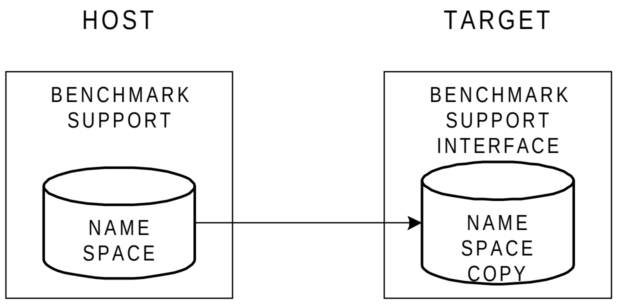

Figure 12 Name space copy

Remote access: Remote access (see Figure 13) is a way of accessing the name space at the host from the target. The target could be compared to a client in a database client/server architecture. The main disadvantage with this approach is that there would be an extra component that could affect the measurement.

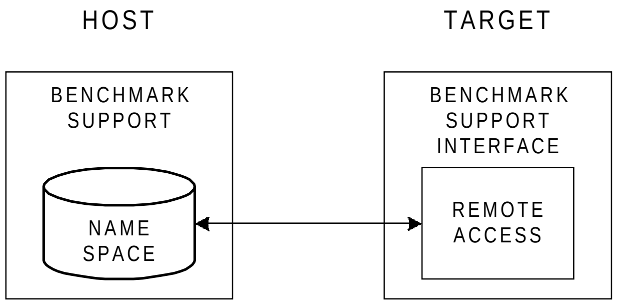

Figure 13 Remote access to name space

Summary: Both approaches, name space copy and remote access, are possible. One problem with a name space copy is how the copy should be transferred to the host without affecting the measurement. One way of dealing with this problem is to convert the name space into source code and to link it to the rest of the target part of the benchmark support. The major problem with accessing the name space from the host, is that this probably will affect the measurement. This area must be further investigated and is considered to be future work (see section 6.3.3). For now, the name space is converted to source code and linked with the target implementation of the benchmark support.

4.2.2 Macro substitution and conditional compilation

Macro substitution: Macro substitution is used by the preprocessor in the compilation phase. According to Darnell and Margolis [DM91], a macro is a text string associated to a macro name. Macros are usually used to represent numeric constant values, but can also be used to represent expressions, e.g., function calls. According to the ANSI C standard [ANS90], a macro is defined using the `#define` command, and undefined using the `#undef` command.

By controlling the macro definitions, macro substitution can be used to specify the mechanism used to bind the value to the program. If it is desirable to bind the value during compilation, the following definition can be made:

```
#define XPAR 10

x=XPAR;
```

`XPAR` will be substituted by 10 during compilation.

In contrast, if the binding is to occur during execution, the following substitution can be made:

```
#define XPAR getPARX()

x=XPAR;
```

`XPAR` will be substituted to a function call to the name space during compilation:

```
x=getPARX();
```

Conditional compilation: Conditional compilation is only possible in the compilation phase. Using pre-processor directives according to the ANSI C standard [ANS90], such as `#if`, `#else`, `#elif` or `#endif`, selected sections of source code can be left out. One advantage with conditional compilation is that the unwanted sections are left out; they do not occupy memory. The following construct could be used to control the binding phase:

```
#if XPARCOMPILATION

x=10;

#else

x=getPARX();

#endif
```

Note that a macro such as the `XPARCOMPILATION` must be supplied for every parameter to specify the binding type (static or function).

Summary: Macro substitution and conditional compilation are quite similar; imagine macro substitution being conditional compilation where the condition/decision for the definition is left to the creator of the macro. Macro substitution has the advantage of not inflicting the source code with compiler directives. Using macro substitution, there is also less work for the developer who does not have to state the compiler directives. The macro information can be generated, automatically, from a specification where it is stated which values to bind in a specified phase. Macro substitution must be combined with a value passing mechanism via name space access to bind a value in the execution phase. Binding in the compilation phase is not a problem; the macro is simply substituted for a constant generated from the benchmark support system.

4.2.3 Value passing mechanisms

As stated earlier, macro substitution or conditional compilation can be used to achieve a transparent benchmark developer interface (see section 4.2.2). With respect to the specified binding phase, different value passing mechanisms can be used in conjunction with either macro substitution or conditional compilation. A value passing mechanism can supply the benchmark program with a desired value in the execution phase. Three types of value passing mechanisms have been identified: function call, reading environment variables, and message passing.

Function call: A value can be passed through a function call. The value is returned from the function to be bound to a variable. There are two possible solutions to realize the name space access:

i) One function for each parameter, e.g.,

```
x=getXPAR1();
```

There is no need to pass any arguments to the function because one function is generated for each parameter.

ii) One function is called with a unique key for each parameter and the value is read from the name space, e.g.,

```
x=getPar(key)
```

The same function is used for every parameter, but the unique index (key) guarantees that the correct value will be returned to every parameter. This approach requires access to the name space. The correct value is located through the corresponding index.

Environment variables: Environment variables are variables global to the current environment and they are associated to values. In UNIX (csh), environment variables are set using the `setenv` command, e.g.,

```
setenv XPAR 20
```

Environment variables can be read through the `getenv()` function call during execution. Reading environment variables may be considered as a function call but is special since it requires support from the target environment. The `getenv()` function searches the environment list for the string and, providing that the string is present returns the associated value. The function returns the value as a string, so further type conversion may be necessary, e.g.,

```
int x;
x = atoi(getenv("XPAR"));
```

Generating environment variables for every benchmark parameter would be a possible approach to implement the name space and passing values to the benchmark program during execution. On the other hand, assuming that the target supports environment variables, the memory the variables are allowed to occupy is limited. Many benchmark parameters may result in lack of memory.

Message passing: Message passing mechanisms are only available in the execution phase. Message passing is used for communication between processes, using the send- and receive primitives to exchange messages. Values can be sent to the benchmark program in order to bind the received values to parameters. Using message passing to pass values to the benchmark program, requires communication between the benchmark program and some other process that can access the name space.

Summary: There is a need to reduce the factors that influence the measurement. Message passing requires maintenance of another process that communicates with the benchmark program and may affect the measurement. Reading environment variables is a possible approach but is limited in memory size and requires operating system support. Using function calls in conjunction with macro substitution is a feasible approach and can be realized in three different ways (see section 4.2.4).

4.2.4 Generating functions through macro substitution

There are three different ways to implement macro substitution for name space access and value binding in the execution phase: i) passing a macro argument; ii) assigning a function for each substituted parameter; iii) assigning the same function for all parameters, but calling the function with a unique identifier for each parameter.

Table 3 illustrates how macro substitution can be achieved, depending of what phase the binding is specified to occur. The *program source* column represents the specification in the program source code and the *macro definition* column represents the appearance of the automatic generated definition file. Focus is put on substitution in the execution phase, since the macro only has to be substituted for a constant in the compilation phase.

#	Phase	Program source	Macro definition
1	execution	`x=GET(XPAR)`	`#define GET(z) getPar(z)`
2	compilation	`x=GET(XPAR)`	`#define GET(z) z`
3	execution	`x=XPAR`	`#define XPAR getParXPAR()`
4	compilation	`x=XPAR`	`#define XPAR 10`
5	execution	`x=XPAR`	`#define XPAR *(int*)getPar(key)`
6	compilation	`x=XPAR`	`#define XPAR 10`

Table 3 Macro definitions

Passing a macro argument (1) is a flexible solution to parse a macro and to substitute the macro to something else, e.g., a constant. The drawback with this approach is that there is no way to check the type of the macro, i.e., int, float, etc. This method also requires further substitution. For example, if `XPAR` should be associated to the value `10`, `x=GET(XPAR)` must first be substituted to `x=XPAR`. After that, `x=XPAR` must be substituted to `x=10`. There also must be conditional compilation directives that prevent `x=GET(XPAR)` from being substituted directly to `x=GET(10)`.

Assigning a function for each parameter (3) is also a possible approach. The major drawback is that assigning a unique function for each parameter will result in many functions if there are a lot of parameters. Queries to the name space, such as increasing the value of a parameter, will render a new function for each parameter.

Assigning the same function for all parameters (5) is possible if a unique index, (`key`), is passed as an argument to the function, `getPar(key)`. With this solution, type checking can be performed.

Summary: Passing macro arguments (1) is not a suitable approach, since type checking cannot be performed. The two remaining approaches, generating a function for each parameter and one function for all parameters, are both possible alternatives. Both alternatives will be implemented and tested for further evaluation (see section 5.2.2).

5. Results

5.1 Parameter types

A representation of the benchmark organization has been found. This makes it possible to organize parameters according to the definitions in section 3.1.1.

The areas defined to contain parameters were analyzed and parameters were found regarding: design, benchmark execution, and hardware.

Design: Storage of the result file.

Benchmark execution: Loops, benchmarked code (including user defined parameters), and reference code.

Hardware: Interrupts, caches, and pipelines (if possible)

The target operating system was not analyzed since this work was done by Rydgren [Ryd97]. The identified parameters cover all possible parameters that are desired to configure, since the areas defined to contain parameters were thoroughly analyzed. The ability to let the developer create user-defined parameters, including range iterating parameters, has also been covered.

5.2 Transparent benchmark developer interface

5.2.1 Name space distribution

The name space must be accessible at the target in order to bind values in the execution phase. The two methods examined were: i) a copy of the name space and; ii) remote access. The method chosen is to copy the name space on the grounds that this method affects the target environment least. A minimum copy of the name space is linked and transferred along with the target part of the benchmark support. Only information related to the executing benchmark sweep need to be copied. As stated in section 4.2.1, this area has to be further examined.

5.2.2 Macro substitution and function calls

Macro substitution is chosen as the best alternative to achieve the transparent developer interface. Two different ways to combine macro substitution and function calls were identified as equally good. Necessary implementation is made in order to evaluate them. Both implementations try to bind the integer 10 during compilation and 700 during execution. Only integers are used in these examples, but the full implementation should be able to handle the types integers, floats, doubles, and strings. Both macro substitutions and function calls will be automatically generated by the benchmark support system.

The following implementation illustrates the alternative with one function call for each parameter:

```
#define CONFPAR1 10
#define CONFPAR2 getparCONFPAR2()

int bc__CONFPAR2=700;

int getparCONFPAR2()
{
  return bc__CONFPAR2;
}

void main(int argc,char **argv)
{
  int x,y;
  x = CONFPAR1;
  printf("Value of x is: %i\n",x);
  y = CONFPAR2;
  printf("Value of y is: %i\n",y);
}
```

The result of execution is:

```
Value of x is: 10
Value of y is: 700
```

In this example, `x = CONFPAR1` is substituted to `x = 10`, and `y = CONFPAR2` is substituted to `y = getparCONFPAR2()`. The implementation shows that this is a possible approach, but many parameters result in a lot of functions. All functions will be automatically generated, so the only drawback is that they occupy a lot of memory.

Next, the alternative with one function call for all parameters is implemented.

```
#define CONFPAR1 10
#define CONFPAR2 *(int*)getpar(5)

int list[12]={200,300,400,500,600,700,800,900,1000,1100,
1200, 1300};

void *getpar(int index)
{
     return((void*)&list[index]);
}

void main(int argc,char **argv)
{
  int x,y;
  x = CONFPAR1;
  printf("Value of x is: %i\n",x);
  y = CONFPAR2;
  printf("Value of y is: %i\n",y);
}
```

The result of execution is:

```
Value of x is: 10
Value of y is: 700
```

Here, `x=CONFPAR1` is substituted to `x=10`, and `y=CONFPAR2` is substituted to `y=*(int*)getpar(5)`. The `getpar` function is called with an identifier that is specific for the parameter, 5 in this case. The `getpar` function returns a value that must be typecasted and dereferenced to handle all desirable parameter types. By using this type of macro substitution, typechecking is guaranteed and there only have to be one implementation of the name space function call. The drawback with this solution is that the typecasting might seem a bit confusing and that a unique key has to be generated for each parameter.

Summary: Both methods have their advantages and drawbacks. The second alternative, one function call with unique keys for all parameters, is chosen. The first alternative will render many functions if there are many parameters. Also, when using the second alternative, there only have to be one additional function call to update the name space when using range iterating parameters. With the first alternative, there would have to be one additional function call for every range iterating parameter.

5.2.3 Example of benchmarked code

The use of configuration switches and macro substitution is illustrated in an example of benchmarked code. This example measures the timedelay of remote procedure calls by sending and receiving a signal to another process. The timestamp generation is not included in the example. This example assumes that the developer has specified the use of a parameter called `XPAR2` and that the value is to be bound in the execution phase (see example of specification file in section 4.1.1).

A user-defined parameter, `bc__xpar2`, is generated and defined globally. `XPAR2` is substituted to a `getPAR(key)` function call, since it is defined as a range iterating parameter. The developer only have to introduce the variable `bc__xpar2` to his code; this parameter is automatically generated from `XPAR2` in the specification file.

This example will display what influence, the message size has on the performance of the operating system.

```
int bc__xpar2;

void initialization(){
     bc__xpar2 = XPAR2;
}

void benchmarked_code(){
     int F_RPC_MessageSize=bc__xpar2;
     SIGSELECT anySig[]={0};
     union SIGNAL *sig;
     sig = alloc( sizeof( test_Msg)+F_RPC_MessageSize,
test_MSG );
     send( &sig, F_RPC_server_ );
     sig = receive( anySig );
     free_buf( &sig );
}
```

5.3 System design

The achieved results make it possible to propose a feasible design of the benchmark support system (see Figure 14).

First, the specification of the benchmark suite is read into the memory at the host. The information from the specification is the name space that contains the parameter settings that will control the benchmark. The information is processed and converted to a suitable format. The definitions needed for the macro substitution are then created according to the parameters' phase information in the specification. Next, the compilation is performed and specified parts are compiled and linked. For example, specified parts may be the benchmarked code or the reference code.

The executable program is then transferred to the target. The executable program contains more than just the benchmark program. It also contains a minimal copy of the name space and the code needed to iterate the benchmark process at the host. The name space copy contains the necessary information to initialize and update the parameters within their range boundaries. The program is started at the target while the host is waiting for the log, i.e., the timestamps from the measurement. The execution of the target program starts with the initialization code where necessary values are read from the name space copy and bound to the program. The reference code, measurement code and benchmarked code are then iterated the number of times specified by the loop parameter. After this, the log is transferred to the host and cleared at the target in order to provide memory for the next log. If numerous iterations have been specified by declaring range iterating parameters in the specification, corresponding values are updated (increased) in the name space copy. Note that updating the name space copy at the target only concerns those range iterating parameters that are specified to be bound in the execution phase. Range iterating parameters specified to be bound in the compilation phase are updated later at the host. If the parameters' range boundaries have been reached or if no range iterating parameters have been specified, the target is rebooted and returned to its' start state.

At the host, the log file is received from the target and stored to disk. If the target iteration is finished, the host awaits another log from the target. Otherwise the host updates the name space and continues its own iteration.

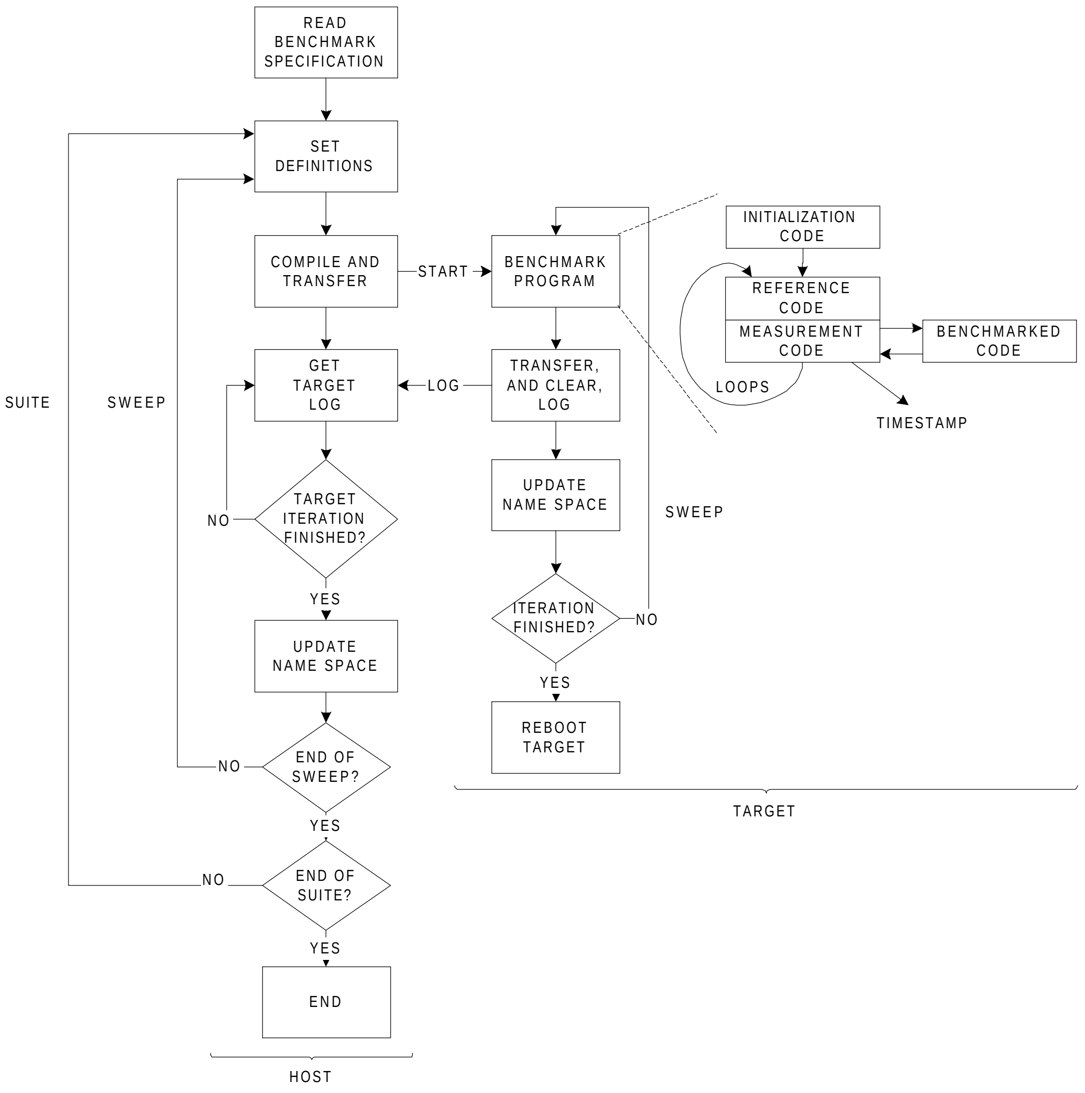

Figure 14 Benchmark system design

5.4 Name space representation

The information regarding the parameter specification must be interpreted and prepared for easy access. The parameter name and corresponding information are stored in the name space. A unique identifier is created for internal use. Apart from returning correct values for binding, further name space services are queries regarding range iterating parameters. For example increasing a value within the range interval, queries whether the maximum iteration value has been reached, etc.

A feasible representation of the name space is illustrated in Figure 15.

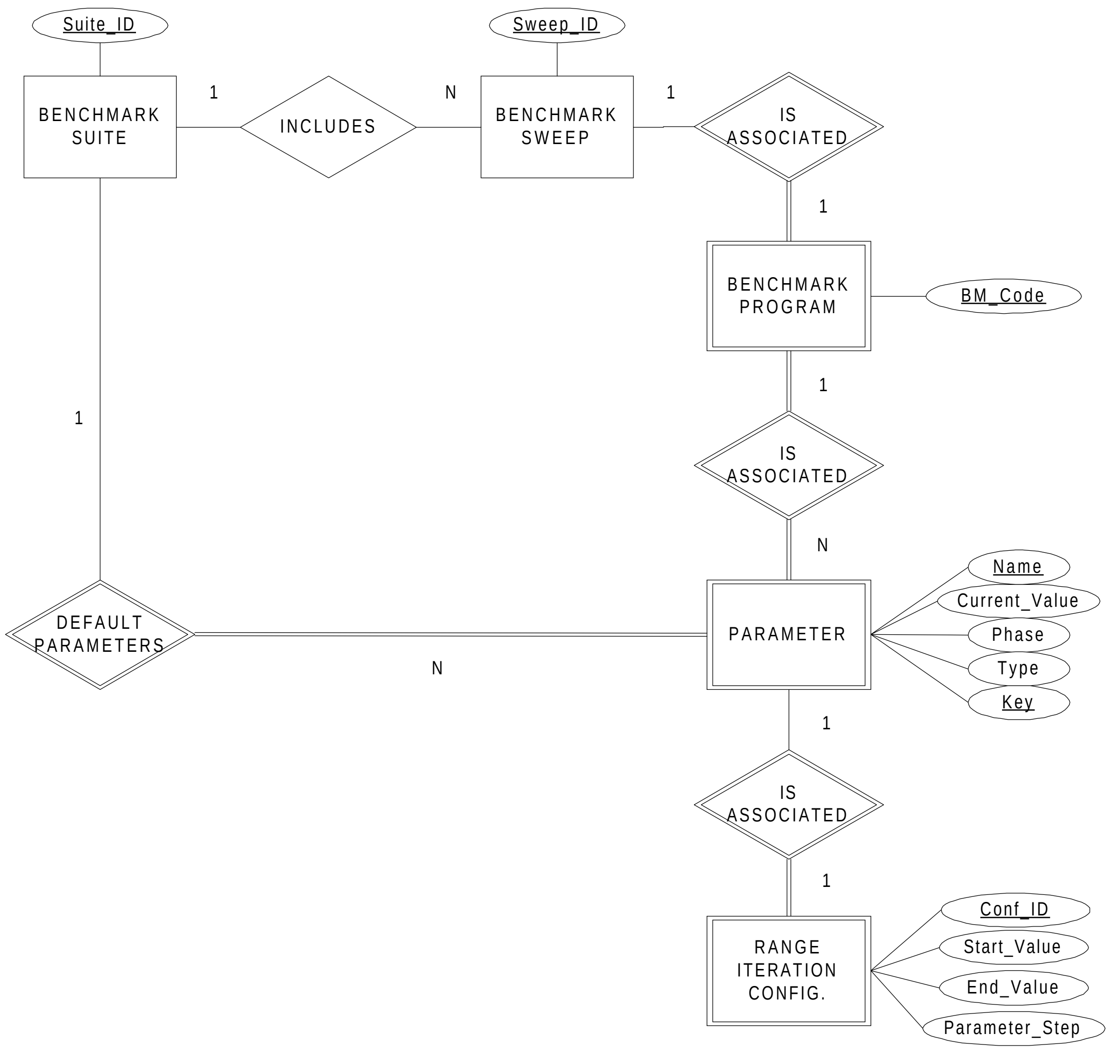

Figure 15 ER representation of name space

5.5 Related work

5.5.1 Benchmarks

The benchmark method used is based on the benchmark method suggested by McRae [McR96]. McRae uses an external timing device. Rydgren [Ryd97] uses the hardware timers instead and this work is based on Rydgrens method. Neither McRae nor Rydgren strives towards configurability, which is the focus of this project. The benchmark method is similar, but the benchmark program has been extended to comprise configuration of parameters and range iterating parameters.

5.5.2 Software configuration

A lot of work has been done in this field. Many solutions to runtime dynamic configuration has been based on CONIC [KM+83]. CONIC is a language specific environment that supports runtime replacement of modules. Durra [BW+93] is another runtime support system, specific for distributed applications.

The list can be made even longer, but common for most approaches are: i) they require complex systems and specific languages that support runtime configuration; ii) they do not have real-time support.

These approaches towards dynamic configuration could not be applied since they would affect the measurement.

Moreover, the benchmark support system is quite unique since it must handle configuration during either compilation or execution. Configuration during execution only is not a possible approach since there is a potentially low control over factors that affects the measurement.

Still, CONIC and DURRA are considered to be the most related work.

6. Summary

6.1 Conclusions

The purpose of this project is to design a benchmark support system to aid the user in configuration of a benchmark, or a suite of benchmarks. It should also help the developer to create own benchmarks and introduce parameters that easily can be configured. It is our opinion that the suggested design is suitable because it provides solutions to the major problems: i) what parameter types are desirable to configure and; ii) how the benchmark developer interface can be made transparent.

6.1.1 Parameter types

First a definition was made regarding what areas influence the benchmark or how the benchmark is performed. This definition had to be made in order to narrow the search for parameters. The areas were categorized into: design, benchmark execution, hardware, and operating system. These areas were analyzed and parameters were identified, apart from the operating system that already has been analyzed by Rydgren [Ryd97]. All relevant parameters in the defined areas were identified and a representation, in order to organize the parameters, were also found.

6.1.2 Benchmark developer interface

The transparent benchmark developer interface is achieved by using software configuration switches. The software configuration switches are used by the developer to introduce parameters in the benchmarked code. Information about parameters introduced this way can be specified later. For example, that a certain value should be bound to the benchmarked code in the execution phase.

The implementation of the software configuration switch was preceded by both a theoretical and a practical study. Different mechanisms were evaluated and implemented in order to find a suitable approach. The result is that the best approach is to use macro substitution in conjunction with function calls for name space access. In order to enable value binding in the execution phase, a copy of the name space is distributed to the target environment. The macro definitions and function calls are automatically generated.

6.2 Contributions

A need for a configurable benchmark method was discovered; analyzing the effect of range iterating parameters may help to predict unanticipated behavior of a real-time system.

In order to configure the benchmark, the benchmark must be parameterized and desirable parameters must be identified. McRaes benchmark method was analyzed and all desirable parameters were identified. Hardware parameters were also identified, but these may change since this is an area under continuos development.

A way of minimizing the effect on the measurement when accessing the name space was also discovered: i) constant values are bound during compilation and ii) name space access during execution is achieved through function calls.

6.3 Future work

This work focused on the design of the benchmark support system and the implementation of a prototype. The aim was to get a portable, extensible, and adaptable solution. The extensibility goal is considered to be fulfilled since a developer can use the software configuration switches to add new code that can be configured.

6.3.1 Portability issues

In this context, portability is the effort needed to move the benchmark support system from one configuration of host-target computer to another. Portability could be considerably enhanced by using a real-time operating system adapter. The benchmark support system is implemented on the real-time operating system OSE Delta and uses operating system specific system calls. A real-time operating system adapter could be implemented using the guidelines suggested by Ivarsson [Iva97].

Moreover, the target environment is assumed to have certain properties or features (see section 3.4.1). It would be of great interest to analyze what effect constraints on one or more of these properties or features will have on the benchmark support system. For example, how can the benchmark support system be realized without a hardware monitor.

6.3.2 Adaptability issues

Adaptability could also be improved, since no thorough analysis has been conducted regarding the coupling of the benchmark support system to existing development tools and development environments. For example, it may be desirable to save the result in a format specific for some analyzer tool.

6.3.3 Name space distribution

The name space copy is realized by converting the name space to source code and link it to the target part of the benchmark support system. This is a complicated way to distribute the name space to the target. Alternative methods need to be investigated.

Acknowledgments

I would like to thank the distributed real-time systems (DRTS) research group at Högskolan, Skövde, for their contribution and support; especially my supervisor Jonas Mellin for his invaluable comments and faith in me. The comments from Fridrik Magnusson and Ragnar Birgisson were also highly appreciated.

I would also like to thank my girl-friend, Marie, for her love, support, and durability.

Finally, special thanks to the Fraunhofer IIS institute for the MP3 format; my collection of MP3:s was my only company during all those late nights…

References

[ANS90] (1990), Programming languages - C, ISO/IEC JTC 1/SC 22/WG 14, ISO/IEC 9899:1990

[BW+93] Barbacci M.R., Weinstock C.B., Doubleday D.L., Ardner M.J., Lichota R.W. (1993), Durra: a structure description language for developing distributed applications, Software Engineering Journal, Vol. 8, No. 2, March 1993

[DM91] Darnell P. A., Margolis P. E. (1991), C - a software engineering approach, 2nd edition, Springer-Verlag

[FP96] Fenton N. E., Pfleeger S. L. (1996), Software metrics: A rigorous and practical approach, 2nd edition, International Thomson computer press.

[Gra96] Grace R. (1996), The benchmark book, Prentice-Hall ch. 1

[Iva97] Ivarsson B. (1997), Guidelines for design of an application group specific API for distributed real-time operating system, HS-IDA-EA-97-106, Department of Computer Science, Högskolan Skövde

[KM+83] Kramer J., Magee J., Sloman M. (1989), Constructing Distributed Systems in CONIC, IEEE Computer Society Press, Vol. 15, No. 6, June 1989

[McR96] McRae E. (1996), Benchmarking real-time operating systems, Dr Dobbs journal, May 1996

[Mot97] Motorola Inc. (1997), MPC750 RISC Microprocessor User's Manual, 1997

[Nee93] Needham R. M. (1993), Names ch.12, Distributed systems ch12., Sape Mullender (editor), Second edition, Addison-Wesley

[Ryd97] Rydgren E. (1997), Systematic benchmarking of real-time operating systems for telecommunication purposes, HS-IDA-EA-97-204, Department of Computer Science, Högskolan Skövde

[Sch93] Schütz W. (1993), The testability of distributed real-time systems, Kluwer Academic Publishers

[TP95] Tuulari E., Päivike H. (1995), Välj rätt realtidsoperativsystem till mikroprocessorn, Elektroniktidningen, Nr 6, 27-30

Appendix A - Source code

Appendix A is only available upon request from the library at Högskolan Skövde.

E-mail: biblioteket@bib.his.se