The Architect's Workbench

Bravenboer, R.G.

Award date:
1990

Department of Electrical Engineering
Digital Systems (EB)

Master thesis:

The Architect's Workbench

by R.G.Bravenboer

Supervisor : Prof. ir. M.P.J. Stevens
Coach : Ir. W.J. Withagen

Period : February - October 1990

Eindhoven University of Technology

ABSTRACT

This master thesis describes the Architect's Workbench, a high level simulator that supports a top-down architectural analysis of buffering architectures. For this purpose a range of instruction set architectures is available for simulation.

This thesis will also describe problems that occurred when the workbench was ported from the Vax/Sun environment to the Apollo Domain environment, and the solutions that have been created.
Most of the problems occurred because of the difference in compilers and run time environment. Especially the connections between pascal and C programs caused problems.

Some thought is given to the adaptation of the workbench to a tool with which the performance of the buffering architecture for the C-processor currently under development in the Digital Systems group of the department of Electrical Engineering at Eindhoven University can be investigated. The changes to be made are described.

CONTENTS

0. Introduction 4

1. Introductory chapter 6
   1.1. Research on architectures 7
   1.2. The buffer hierarchy 9
   1.3. Performance measurement 13
   1.4. Conclusions 15

2. The Architect's Workbench 16
   2.1. The compile time software 16
   2.2. The run time software 21
   2.3. The Architect's Workbench 22
      2.3.1. Instruction traffic simulation path 24
      2.3.2. Data traffic simulation path 25
      2.3.3. Mixed traffic simulation path 25
   2.5. Conclusions 26

3. Virgin 27
   3.1. New features 27
   3.2. Using virgin 28
   3.3. Conclusions 33

4. Testing the Architect's Workbench 34
   4.1. U2a 34

4.2. The u2a pascal library 36
4.3. The instruction traffic simulators 39
4.4. The data traffic simulators 40
4.4.1. First level buffer testing 41
4.4.2. Second level buffer testing 42
4.5. The combined traffic simulator 42
4.6. Conclusions 43

5. Adapting for the C-processor 44
5.1. Simulating the stack cache 44
5.2. Adapting hllref 46
5.3. Conclusions 47

6. Conclusions 48

Literature 50

Appendices:
1. List of extensions used by the Architect's Workbench 51
2. Virgin 55
3. Cache.conf 68
4. Gethllref & ucode.h 70

Chapter 0: INTRODUCTION

At Eindhoven University of Technology the Digital Systems Group (EB) is occupied with research in the area of structured design and realization of digital systems and VLSI building blocks. Currently part of this group is involved in the C-processor project, designing a highly advanced microprocessor tuned on supporting high level programming languages.

Processor - Memory traffic is a serious constraint for computer systems. Microprocessors in particular are vulnerable to performance degradation because of pin/bandwidth limitations. To satisfy memory bandwidth requirements, and achieve a high processor throughput, buffering of both instruction and data traffic is useful. Buffering can be very successful, because processor requests obey the principle of locality. All requests are both temporally and spatially associated with each other. Optimal reduction of traffic is achieved when processor requests for data or instructions are satisfied from the buffer instead of from main memory. A means of deciding which buffering configuration produces such an optimal reduction, given the architectural characteristics of the processor involved, is thus needed.

In the United States of America, at Stanford University, an integrated software tool has been developed which simulates the behavior of various processor architectures with various buffer configurations. This integrated architecture evaluation tool, which is called the Architect's Workbench, describes the performance and utilization of buffer hierarchies through the creation of reference patterns. These patterns are registered while simulating the processing of standard test programs, benchmarks, for a particular buffering scheme.

Within or group there is an urge for the C processor mentioned earlier to be simulated with various cache designs. These simualtions will greatly reduce the number of tests on realizations in hardware that are otherwise needed to select the optimal buffer configuration for this processor, and will thus save money. For these simulations to be realistic however, timing information in the simulation of data traffic is needed. This thesis will describe the adaptations of existing software and the development of new software for the Architect's Workbench to be able to perform such simulations.

Chapter 1: INTRODUCTORY CHAPTER

This chapter introduces the basic concepts on which research using the Architect's Workbench described in this thesis is built, such as processor-to-memory traffic, and vice versa, traffic buffering, and tradeoffs in buffer design. It also describes several buffering models, as far as relevant for, or available from, the Architect's Workbench.

When analyzing traffic and traffic buffering, in order to optimize traffic handling, one would like to know what constraints a chosen processor architecture will impose on the buffering and memory systems. In order to be able to specify these constraints some performance figures are needed. In paragraph 1.3 will be discussed how these figures can be obtained from the measurements the Architect's Workbench provides.

The amount of traffic needed to sustain full processing speed determines the memory bandwidth requirements. Memory bandwidth is the amount of data the memory system can deliver every second. Memory bandwidth can be influenced strongly by the following issues:

- A Buffer Hierarchy
- Compile Time Software
- The Instruction Set Architecture

1.1 Research on architectures

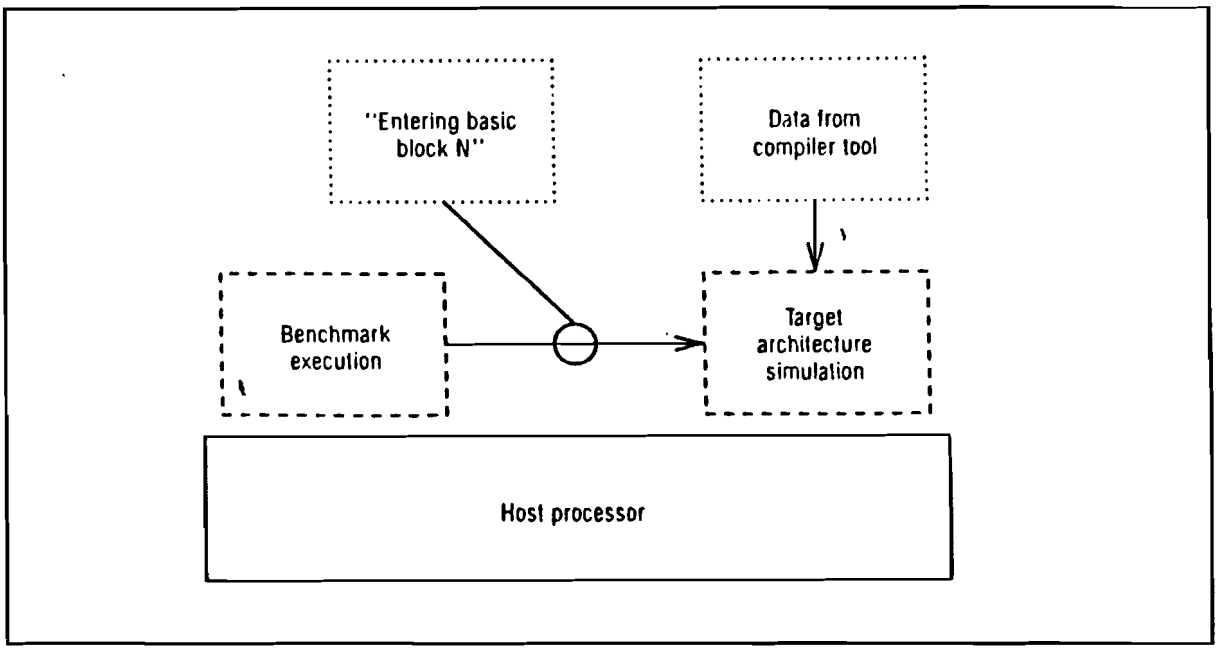

Figure 1.1: The CARA methodology.

The architecture of a processor forms the link between the hardware implementation (design, technology) of a microprocessor and the instruction set that it supports. The most straightforward way to examine a particular architecture is to examine the operation of an existing implementation of such an architecture in hardware. A cheaper approach however is to simulate that architecture in software, on existing hardware, through examination of the performance of its instruction set architecture. Analytical approaches usually involve measurement of some aspects of typical programs, and application of these aspects to models of the simulated architecture. CARA, Compiler Aided Research on Architectures, in it's general form requires that a set of standard programs, called benchmarks, are written in high level languages. Analysis of the performance of these high level language programs is then performed by compiler tools. The benchmarks are first translated to an intermediate format, and then broken up into basic blocks. Each basic block, a structural unit of a program as will be explained in more detail in chapter 2, is then analyzed separately, and the results are stored in tables. Furthermore a call to a trace routine is inserted at the

start of each basic block. The benchmark is then compiled and executed on a host machine. Program flow is registered through the inserted calls to trace routines, on entry to each basic block. A final simulator then combines this trace with the stored block characteristics to construct an overall simulation. The main reason for separating the trace and block info has been the attempt to reduce the data file sizes. (Even so trace files of up to 3.5 Megabytes do occur when executing certain benchmarks, like the pascal compiler 'pasm').

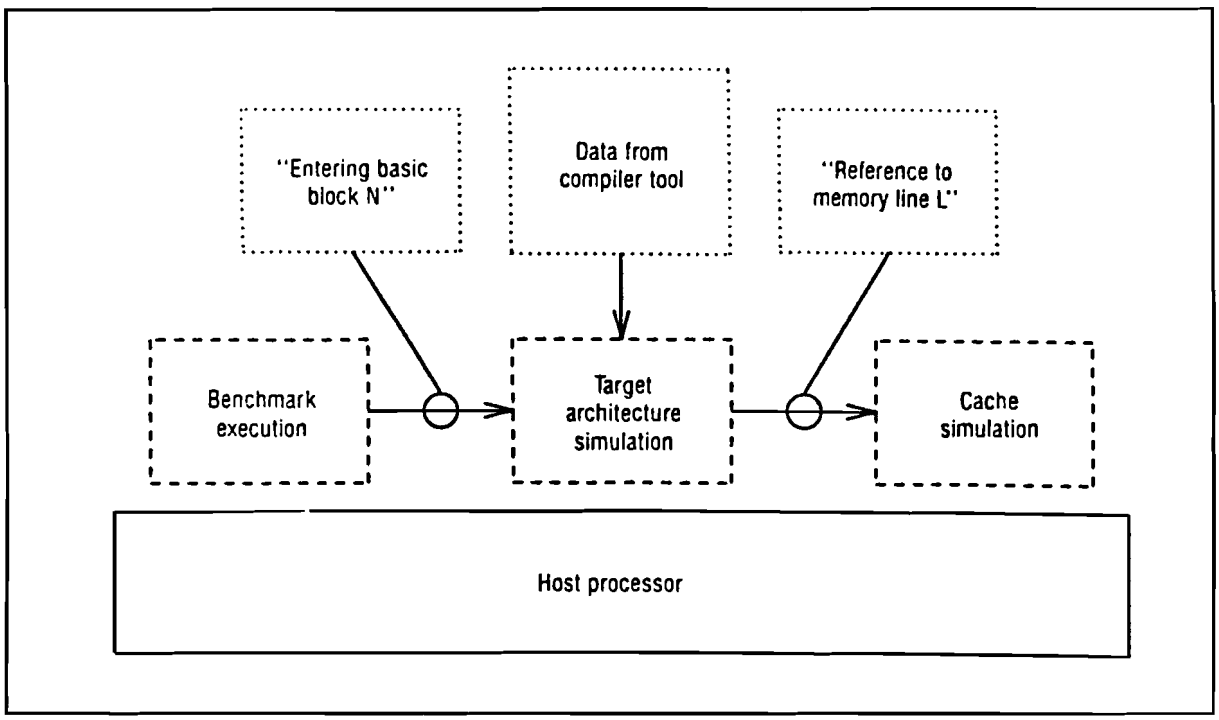

Figure 1.2: The extended CARA methodology.

CARA however may be used for more then just processor architecture analysis. As in the Architect's Workbench it may provide a whole range of buffer and cache simulators. In a realistic simulation also some general hardware parameters such as register sets, bus widths, cache sizes and cache organization will be specified.
If during the simulation of the target architecture yet another trace is generated, one which contains address information, then this trace can be used to feed these simulators. Figure 1.2 shows this extended CARA methodology, as implemented in the Architect's Workbench.

The instruction set architecture greatly influences the magnitude of the instruction traffic when counting the number of instructions read. For example there are architectures designed to minimize instruction traffic, using more complex instructions, and at the other end there are the reduced instruction set computers, which are designed to minimize cycle time. Therefore when examining the performance of architectures through comparison of their instruction counts one should take their expected actual cycle times into account.

1.2 The buffer hierarchy

The principle of buffering, especially when using caches, is based on locality, i.e. related that is data is positioned close to each other. There are two forms of locality. Temporal locality, which is best for data traffic, and spatial locality, which is best for instruction traffic. A buffer hierarchy can use this principle to reduce the read and write times through keeping often needed code and data close to the processor, thus minimizing the execution time of programs. This hierarchy thus provides an interface between processor and memory. It may be ordered in multiple layers in order to combine several buffering mechanisms, or to combine the advantages of large buffers with those of small buffers, or to combine on and off chip buffer parts.

One kind of buffer is the **single register set**. It's contents are addressed through specification of the appropriate register within the set. This minimizes both the instruction traffic (shorter address, less bytes) and the data traffic (shorter distance, the amount of cycles needed to acquire the data drops). The main parameters used to specify single register sets are set size, functionality and instruction formats. The effectivity of a single register set is greatly influenced by the way the data are allocated to its registers.

Because registers are an integral part of the instruction set architecture, a compile-time program has to exploit them, and thus has a great influence on the performance.

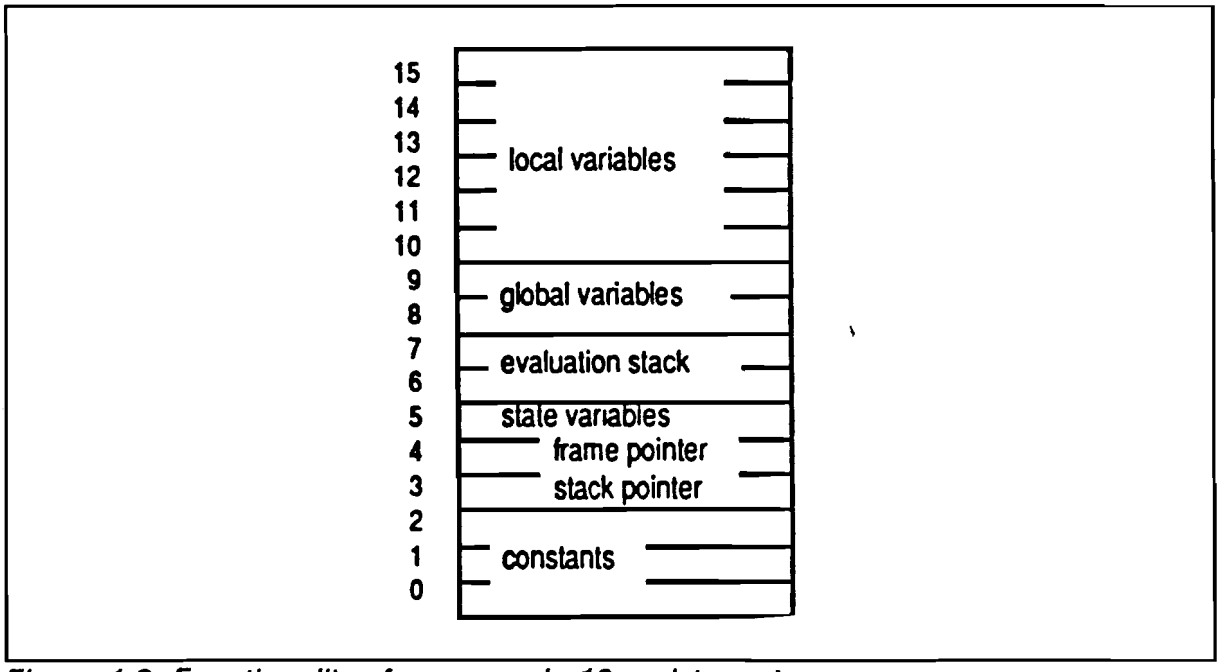

Figure 1.3: Functionality of an example 16-register set.

The format of the instructions only influences the instruction traffic.

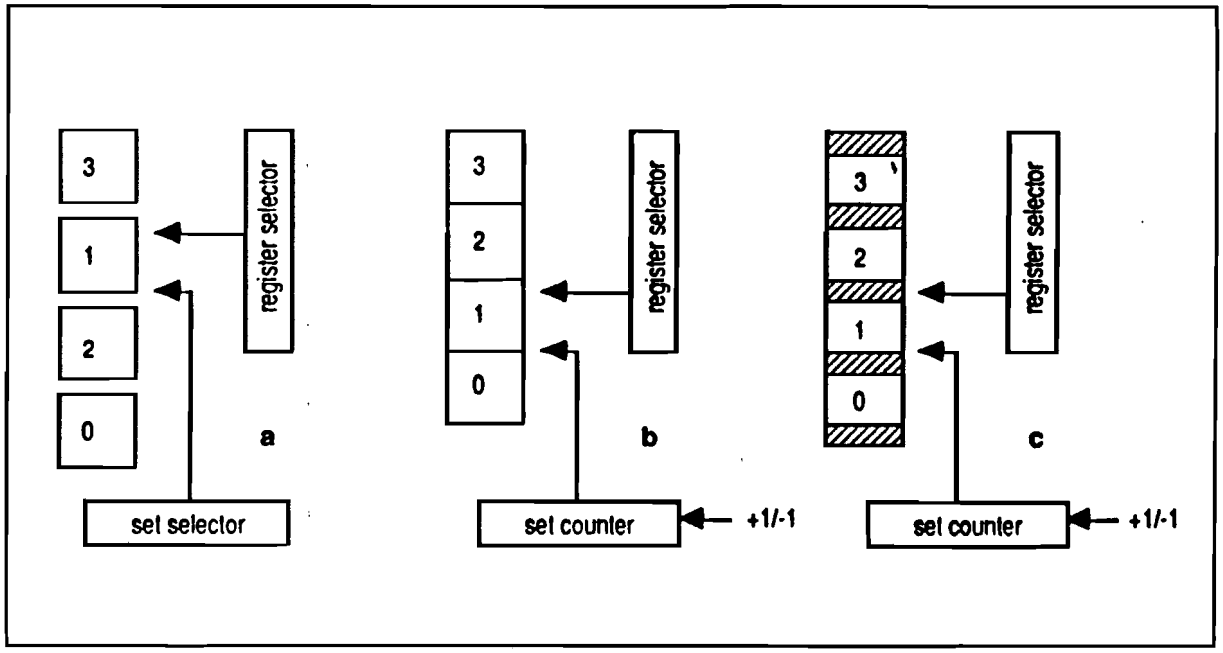

Figure 1.4: Multiple register set organizations.

The functionality of registers greatly depends upon the operations allowed on a register. If all operations are allowed on it, the register is a general purpose register (GPR), otherwise it is a special purpose register (SPR). The effectiveness of a register allocator is influenced by both number and functionality of the registers in a set. Figure 1.3 shows the layout of an example 16 register set. Most register allocators have access to the general purpose registers, the upper 8 registers. The amount and implementation of additional registers varies with the architecture involved.
A more sophisticated buffer is the **multiple register set**, introduced to save time by not saving registers on context switches such as interrupts, proces switches, messages and procedure calls, but using another register set when entering a new context. Some additional parameters for multiple register sets, in addition to those of single register sets, are parameter overlap and strategies for underflow and overflow handling and aliasing detection. Figure 1.4 gives some examples of multiple register set organizations. Configuration A provides a number of completely separate sets which may be allocated using pointers. Organization B switches sets at procedure calls and returns, and allocates the sets in a stack like manner, using a counter. The stack is oriented in a circular manner, the head is attached to the tail, thus allowing to store the bottom of the stack when room for the top is needed. This provides a cheap implementation of the least recently used replacement strategy. Organization C behaves like register set B, but uses overlap of register sets to pass parameters between procedures.

A **stack buffer** (contour buffer, stack cache) resembles the B type multiple register set, except for the set sizes, which are not fixed. Every set allocates as much of the stack as it needs, including room for static and dynamic links. Like multiple register sets B and C it is constructed head to tail. Additional parameters that specify the stack buffer are allocation/deallocation strategy and memory layout.

Another way to buffer data is through the use of **caches**. A cache is a small piece of fast accessible memory, preferably close to the processor, which tries to contain the data most likely to be accessed in the near future. Caches come in a range of fetch,

write back and replacement strategies. Most conventional caches use line fetching on demand.
Data which has been changed has to be written back to memory. One may choose to do this on replacement (simultaneous read and write) or on changing the data (write through).
The main parameters to specify a cache are size, line size, degree of associativity, replacement strategy. A cache with an associativity of one is a direct (linear) cache, each memory line fetched has one possible location to be stored in the cache. An associativity of N means that each line of memory may end up into one of N cache lines. A fully associative cache allows a memory line to be loaded into any one of the cache lines.
Choosing between accessible lines may occur at random, or replacing the least recently used (LRU), or applying a first in first out (FIFO) strategy.

The traffic between a processor and it's memory system consists of two fundamentally different kinds, instruction traffic and data traffic. The instruction traffic is a one way stream, from the memory to the processor, it's contents are read and executed, but not changed. Instruction traffic is mostly a consecutively stored and executed stream of data. Buffering of this data therefore can best be done by a cache.

The data traffic is necessarily two way since data may be changed, and thus has to be written back to memory some time. Also data traffic contents may originate from, or be directed to, different addresses in memory, or elsewhere. Therefore data traffic streams require more complicated ways of buffering, and thus have to be simulated using different simulators.

Approximately 75% of all data references are simple local, simple global, and run-time management variables. Most of these references are local, confined to the scope of the active procedure. Because of this limited scope, a relatively small buffer may capture most references.

1.3 Performance measurement

In order to be able to compare the performance of different buffer architectures and instruction set architectures, the definition of performance measures is important. Some performance measures one could think of are:

- The hit ratio R_{hit}, (1 - R_{miss}), defined as the ratio of the number of times the processor finds a memory reference in cache to the total number of memory references.
- The miss penalty, the amount of cycles lost when a memory reference can not be found in cache.
- The traffic ratio R_{traf}, defined as the ratio of the number of bytes transferred between cache and main memory and the number of bytes transferred between the processor and the cache.
- The service time, the amount of cycles needed to run a trace.
- The access time, the amount of time the buffer and memory actually need to provide for a hit or a miss.
- The chip surface used, which especially for an on-chip buffering mechanism is a constraint.

Three parameters related to cache performance which can be measured quite easily, are the miss rate, the traffic ratio and the service time. The miss rate and traffic ratio are derived as follows:

$$R_{miss} = T_{bypass}/T_{base}$$
$$R_{traf} = T_{mem}/T_{base}$$

T_{base}, the baseline traffic, splits into a stream which is satisfied by the buffer, T_{buf}, and a stream which bypasses the buffer and is directly fed from main memory, T_{bypass}.

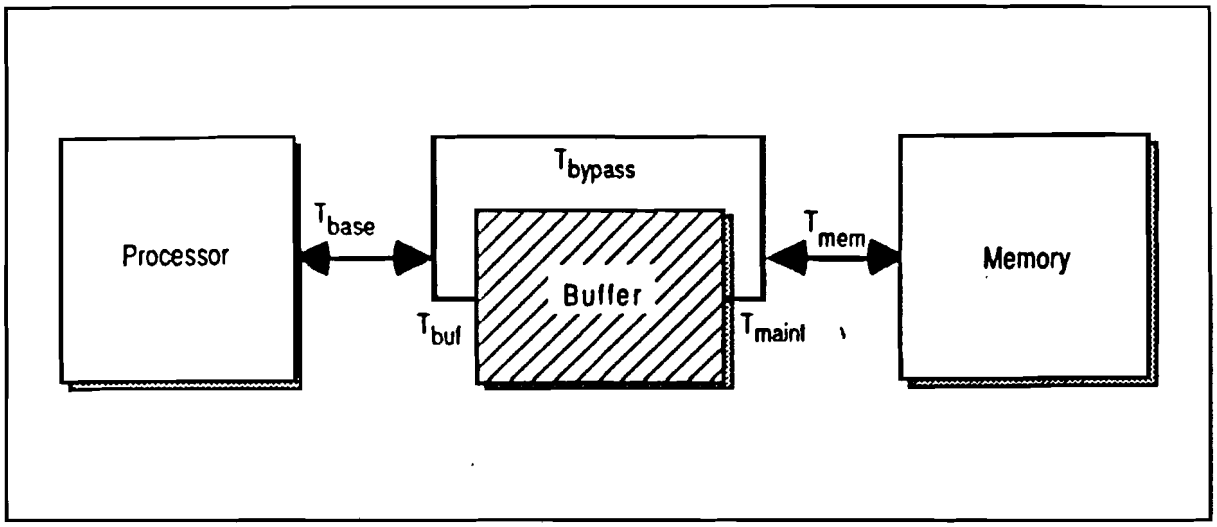

Figure 1.5: Buffer performance model.

T_{mem}, the memory traffic after buffering, consists of T_{bypass} together with the traffic needed to maintain the buffer, T_{maint}.

From the processor's point of view one is interested in the amount of delay the buffer and memory system impose on it. In order to be able to estimate this delay one needs the access times, including the miss penalty, and the miss ratio.
From the memory systems point of view the bandwidth requirement, bytes per second, imposed on it by the buffer and processor is important. This bandwidth requirement can be calculated from the expected processor clock frequency, the average amount of processor references per clock cycle, and the hit ratio.
The parameters which is provided by trace simulation in the Architect's Workbench is the miss (hit) ratio. Except for the average reference per clock cycle all other parameters can be considered external data. The average reference per clock cycle can at present not be calculated because the workbench traces do not contain timing information.

There are four basic parameters which characterize a reference in the data stream between processor and memory, the data type, the memory address (or logical

segment), the direction (read or write), and the time passed since the former reference. The data type is needed to define the amount of memory needed to store the data, the address to identify a piece of data (is it already in the buffer?), the direction to define which action the buffer should take, and the time passed to be able to investigate timing effects. Timing effects are especially important when a buffer tries to match different processor and memory timing and delays optimally. Momentarily the workbench traces contain all of these parameters except for the timing information.

1.4 Conclusions

A realistic simulation of a buffer mechanism in software requires benchmarks (realistic program load) and a way to implement instruction set architectures, because these architectures strongly influence the load which is actually presented to the buffer. The extended CARA methodology presents a way to create these realistic loads, simulating instruction set architectures, and thus to examine the performance of buffers mechanisms.

Because the trace files resulting from such simulations can be very large, they are split up into a basic block description file and a program flow registering file.

Through for instance the effectiveness of register allocation, compile time programs can strongly influence buffer performance measures.

Two of the more important buffer measures, the traffic ratio and the miss ratio are provided by the Architect's Workbench.

Although there is a demand for timing information to be included in the simulations, the Architect's Workbench does not yet include this timing information in it's traces.

Chapter 2: THE ARCHITECT'S WORKBENCH

The Architect's Workbench has been built according to the extended CARA methodology. This methodology can basically be seen as divided into two stages. Executing the first stage, representing the compile time software, results in the creation of a static trace file, the basic block description file. The second stage, representing the run time software, starts with creating a dynamic trace file, the program flow description file, and contains, as will be described in paragraph 2.2, the actual simulators of either separate data or instruction streams or the combination of both.

2.1 The compile time software

Because various high level languages tend to support different program characteristics, there should be a way to write benchmarks in several of the most important of these languages. In order to ease the processing of benchmarks written in different high level languages, an intermediate language, U-code, has been chosen. The use of this language results in a situation in which several front ends translate a variety of high level languages such as C, pascal, fortran, and in the future maybe others, to U-code. In this way the rest of the first stage, such as register allocation and compilation, becomes independent of a change of benchmark language, thus reducing the size of and increasing the flexibility of the Architect's Workbench.
A scheme of the compile time software organization is given in figure 2.1.

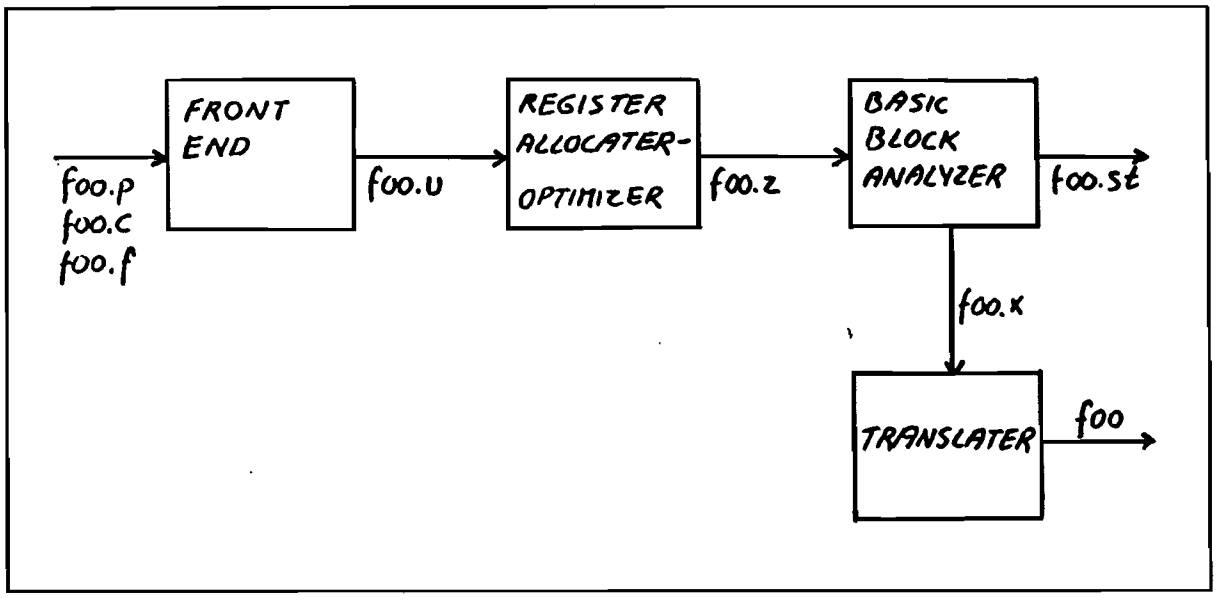

Figure 2.1: Compile time software organization.

One of the functions performed by a compiler (the compile time software) is the assignment of data to registers. The way in which this is done affects the execution characteristics of the program. A good register assignment scheme can, for example, reduce the number of movements of data items between main storage and registers, and allow use of the faster register-to-register type of instructions.

In some allocators register allocation takes place in parallel with code generation, because of the argument that the code generator has full knowledge of the target machine, e.g. restricted instruction formats or special purpose registers can be incorporated into the allocation strategy. A second approach splits the allocation into two steps. The first step is independent of the target machine and involves allocation of the general purpose registers and code optimization in an intermediate language such as U-code. The second step maps the optimized code onto the target machine, compiling down to the target machine assembler language.
This two step approach is also used in the Architect's Workbench, although for the reason of simulation this second step is split up into two parallel actions. The first action stores the optimized code in a static trace file, to be used for simulation of the target machine later on, while the second action assembles down to a host machine

(instead of the target machine) to create a trace file containing program flow information.
Within the Architect's Workbench thus only the general purpose registers are considered for allocation.

The occurrence of a constant or data name in an executable statement can be characterized by its effect on the associated data item during statement execution. A data item is *defined* when statement execution causes a new value to become associated with the data item. A data item is *used* when the current value of the data item is required for correct statement execution. Each data item may thus be appointed its definition-use-chain, defining its temporal validity. Interference occurs when two variables are both valid at some point within the program. Total interference occurs when every time a variable is used, it interferes with the same variable. Using DU-chains, register allocation is reduced to a graph coloring problem. Weighing functions are used to decide which data should have a high allocation priority.
The Architect's Workbench's register allocators make use of these DU-chains.

Before continuing something should be said about basic blocks. The basic block and the region are defined as structural units of a program.
When considering a program as a list of statements (with each statement an ordered sequence of delimiters, operators, constants and identifiers) a basic block is an ordered subset of the statements in this program defined as:

$$
\begin{aligned}
B_i &= \{ \quad s_a, s_{a+1}, \ldots, s_b \ \} \\
&= \{ \quad s_i \mid s_i \in P, \\
&\qquad s_i \text{ is executed before } s_{i+1}, \\
&\qquad s_i =! \ s_a \text{ may not be branched to from } s \notin P, \\
&\qquad s_i =! \ s_b \text{ may not branch to } s \notin P \ \}
\end{aligned}
$$

in which:

- P is the whole program
- B is a basic block
- s is a statement

A region can now be described as a collection of basic blocks, behaving as one basic block, provided one or more internal loops are ignored. Thus a region may vary from a basic block up to a whole program, as long as it contains one single exit point. Allocation within a block is considered local, while allocation within a region is considered global. Within the Architect's Workbench procedures are treated as regions. Thus a local assignment is assignment within a basic block, while global assignment is assignment within a procedure.

Local assignment is attractive because a compiler can easily, and in a reasonable amount of time, partition a program into basic blocks and derive the necessary interference characteristics. Local assignment however cannot usually maintain assignment history when crossing block boundaries, thus forcing all variables to be moved to main memory. This is especially inefficient when considering loops past block boundaries.

One-one register assignment allocates exactly one variable to each register in the region of assignment. The register allocator 'order' thus allocates variables one procedure (global, as will be explained on the following page) at a time. And, because it performs no data flow analysis, it maps local variables in registers for the entire time a procedure is active. Apart from the fact that only a certain amount of variables can be allocated to registers, this method has one major drawback: all registers are saved across procedure calls, because the callee might either need the registers, or some of the variables as non local variables.

This method however is still interesting because it is used in commercially available compilers. It is also used as a standard for measuring the effectiveness of other allocation strategies.

As opposed to most commercially available compilers, the following assignment strategies attach a greater importance to the creation of an optimally executable program than to the time involved in the compilation.

Many-one assignment allocates at least one variable to a single register in the region of assignment. Variables which do not interfere can be allocated to the same register. Assignment of a variable to a register set works as follows. Take a register from the set and assign the optimal combination of variables to it. Take the next register and optimally assign as much of the remaining variables as possible. Continue until the whole register set is allocated. Each variable that has been assigned a register uses that single register, possibly together with some other variable.

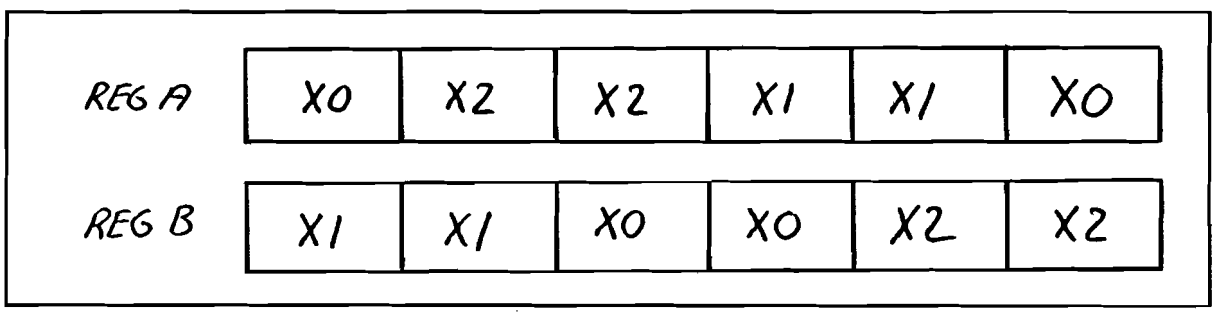

Figure 2.2: Many-few-assignment. (Example: 3-2-assignment)

Many-few assignment allocates at least one data item with each of several registers in the region of assignment. Several variables X0, X1, X2 are assigned to several registers, REG A, REG B, as shown in figure 2.2.

A basic block generator, 'hllref', extracts the information necessary to describe each basic block and puts it in a file foo.st. Furthermore, in the U-code file, it includes all calls to trace generation routines in order to be able to generate a trace during run time. Next a script file called 'utrans' calls a compiler which translates the U-code to machine-code for the host machine on which the Architect's Workbench runs. In case the host is an Apollo, 'u2a' will be called by Utrans. Next the Apollo assembler 'as' will compile the machine code file into a binary file, and the Apollo binder will link

'ld' a library containing trace routines and other standard routines to this file. The benchmark is now ready to be executed.

2.2 Run time software

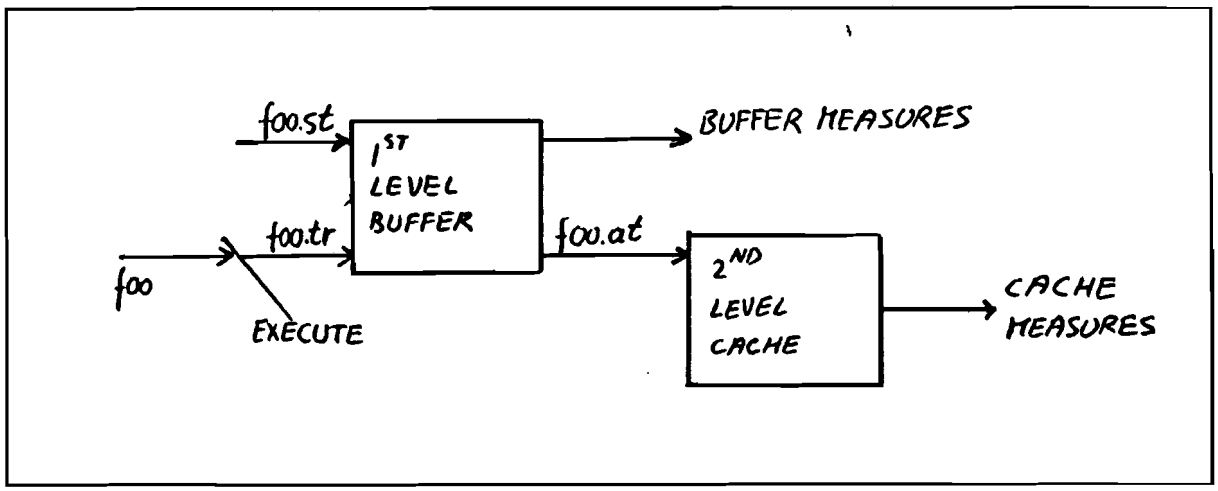

Figure 2.3: Run time software organization for a data traffic simulation.

After the first stage of the CARA methodology (compilation of the benchmark), the second stage (simulation) follows. This second stage starts with the execution of the created executable benchmark. The benchmark executes as it normally would have done, from the user point of view, but meanwhile, every time it executes a call to a trace routine, the actual basic block number is written to a trace file foo.tr, thus registering the program flow. In combination with the static trace file foo.st a buffer simulator can reconstruct the complete program flow and analyze all instruction fetches or data references and simulate the working of a buffer mechanism to provide buffer performance measures.

The Architect's Workbench distinguishes three kinds of simulations. The first kind, the data traffic simulation, is shown in figure 2.3 as a two level buffer simulation. The first level simulates program execution and buffer performance. Furthermore it generates a new trace foo.at which contains the address sequence of all references to

main memory that could not be covered by the first level buffer. The second level buffer simulator uses this address trace to extend the simulation to a cache.

In multi-level buffering first level (register oriented) and second level (cache) buffers have distinct purposes. The first level buffer determines the peak performance of the processor, while the second level buffer determines the actual performance. Hence, a cache does not hide architectural (first level buffer) flaws, but emphasizes them, because it reduces the influence of the memory system on performance.
In fact the Architect's Workbench allows up to three buffer levels to be simulated. The first level buffer may be a single or multiple register set, or a stack buffer. The second level may either not exist, or be a fully associative cache as implemented in the buffer simulators, or be an address trace foo.at for a separate simulator. In case off a fully associative cache the third level exists of one of the other three choices. The appropriate choice should be set in 'cache.conf', the cache configuration file, as specified in appendix 3.

The second kind of simulation, instruction traffic simulation, is implemented as a one level cache simulator. For the third kind, the mixed traffic simulation, the address trace from the first level data buffer is combined with the instruction reference trace to generate an extended address trace, which is then fed to the second level cache simulator.

2.3 The Architect's Workbench

As the previous paragraphs indicate, running a simulation on the Architect's Workbench without using a guiding simulator can be quite complicated. Figure 2.4 on the next page gives an impression of what simulation paths are possible, which simulators may be used, and what data files do occur along the way. Furthermore, in the following sub-paragraphs some more detailed examples are given about the three main simulation paths, for data, instruction and mixed traffic simulations.

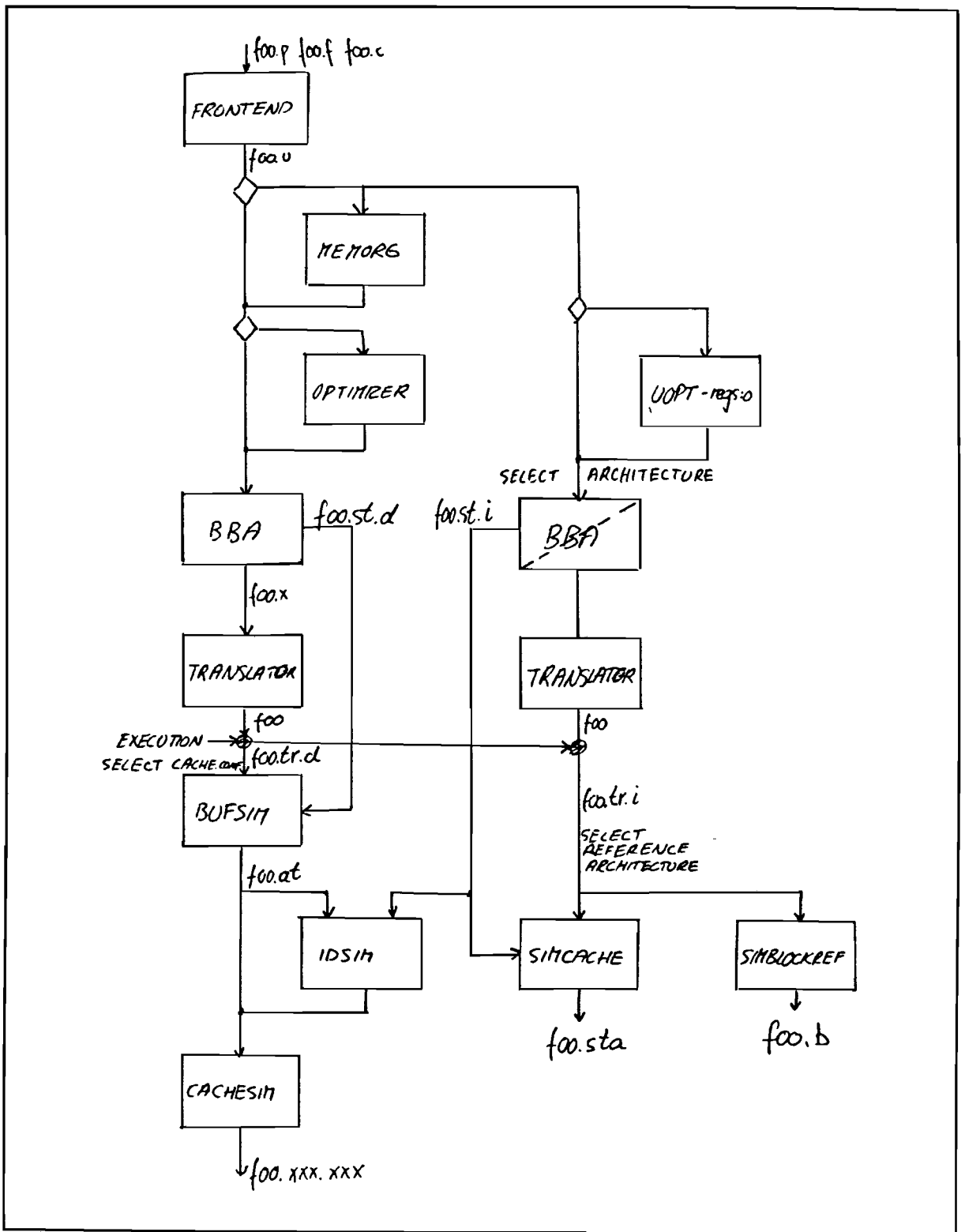

Figure 2.4: Layout of the Architect's Workbench.

Several appendices contain further data as to where to find and change parameters of caches and instruction set architectures.

'front ends' : Uccawb, ufortawb, upasawb, upas17awb
'optimizers' : Uoptawb, order, inter
'translators' : U2a, u2sun, u2vax
'buffer sim.' : Srssim, srssub, mrssim, mrssub, bufsim, bufsub
'cache sim.' : Asscch, setcch, setsub, setrep

2.3.1 Instruction traffic simulation path

Running an instruction traffic simulation with the Architect's Workbench is primarily done according to the following steps:

1 Generate a U-code file using a front end.
2 If desired perform optimizing with 'uoptawb' without register allocation, using the switch '-regs:0'. The static analyzers perform there own allocation, as far as they use registers.
3a Run the basic block generator 'hllref' to create an instrumented U-code file.
3b From the file awb.parameter select an instruction set architecture. Run the corresponding static analyzer ('dcastat', 'regstat', 'stackstat', 'genrstat') using the accompanying flags. Use the switch '-u' when 'uoptawb' has been used. The basic block description file foo.st is now created to replace the one created by 'hllref'.
4 Use a translator to create an assembler file, use assemble it, and link the corresponding library to it. Now there is an executable benchmark.
5 Execute the benchmark to create a trace file foo.tr.
6 Take from a previous cache simulation the number of bits read and put it in a file foo.mt. Remove foo.mt if no referencing simulation is intended. Run 'simcache' or 'simsmcache' to simulate an instruction cache. Results appear in

foo.sta. If a file foo.mt exists then this is assumed to be a referencing simulation. Relative traffic will be calculated using the number of bits read by both caches.

2.3.2 Data traffic simulation path

1 Generate a U-code file using a front end.
2 If desired perform optimizing.
3 Run the basic block generator 'hllref' to create the instrumented U-code file and the basic block description file foo.st.
4 Use a translator to create an assembler file, use assemble it, and link the corresponding library to it. Now there is an executable benchmark.
5 Execute the benchmark to create a trace file foo.tr.
6 Set a link named 'cache.conf' to a cache configuration file. This file specifies the buffer levels and cache parameters, as explained in appendix 3.
7 Run a first level buffer simulator, creating foo.at.
8 Run a second level cache simulator, using foo.at.

2.3.3 Mixed traffic simulation path

A mixed traffic simulation basically starts with generating a static instruction reference file foo.st. Furthermore a complete data traffic simulation is performed up to the generation of the address trace file foo.at. These two files are now merged by idsim. Merging is performed by reading two instruction references after every data reference, unless there are no instruction references left in the current basic block, or an immediate write occurs in the data stream. In case of an immediate write the remaining instruction references in the current basic block are all read at once. The newly created trace file foo.at is then fed to the second level cache simulator.

2.4 Conclusions

The Architect's Workbench provides a range of possibilities for the simulation of instruction set architectures and buffers, using various kinds of traffic.

While growing the Architect's Workbench has become increasingly hard to penetrate. This chapter contains a description of the structure of the workbench. Even so, due to the large amount of programs and the complicated ways they are interconnected, it is not easy to work with the Architect's Workbench.

Chapter 3: VIRGIN

Testing the Architect's Workbench still is a quite complicated task since it is a large and complex structure of programs. Getting acquainted with the workbench therefore starts with reading the Introductory User's Guide [10], which is a bit vague and incomplete in details, followed by studying the reports of researchers who have made use of the workbench in their architecture and buffer performance studies. After reading comes trying some simulations yourself. At this stage 'virgin' was written. Mainly as a way to try and write down the structure of the workbench, and the relations and connections between its contents, it developed into a simulator.

3.1 New features

In order to make it easier to use the Architect's Workbench, it has been equipped with various script files. These script files together work somewhat like a simulator, asking the user to specify choices and parameters, and relieving the user of most of the dumb work. The main simulation script files, 'isim' and 'virgil', however have been written for specific research done by earlier users. They allow limited access to the possibilities of the Architect's Workbench. For example, the data cache simulation in the Introductory Users Guide is impossible using the data traffic simulator 'virgil'.

'Virgin' makes the existing script files obsolete. It directly accesses all programs that are part of the workbench. The script is like a primitive simulator leading the user along his simulation path through the Architect's Workbench, based on choices he

makes on the way. 'Virgin' offers much more freedom than the old script files. New features of 'virgin' are:

- Specification of a user simulation directory instead of making use of the common benchmark directory.
- The possibility to add parameters, as specified in the manual pages, to the simulators.
- The ability to bypass code optimizers.
- A copy on screen of every command executed, in order to show the user what commands he should give to perform a similar simulation himself. This is a teaching function. Anyone doing research will not run complete simulations, but, once he has the basic files, execute some parts with various parameters. Reading 'virgil' shows him how to write his own script file.
- 'Virgin' performs in a clear and logical order all the functions that before were performed by the script files 'virgil', 'idsim', 'utrans', 'z2tr', 'lookup-iarch', 'askugen', 'show-cmd', 'askcache', 'u2tr', 'ugen', and 'obsolete'.

3.2 Using virgin

STEP 0: Initialization

In this step the Architect's Workbench header is printed on the screen.
A path is set to a user directory specified in 'virgin', where an additional Instruction Set Architecture parameters file 'more.parameters' may be kept. Furthermore paths are set to several workbench data files containing respectively instruction set architecture parameters, cache architecture parameters and benchmarks.
Also a host machine, Apollo or Sun, is chosen, because files assembled on them do not get the same file name extensions, while files translated on them need different translators. The operation of 'virgin' on the Sun has not yet been properly tested.

The error and warning messages are initialized, and a warning is given for any workbench program that is supposed to be present but can not be found.
It is possible on the command line of 'virgin' to specify the benchmark to be simulated and it's extension, but this feature has not been extended to other input yet.
Next the user has to specify a simulation directory, the directory in which all the intermediate and result files are going to appear. The user may enter 'dummy' if he wants his current working directory to be his simulation directory.

STEP 1: Selection of the type of simulation

The user now specifies whether he wants to do an instruction 'i', data 'd' or mixed 'id' traffic simulation. The mixed simulation will initially be treated as two separate instruction and data traffic simulations, after which the files actually needed will be joined together.
Next the user specifies the benchmark language, C, pascal or fortran, and in case of pascal whether it should be a 16 bit or the default 32 bit instruction set simulation.
'Virgin' then presents the available official benchmarks to choose from. The user may either specify one of those, or any other program already in the simulation directory, as long as it's extension is 'p', 'c' or 'f'. If the benchmark is found in the workbench's benchmark directory then it is copied to the user's simulation directory.
This causes a simulation error further on for the benchmarks 'pasm' and 'pcomp' because they also need data files to work with, for which copying is not provided yet.

STEP 2: U-code generation

The appropriate front end compiler is executed, before which options as specified in the workbench's user manual may be added.

STEP 3: Preparations for the instruction traffic simulation

This step is only executed in case the simulation is an instruction traffic simulation. The instruction set architecture parameters file and the additional user parameters file, as mentioned in the introduction step, are read, and from their combined contents the user may now select an architecture for simulation. From this choice, within this step, the appropriate static analyzer and the switches that go with it are **selected**, and within the simulation directory a subdirectory is created for the results of the instruction cache simulation to be stored in. This subdirectory is named after the instruction set architecture that will be simulated.

STEP 4: Register allocation

Since for a mixed simulation two simulation paths are followed in parallel, two versions of several files are going to exist at the same time. To those files, where necessary, an additional extension '.d' or '.i' will be added.
In this step a code optimizer annex register allocator (or none) is selected. There are two schemes for selecting an optimizer, one for a data buffer simulation, and another one for an instruction buffer simulation. The reason for this is that in case of a mixed simulation one may want to specify different optimizers for the data and instruction buffer simulation.
Before the optimization one can execute the memory reorganizer 'memorg' which does not work well with the optimizer 'uoptawb'. For an instruction traffic simulation however one has to use 'uoptawb' **without** register allocation (is taken care of automatically), or none, because the static analyzers 'xxxstat' do not take optimized code.
In case of a data traffic simulation from a menu a register allocation strategy is chosen. For some of the allocators the user is asked to specify certain basic parameters, like number and kind of registers for allocation by the global many-to-one register allocator 'uoptawb'. Finally one is asked whether one wants any additional switches to be set, whereafter the register allocator is executed.

STEP 5: Static and dynamic trace file generation

Because of the possibly parallel simulation the step 5 script may be executed twice.
First the trace library to be loaded with the benchmark is selected. Next the basic block allocator 'hllref' is executed to create an instrumented U-code file and a static trace file foo.st describing all basic blocks. The U-code file is then translated and the resulting assembler file is assembled by a host system assembler, after which the binary file is loaded with the selected library by a host system loader. The executable result is executed and creates a dynamic trace file foo.tr.
In case of an instruction traffic simulation the selected static analyzer is executed which removes the static trace file created by the register allocator 'hllref' and replaces it with a new one which is created using the parameters of the selected instruction set architecture.
The user may have added parameters for 'hllref' and the translator. The parameters for the static analyzers, in case of instruction traffic simulation, have been set by the selection of the instruction set architecture.

STEP 6: Initializing cache.conf

In case of a data traffic simulation, which exists of a first level buffer simulation and optionally a second level cache, the user chooses a data cache architecture from the ones presented by 'virgin'. In fact a link is created to the appropriate configuration file. The file is read by the first level buffer to know whether a second level fully associative cache should be simulated before the third (was second) level cache. The second (possibly third) level cache reads the file for cache parameters.

STEP 7: Simulate a data buffer or do an instruction cache simulation

In case of an instruction cache simulation 'simblockref' can be executed to create a file containing basic block reference counts. The user makes a choice between a simulator for caches (512b to 16Kb sizes) or small caches (128b to 4Kb sizes).

Next a reference instruction set architecture is selected, or 'self', in order to acquire the number of bits read in the referenced simulation. In fact the referenced cache architecture simulation results are referenced through the name of the instruction set architecture that that cache was simulated with. This construction exists because all simulation result files get the same name foo.sta, but are stored in a subdirectory named after the instruction set simulated.
In fact the number of bits read from the referenced cache is the only reference information used, in order to create a correction factor for the presentation of relative traffic figures.
In case of a data buffer simulation the first level buffer simulator, simulating register sets or a stack buffer, is selected and executed. An address trace foo.at is created, to be used by the second level buffer simulator. Options may be added for this buffer simulator.

STEP 8: Mixed simulation

The simulator 'idsim' combines the address trace generated by the first level buffer simulator with the static trace file created by the static instruction analyzer in step 5 to form a new address trace file to be exchanged with the old one.
Options may be added.

STEP 9: Data and mixed traffic cache simulation

A second level cache simulator is chosen, which uses the address trace file to simulate cache performance. Options may be added.

3.3 Conclusions

A new script file, 'virgin' has been written which makes all the existing script files obsolete. It offers a lot more freedom in selecting the simulation path and specifying parameters than the old script files, and can be used as a simulator or teacher.

Chapter 4: TESTING THE ARCHITECT'S WORKBENCH

Recently the Architect's Workbench, which has been developed on a Vax system, was transferred, via a Sun system, to the Apollo environment, because of the relative abundance of Apollos in our group. At the transfer the compilers, provided by the host system, changed naturally, and the U-code to host assembler code translator of the workbench was rewritten. Due to these changes the workbench ceased to function as it was supposed to. Except for the U-code to Apollo translator u2a most errors have to be searched in differences between Vax, Sun, Apollo and the Pascal and C compilers on those systems.

The first try at finding errors was made by running several kinds of simulations with the available benchmarks, in order to locate problem spots. Using all benchmarks is likely to cover most of the U-code statements and at least all of it used by the available benchmarks. These simulations resulted in some quite long error lists generated by the Apollo assembler. These happened to be related to u2a.

4.1 U2a

Rewriting u2a had been done through replacing all Sun assembler code in the old compiler by Apollo code. Furthermore some other problems caused by the different conventions of the host systems, like read-only code segments on the Apollo, and thus the treatment of externals, had to be compensated for.

Most of these errors could be described as typing errors when exchanging Sun for Apollo code. Some code statements had been overlooked. A full check of the listing of u2a showed some additional errors in compiler pieces for the compilation of U-code statements which had not been used. A known remaining error is:

```
add.l #     16,sp
Illegal symbol in source field.
```

The error resulted when compiling the benchmark pcomp. The error is located in u2a.p, compilation of UINST

Other errors occurred for unrecognized or undefined globals.
This was indirectly caused by the fact that the Apollo system considers the code segment as a read only area, during execution as well as during the loading phase. Labels that can not be resolved by the assembler thus can not be accessed by the loader. For external routines and variables, whose address is not known at compile time, this requires a special treatment. First a label has to be created in the data segment. This label can be resolved by the loader. For a call to an external routine, this label can also be loaded into a register. Executing 'jsr (register)' then actually calls the external routine. Conform the Apollo convention the register A0 is used. However, the name of the external label, in the data segment, should be different from the name of the variable that is used in the code segment. The variable and the label thus have to be coupled in the data space also.

For u2a the easiest solution requires the creation of include lists (to be included in the benchmark under compilation) which contain a piece of data space for all external addresses that can possibly be referenced. Then the only thing u2a has to do is adapt the names of the U-code routines that are recognized as externals, giving them an extension. Lists have to be available to u2a from which u2a can decide whether a called routine is an external or internal routine.

An unrecognized global occurs when u2a does not recognize a call as a call to an external (global) routine, and it gets the 'internal' status, thus trying to access a non existing internal routine. An undefined global occurs when a call gets the 'external' status, but his patch is not in the include list.

4.2 The u2a AWBpascal library

In fact there are some more lists and include files. The way in which they are related will be explained using figure 4.1. When a benchmark is compiled to assembler code, math, io and other 'standard' routines may be referenced. These routines are supplied by the compiler in an include library which, after compilation, is linked to the benchmark by the loader. In case of an AWBpascal (workbench pascal) benchmark this will be an AWBpascal library (not 'standard' Apollo pascal or any). This AWBpascal library therefore is also compiled using upasawb and u2a.
Some 'basic' and (in case of a trace library, trace) routines are implemented in a C library, which on its turn is linked to the AWBpascal library.

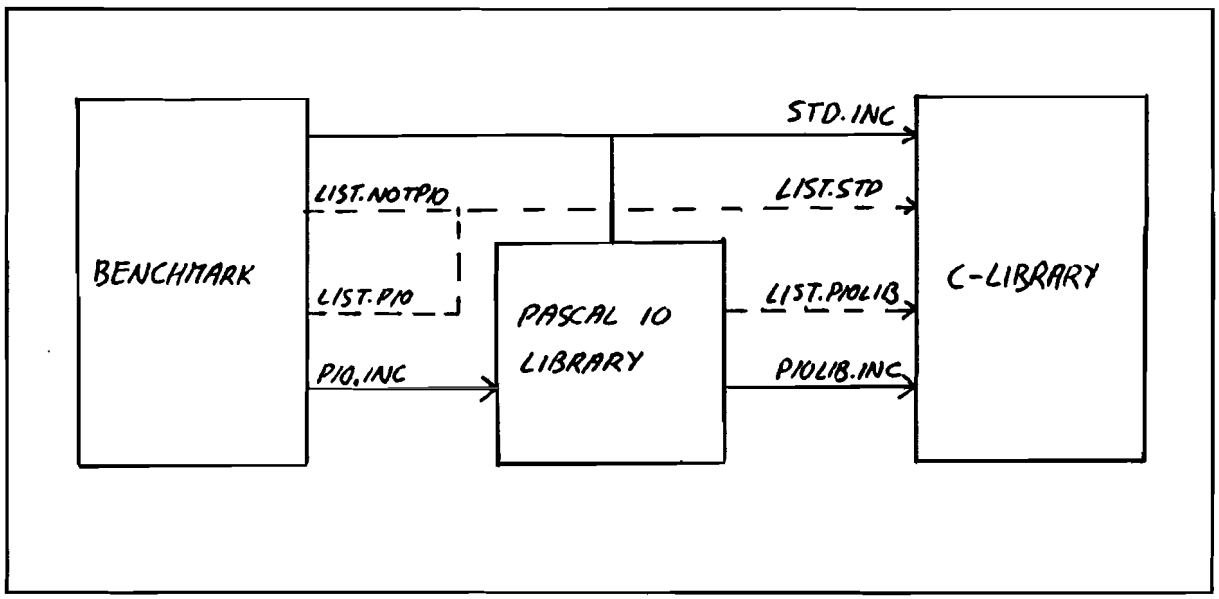

Figure 4.1: The workbench pascal library structure.

Basically there is the following: a set of C routines which is called (external, with respect to AWBpascal) by the AWBpascal include library, and is linked to it after the AWBpascal library is compiled. These C routines are mentioned in list.piolib, the checklist of library externals for u2a, while their declaration in the data space has been taken care off in piolib.inc, which is included in the AWBpascal library at compilation of the library by u2a.

Next there is list.pio, which mentions all routines the benchmark may call from, or via, the AWBpascal library, as well as an include file pio.inc which is included in the benchmark during compilation of the benchmark by u2a. List.pio is not used directly. There also is, hand made, list.notpio, which mentions all routines (trace, math, and others) that are not in list.pio.
The script file xrf_pio creates list.pio and pio.inc through scanning pio.p and tracepio.p, the AWBpascal libraries, for procedure names. Therefore the declaration of the external C routines called by the pascal library happens in an include file pio.ext in the library, as to not to get their names in list.pio, unless they are explicitly referenced via the library. This happens for instance with the open routine, which may be referenced by both the AWBpascal library as well as the benchmark.

Finally the include file std.inc is included in both the include library and the benchmark. This file provides general system routines and calls for the C routines which are not part of the set called by the AWBpascal library, like math. The file list.std is created by taking list.pio and adding the math and system functions, list.notpio.

Thus:

- In case of a library, std.inc and piolib.inc are included by the assembler.
  List.piolib is scanned by u2a for routines with special treatment.
- In case of a benchmark std.inc and pio.inc are included by the assembler.
  List.std is scanned by u2a for routines with special treatment.

The largest problem in debugging benchmark execution and the included libraries is caused by the fact that there is no debugging information available for any standard debugger. You can not step through, you may just watch results appear during execution. The only way around it is to include your own comments and debugging routines and re-compile, which takes a lot of time.

Within the C library _dprint(number) and _fdbprint(FDB) are included which allow respectively printing of an integer or (most of) the file description block to the error stream stderr, which writes to screen immediately. In the AWBpascal library, in pio.ext, these are declared as $Dprint(number) and $Fdbprint(FDB), which just call the C routines. These debugging routines can thus be called from any place in any of these libraries without further declaration, provided you choose the right procedure name. If calls to the AWBpascal declarations are entered in list.std and std.inc then they may even be accessed by the benchmark itself.

The file description record (structure) is defined in the AWBpascal library as well as in the C library and is passed on most procedure calls. Care should be taken that these declarations stay compatible. They should contain the same set of variables in the same order, using the same number of bytes. For instance, because C by default assumes integers to be 32 bit, in pascal the standard 16 bit integers have to be redefined as 32 bit integers. Problems occurred because the two AWBpascal libraries used different declarations, while both used the same C library.

Because for both AWBpascal libraries the same include lists list.pio and pio.inc are used, and the libraries themselves are not equal, the linker stage causes some undefined globals errors which are harmless. The routines involved are in case of an instruction traffic simulation using plib.a: _$$XXX_ex, with XXX = get, put, rdc, rdi, rln. In case of traceplib.a these are rdb, rdenum, rdident, rdset, strset, strwrite, wrb, wrenum, wrset. In the latter case however they are mainly not implemented yet, although not used by the current benchmarks either.

Within the C library part a bug exists which can cause a segmentation fault upon the closing of data files by benchmarks which make use of such files. For the time being a warning message will appear upon calling the routine _close.c.

In the u2a compiler currently only the creation and inclusion of the pascal libraries (trace)plib.a is implemented.

4.3 The instruction traffic simulators

'/awb/icsims' is the directory in which the sources of the instruction traffic simulators reside. The instruction cache simulators simcache and simsmcache had been ported from pascal to C. There were some problems in calling trace accessing routines because some routine calls in the ported simulators passed variables, as it is normal to do pascal, while in the associated C library pointers to variables were expected.

The simulations of instruction caches have been reproduced as in the Introductory User's Guide, except for some details. For a cache which is larger then the amount of memory needed to store the whole benchmark the amount of lines fetched is equal to the amount of distinct lines referenced and the total amount of lines in memory. After correcting the offset of the reference counter with 1 this was true. Also since the production of the Introductory User's Guide someone has fixed the counting of distinct lines referenced, and disabled the creation of the last line in the data file. This line used to mention the architecture simulated and the one referenced. This cannot be fixed in a simple way since the simulator does not even get that information any more.

4.4 The data traffic simulators

After getting u2a in a workable condition it was tried to reproduce the simulations presented in the Introductory User's Guide, in order to compare the results produced by the simulators with the results as they ought to be. The benchmark myfile.p, and an exact description of the simulations and the results are included in appendix 2.

The directory '/awb/datasys' contains the sources from which the first level data traffic buffer and the second level cache simulators are built, as well as those for the basic block allocator hllref. The simulators are created as enlightened in figure 4.2.

The most complicated one is the buffer simulator. The main buffer part simsim.p contains those routines which are common to all buffer simulators, such as reading the static trace file, reading the dynamic trace file, initializing the output file, error routines, and common initialization. Some of these routines, error and printing procedures are thus used as global routines. It is however not allowed to create global routines in the module which contains the main program block. Therefore simmain.p is declared as main program, and it calls simsim.p.

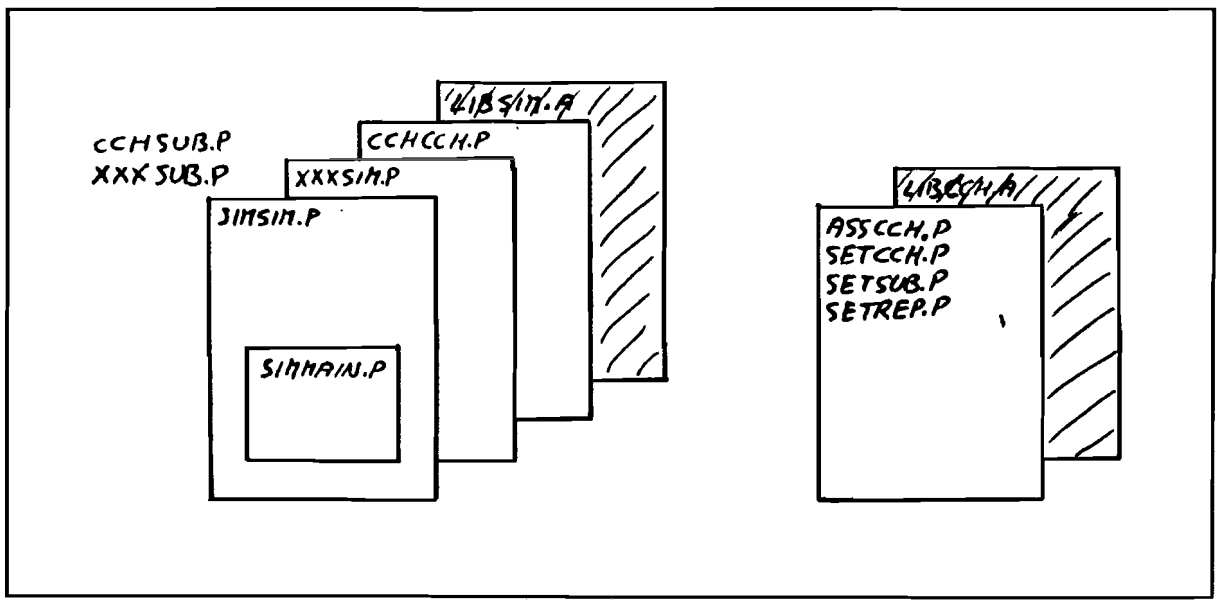

Figure 4.2: Buffer simulator buildup.

Simsim.p also calls the actual buffer simulation routines, as well as initialization of specific variables. These are located in XXXsim.p, after which the final simulators are named, for the appropriate buffer simulation (XXX = Single Register Set, Multiple Register Set, and stack BUFfer).
XXXsim.p then calls cchcch.p, which generates the address trace file, and allows the user to simulate an additional (second level) fully associative cache.
If this cache is using subblocks, then these modules change to XXXsub.p and cchsub.p.
The library libsim.a which gets included with the rest supplies C routines for the handling of references to the dynamic and address trace files.

The cache simulators just consist of one program linked with a C library libcch.a for the trace routines.

4.4.1 First level buffer testing

When testing the data traffic simulators the first problem that occurred in accessing the trace file foo.tr. While the simulators happened to be written in Pascal, the trace handling routines were written in C. Pascal passes variables by value while C passes them by reference. Some violations on this were found.

Furthermore the Domain Pascal manual, as well as the Domain C manual state that integer values and character strings are passed correctly when using 'val_param' in combination with 'external', but they also explicitly state that no guarantee whatsoever is given for something more exotic as enumerated data types or records. Therefore a conversion file conv.p has been loaded with the C routine library which handles variable passing by explicitly converting enumerated types in Pascal to integers in C, and vice versa.

The next error found involves several variables which were declared throughout the pascal sources, and were also declared and referenced in the C library. Because they were nowhere defined external or passed on routine calls, they did not get linked through, thus producing a junk address trace. They have now been defined in simsim.p and are referenced as externals anywhere else. The variables involved are:

- blockcount
- blockmask, blockshft, subblockshft
- cachein, cacheout, cachebs, cachebn

The last error involves the counting of references made. In simsim.p in the procedure scantrace the amount of references is counted in hllrefcount. It is checked against a continue flag and hllrefmax. Hllrefmax gets defined as maxint. On the Vax this used to be for a 32 bit integer. The Apollo however assumes integers to be 16 bit unless defined otherwise. This had been solved through redefining integer as integer32 in the file local.inc, which gets included in any pascal program. Maxint however was still defined for 16 bit, which is much to small for most benchmarks. This is now also corrected in local.inc.

4.4.2 Second level buffer testing

The second level buffer has not been tested intensively. The test files produced similar results as in the Introductory User's Guide. The declaration of the C routines has been checked and there is no special reason to suspect any problems.

4.5 The combined traffic simulator

The combined traffic simulator essentially is built of a trace merger followed by the standard second level buffer. No problems have been found.

4.6 Conclusions

Several errors created by the change of host system and compilers have been found. They have been corrected throughout the workbench. Several parts of the workbench have been tested and found correct. Although not all parts of the Architect's Workbench have been tested in detail, it is expected that all code that has not been changed functions properly now.

Although a lot of them have already been corrected, errors may still be found in the U-code to Apollo translator u2a. Currently a trainee is working on their detection.

U2a does not yet support libraries for C and fortran benchmarks, nor have their front ends been checked.

At least one error does still exist in that part of the C library that is concerned with basic file handling for the AWBpascal library. It is located in the close routine.

A test of the Architect's Workbench in great detail can best be performed by running several strategic simulations on a properly working version of the workbench elsewhere, and comparing the results of those simulations, including all intermediate files, with the same simulations on the Apollo. This should also yield the last remaining errors.

Chapter 5: ADAPTING FOR THE C-PROCESSOR

Within the C-processor project the demand has arisen to develop a suitable buffering mechanism, possibly using the Architect's Workbench. For the developers it would be desirable if the simulations needed would not only yield rough capacity measurements, but also realistic timing measurements. This calls for a way to include timing information in the Architect's Workbench.

5.1 Simulating the stack cache

The C-processor will be a stack based machine and will use a stack cache to hold local variables and pass parameters to procedures and functions. Besides this stack cache the processor will be equipped with an instruction cache and (probably) a global data cache. Local data to the stack cache may either be the part of stack that fits in the stack cache, or all data on stack, or an intermediate definition. Since it is not yet decided that a global data cache will be present, the stack cache most likely to be implemented will consider all data on stack local, in order to give better results. The locality of stack data is highly compiler dependent. Therefore simulations are needed to decide which definition gives the best performance. Cache performance can be evaluated by feeding a cache model in software with a series of references. These traces can be obtained by running a workload on a software model of the processor. This is the method used by the Architect's Workbench.

The service time of the cache, the number of cycles needed to run a trace, can be calculated from the amount of references that have produced hits and misses. These

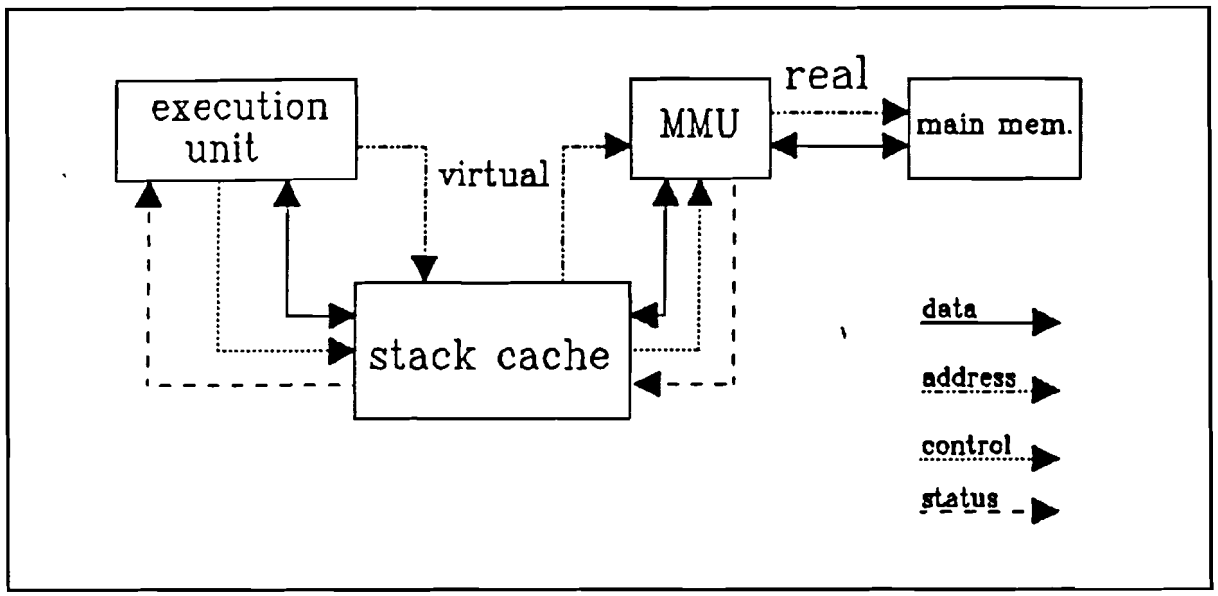

Figure 5.1: The stack cache in the C-processor.

reference counts are available in each cache simulation output file. Also the miss rate and traffic ratio are provided. A more accurate simulation can be performed if the time between references is known.

Using the traces foo.st and foo.tr, as implemented in the Architect's Workbench, timing information should be included at a spot at which the instructions of the target architecture (or U-code) are still available. This is certainly not at the point at which the dynamic trace file foo.tr is created. This file only registers entrances to basic blocks and the addresses that are referenced within those basic blocks at run time. At the creation of the static trace file foo.st however the U-code is available, from which an estimate of the instruction sequence of the simulated architecture can be made, and the time passed between references to memory can be calculated. In the final stack cache simulator the timing information from the static trace file can then be combined with the addresses stored in the dynamic trace file tp generate a realistic load for cache simulations.

The way in which the Architect's Workbench should be adapted to support timing information to simulate the C processor stack cache may be summarized as follows:

- A table should be created with estimated execution times (in clock cycles) for each U-code instruction.
- An adapted version of the basic block generator hllref should, while creating foo.st, read this table and calculate the number of clock cycles passed between memory references.
- A stack cache simulator should be written (possibly through adapting the first level stack buffer simulator bufsub) to simulate the stack cache.

5.2 Adapting hllref

Adapting hllref to support timing information (clock cycles) can be done in the following way:

First there should be a list containing the estimated execution times for all U-code instructions. U-code instructions embodying comments and other trace management should get an execution time of 0. The list can best be read from a file, to a table containing two positions for each U-code instruction, one for the U-code instruction name and one for it's execution time. (The header file ucode.h contains a list of all allowed U-code instructions).

When hllref reads the U-code file calling the procedure ucodefilescan, for every instruction read the procedure ucodeinst is called to analyze that instruction. Just before that point, for instance through a procedure checktable, the current instruction's execution time should be added to a timer. In ucodeinst, if a instruction inhabits a reference, the procedure addref is called to add a reference to the list hllrefs. At that point the timer should be saved with the reference and reset. Then every reference contains the time passed since the last reference.

Also there are several instructions which may trigger the variable newblock to become true. Back in ucodefilescan upon true the procedure genblock is called to

write the simulated block to the static trace file foo.st. This happens through the procedures putreflist and putref, which empty the list hllrefs. The procedure putref should save the timing info with the reference. Every reference is encoded as a string of characters to which the time can easily be appended. (The static file is read by the procedure gethllref, appendix 5, in the buffer simulators. This procedure should, when decoding the string, also read the timing info).
An extra timer should be added to the last reference of each block in order to save the time between the last reference and the end of the block. This can not be an extra reference since simulators count references.

There is however a problem in that no information is available from the io library. Because the library is also analyzed by hllref this info is available in the static trace, and could somehow be inserted in the dynamic trace. This quite complicates the matter.
Another solution would be to put this timing info in a table also to be read by hllref. Hllref does check on library routines, so here the info could be inserted in the benchmark's trace.

Furthermore in this way there will be no timing information on the included C library.

5.3 Conclusions

In this chapter a way is presented in which the Architect's Workbench can include timing info in it's static trace. A description has been given of the adaptations that have to be made to hllref.
There remains however a problem with the libraries to be solved.

Chapter 6: CONCLUSIONS

A realistic simulation of a buffer mechanism in software requires benchmarks (realistic program load) and a way to implement instruction set architectures, because these architectures strongly influence the load which is actually presented to the buffer. The CARA methodology presents a way to create these realistic loads, and thus to examine the performance of instruction set architectures and buffers.
The most important buffer measures can be derived from the traffic ratio and the miss ratio. These measures are provided by the Architect's Workbench. Although there is a demand for timing information to be included in the simulations, the Architect's Workbench does not yet include this timing information in it's traces.

A new script file, 'virgin' has been written which makes all the existing script files obsolete. It offers a lot more freedom in selecting the simulation path and specifying parameters than the old script files, and can be used as a simulator or teacher.

Several errors created by the change of host system and compilers have been found. They have been corrected throughout the workbench. Several parts of the workbench have been tested and found correct. Although not all parts of the Architect's Workbench have been tested in detail, it is expected that all code that has not been changed functions properly now. Errors may still be found in the U-code to Apollo translator u2a. U2a does not yet support libraries for C and fortran benchmarks, nor have their front ends been checked. At least one error does still exist in that part of the AWBpascal library that is concerned with file handling.

A way is presented in which the C-processor stack cache can be simulated within the Architect's Workbench, through the inclusion of timimg information in the static trace file. Changes to be made to hllref have been documented. Afterwards, the rest of the workbench might be equipped with timing information also.

LITERATURE

[1] Day, W.H.E.
Compiler assignment of data items to registers.
IBM Systems Journal, No. 4 (1970), pp. 281-316.

[2] Freiburghouse, R.A.
Register allocation via usage counts.
Communications of the ACM, Vol. 17 (1974), pp. 638-642.

[3] Huck, J.C. and M.J. Flynn.
Analyzing computer architectures.
IEEE Computer Society Press, nr. 857, 1989.

[4] Kelly-Bootle, S. and B. Fowler.
68000, 68010 and 68020 primer.
Howard W. Sams & Co. Inc., Indianapolis 1985.

[5] Mitchell, C.L.
Processor architecture and cache performance.
Stanford University, Technical report: CSL-TR-86-296, July 1986.

[6] Mitchell, C.L. and M.J. Flynn.
A workbench for computer architects.
IEEE Design & Test of Computers, February 1988, pp. 19-29.

[7] Mulder, J.M.
Tradeoffs in processor-architecture and data-buffer design.
Stanford University, Technical report: CSL-TR-87-345, December 1987.

[8] Paalman, J.A.H.
A stack cache for the C-processor.
Eindhoven University of Technology, Graduation report, August 1990.

[9] Smith, J.E. and J.R. Goodman.
Instruction cache replacement policies and organizations.
IEEE Transactions on Computers, Vol. C-34 (1985), pp. 234-241.

[10] Torellas, J. et al.
Introductory user's guide to the Architect's Workbench tools.
Stanford University, Technical report: CSL-TR-88-355, May 1988.

[11] Withagen, W.J.
Porting the AWB to the Apollo environment, "Hacking Away".
Eindhoven University of Technology, in preparation.

[12] Unix Programmer's Manual
A Unix based help utility within the Architect's Workbench supplying information about workbench programs.

[13] Various Unix and Apollo manuals.

APPENDIX 1: List of extensions used by the Architect's Workbench

The following list gives in a compressed way information about all known file extensions that may occur. It is the purpose of this list to make it easier to work with or at the Architect's Workbench.

Extension	Produced by > Consumed by <	Remarks
'none'	> ld	Executable benchmark
a,asm	> 'translators'	Assembler file
	< as	
at	> 'buffer sim.'	Address trace file
	> idsim	
	< 'cache sim.'	
as.cch	> asscch	Results of cache simulations, for awbplot
	< USER	
b	> 'buffer sim.'	Dynamic basic block count
	< order	
bin	> as	Binary file
	< ld	
c	< uccawb	C-code source file for C-to-U-compiler
dif	> order	Performance statistics
	< USER	
f	< ufortawb	Fortran-code source file for Fortran-to-U-compiler
iz	> memorg	Instrumented U-code from pre-organizer

	< 'optimizers'	
l	> 'front ends'	Compilers list file
	> uoptawb	Optimizer action summary
	< USER	
mt	> copy with 'awk'	Contains masterbits: number of bits read in referenced instruction cache simulation
	< sim(sm)cache	
p	< upasawb	Pascal-code source file for Pascal-to-U-compiler
s	> upasawb	Symbol table info from Pascal compiler
	< USER	
set.{xxx}	> set{xxx}	Results of data cache simulations for awbplot
	< USER	
st.d	> hllref	Static trace file containing basic block information
	< 'buffer sim.'	
st.i	> {xxx}stat	Static trace file containing basic block information
	< sim(sm)cache, idsim	
sta	> sim(sm)cache	Instruction cache simulation results
	< USER	
stai.s	> simsmcache	Results of instruction cache simulations for awbplot
	< USER	
stai.r	> simcache	Results of instruction cache simulations for awbplot
	< USER	
t	> uoptawb	Strings that occur in the U-code program
	< USER	
tr.*	> ex. benchmark	Trace file containing info about program flow
tr.d	< 'buffer sim'	
tr.i	< sim(sm)cache, simblockref	
u	> 'front ends'	Original U-code
	< memorg, 'optimizers'	
w	> stackref	Instrumented U-code
x	> hllref	Instrumented U-code

```
                < 'translators'
z               > uoptawb,order     Instrumented U-code
                < hllref
```

'front ends' : Uccawb, ufortawb, upasawb, upas17awb
'optimizers' : Uoptawb, order, inter
'translators' : U2a, u2sun, u2vax
'buffer sim.' : Srssim, srssub, mrssim, mrssub, bufsim, bufsub
'cache sim.' : Asscch, setcch, setsub, setrep

st.d, st.i, tr.d and tr.i are used in 'virgin' instead of st, sti, tr, and tr to clearly indicate which files belong to which simulation type.

APPENDIX 2: Virgin

The following pages contain a listing of the script file Virgin.
Virgin is a simulator which has been written to ease working with and at the Architect's Workbench.

```
#!/bin/csh -f
##############################################################################
# File: virgin
#
# This script file is (will be) a desktop for the Architect's Workbench.
# It takes care of shaping the flow of commands necessary to run a simulation.
#
# 02/10/90 Created - RGB
#
# Usage:
#    virgin [ <bench>.{p,c,f} ]
#
#    <bench> :: The name of a benchmark. It must be in the correct
#               directory, depending upon the filetype extension.
#               Example: ccal.p should be in the directory
#                        $BENCHDIR/pascal/ccal.
#
# Copyright (c) 1990 by Eindhoven University of Technology
##############################################################################


### STEP 0: Initialize desktop

echo ''
echo ''
echo "    The Architect's Workbench"
echo "    Release 1.7: August, 1989"
echo "    (c) Copyright 1988 Stanford University"
echo ""
echo "    Direct correspondences to awb@umunhum.Stanford.EDU"
echo ''
echo ''

set user       = /usr/user/renebr/awb
set awb        = /awb
set mach       = apollo       #apollo,sun
set BENCHDIR   = $awb/benchmarks
set CONFIGDIR  = $awb/interface/config
set PARAMETERS = $awb/interface/awb.parameters
set MOREPARAMS = $user/more.parameters

set error = $0
set warning = $error:t" WARNING:"
set error = $error:t" ERROR:"

if ($mach == sun) then
    set utrans = u2sun3
else if ($mach == apollo) then
    set utrans = u2a
else
    echo $error unvalid machinetype $utrans
    exit 1
endif

foreach file (  uccawb ufortawb upasawb upas17awb \
                order inter uoptawb memorg \
                hllref $utrans \
                bufsim bufsub srssim srssub mrssim mrssub \
                regstat stackstat dcastat genrstat \
                simblockref simcache simsmcache \
                idsim \
                asscch setcch setrep setsub )
    set $file = $awb/bin/$mach/$file
    if (! -f $awb/bin/$mach/$file ) then
        echo $warning File $file:t not found.
    endif
end

# Parse command line arguments (if any)
if ($#argv >= 1) then
    set bench = $1:r
    set ext   = $1:e
    set what  = $2
endif
repeat $#argv shift # Clear argument variables
```

```
# Select simulation directory
select:
echo ""
echo -n "Please enter simulation directory: "
set SIMDIR = $<
if !( -d $SIMDIR) then
    echo ""
    echo Directory $SIMDIR does not exist.
    if ($SIMDIR != 'dummy') then
        echo -n "Do you want to create $SIMDIR? [y/n] "
        if ( $< =~ y* ) then
            echo ""
            echo mkdir $SIMDIR
            mkdir $SIMDIR
            echo ""
            echo pushd $SIMDIR
            pushd $SIMDIR > /dev/null
        else
            goto select
        endif
    endif
else
    echo pushd $SIMDIR
    pushd $SIMDIR > /dev/null
endif


### STEP 1: Select type of simulation

askwhat:
if !($?what) then
    echo ""
    echo 'What would you like to simulate?'
    echo "  [i]  Instruction traffic"
    echo "  [d]  Data traffic"
    echo "  [id] Integrated I/D traffic"
    echo -n "Your choice [i,d,id] "
    set what = $<
endif
if !(($what == i) || ($what == d) || ($what == id)) then
    echo ""
    echo $error "Invalid selection ($what)"
    unset what
    goto askwhat
endif

asklang:
set sixteen = 0
if ($what != i) then
    echo ''
    echo -n "Do you want to simulate a 16 bit machine (ex. 1750A)? [y,n] "
    if ($< =~ y*) then
        set sixteen = 1
        set ext = p
    endif
endif
if !($?ext) then
    echo ""
    echo Select benchmark language:
    echo -n "Your choice [c,fortran,pascal] "
    set ext = $<
endif
if ($ext =~ c*) then
    set lang = c
    set ext  = c
    set gen  = $uccawb
else if ($ext =~ f*) then
    set lang = fortran
    set ext  = f
    set gen  = $ufortawb
else if ($ext =~ p*) then
    set lang = pascal
    set ext  = p
    set gen  = $upasawb
    if ($sixteen == 1) set gen = $upas17awb
else
```

```
    echo ""
    echo $error "Invalid language ($ext)"
    unset ext
    goto asklang
endif

askbench:
if !($?bench) then
    echo ""
    echo Benchmarks available:
    ls $BENCHDIR/$lang/*/*.$ext | sed 's/^.*\///' | sed "s/\.$ext//" | pr -5 -t -l1
    echo -n "Please choose one: "
    set bench = $<
endif
if !(-f $bench.$ext) then
    if !(-f $BENCHDIR/$lang/$bench/$bench.$ext) then
        echo ""
        echo $error Benchmark $bench.$ext not found
        unset bench
        goto askbench
    else
        echo ""
        echo cp $BENCHDIR/$lang/$bench/$bench.$ext $bench.$ext
        cp $BENCHDIR/$lang/$bench/$bench.$ext $bench.$ext
    endif
endif


### STEP 2: Generate U-code

if !(-x $gen) then
    echo ""
    echo $error "Cannot execute $gen."
    exit 1
endif
echo ""
echo Ready for executing $gen:t
echo -n "Please add options: "
set options = $<
echo ""
echo $gen:t $options $bench.$ext
$gen $options $bench.$ext
if ($status) then
    echo ""
    echo $error "Unexpected result from $gen:t"
    exit 1
endif


### STEP 3: Prepare for instruction simulation

if ($what != d) then
    if (! -f $PARAMETERS) then
        echo ""
        echo $warning File $PARAMETERS not found.
    endif
    if (-f $MOREPARAMS) then
        set PARAMETERS = ($MOREPARAMS $PARAMETERS)
    else
        if (! -f $PARAMETERS) then
            echo ''
            echo $error No parameter files found.
            exit 1
        endif
    endif

    askarch:
    if !($?arch) then
        echo ""
        echo 'Instruction architectures available:'
        egrep '^arch' $PARAMETERS | awk '{ print $2 }' | pr -5 -t -l1
        echo ""
        echo -n "Please choose one: "
        set arch = $<
    endif
```

```
    # Read switches of the specified architecture from parameters file(s).
    set data = `egrep "^arch[ ]*$arch " $PARAMETERS`
    if ($status) then
        echo ""
        echo $error Architecture $arch not in $PARAMETERS
        unset arch
        goto askarch
    endif
    set simulator = $data[3] # stack, reg, genr or dca
    set switches  = "$data[4-]"
    set bdst      = $arch
    if (! -d $bdst) then
        echo ""
        echo mkdir $bdst
        mkdir $bdst
    endif
endif


### STEP 4: Register allocation

if ($what != i) then
    echo ""
    echo REGISTER ALLOCATION FOR DATA CACHE SIMULATION
    askalloc:
    if !($?alloc) then
        echo ""
        echo 'The U-code optimizer Uopt allocates registers.'
        echo 'Mrssim and mrssub cannot take this register allocated ucode.'
        echo ""
        echo 'C does not work very well without optimization in uopt.'
        echo ""
        echo 'Interprocedural register allocation can only be done for Pascal, for'
        echo 'programs which are not going to be simulated with mrssim or mrssub.'
        echo ""
        echo 'Register allocation strategy:'
        echo "  [uopt]  Global many-to-one"
        echo "  [inter] Interprocedural one-to-one"
        echo "  [order] Global ordered one-to-one"
        echo "  [none]  Unoptimized one-to-one"
        echo -n "Your choice [uopt,inter,order,none] "
        set alloc = $<
    endif

    if ($alloc =~ u*) then
        askuoptregs:
# We will need to know the number of registers if we use uoptawb
# register allocation
        echo ''
        echo -n 'Do you want to specify register options? '
        set regs = $<
        if ( $regs =~ y* ) then
            echo ""
            echo "Registers available for GLOBAL ALLOCATION:"
            echo "  [s] separate integer and floating point registers"
            echo "  [u] unified registers"
            echo -n "Your choice [s,u] "
            set regs = $<
            echo ""
            if ($regs =~ s*) then
                echo -n "Number of registers for integers: "
                set regs = -intregs:$<
                echo -n "Number of registers for floats  : "
                set regs = ( $regs -fpregs:$< )
            else if ($regs =~ u*) then
                echo -n "Registers available for global allocation: "
                set regs = -regs:$<
            else
                echo Invalid selection $regs
                goto askuoptregs
            endif
        else
            set regs = ''
        endif
        echo ""
        echo Current options for Uopt are: $regs
```

```
        echo -n "Please add further options: "
        set options = $<
        echo ""
        echo uoptawb $regs $options $bench
        $uoptawb $regs $options $bench
        if ($status) then
            echo ""
            echo $error Unexpected result from uoptawb
            exit 1
        endif
    else
        echo ""
        echo -n "Do you want to reorganize memory? [y,n] "
        set options = $<
        if ($options =~ y*) then
            echo ""
            echo memorg $bench.u
            $memorg $bench.u
            if ($status) then
                echo ""
                echo $error Unexpected result from memorg
                exit 1
            endif
        else
            echo ""
            echo cp $bench.u $bench.iz
            cp $bench.u $bench.iz
        endif
        if ($alloc =~ o*) then
            echo ""
            echo "Current options for Order are: -l:7 -us:3 "
            echo -n "Please add further options: "
            set options = ( -l:7 -us:3 $< )
            echo ""
            echo order $options $bench.iz
            $order $options $bench.iz
            if ($status) then
                echo ""
                echo $error Unexpected result from order
                exit 1
            endif
        else if ($alloc =~ i*) then
            echo ""
            echo Current options for Inter are: -regs:0
            echo -n "Please add further options: "
            set options = $<
            echo ""
            echo inter -regs:0 $options $bench.iz
            $inter -regs:0 $options $bench.iz
            if ($status) then
                echo ""
                echo $error Unexpected result from inter
                exit 1
            endif
            echo ""
            echo mv $bench.{r,z}
            mv $bench.{r,z}
        else if ($alloc =~ n*) then
            echo ""
            echo mv $bench.iz $bench.z
            mv $bench.iz $bench.z
        else
            echo ""
            echo $error "Invalid selection ($alloc)"
            unset alloc
            goto askalloc
        endif
    endif
    echo ''
    echo mv -f $bench.{z,z.d}
    mv -f $bench.{z,z.d}
endif

if ($what != d) then
    echo ""
    echo REGISTER ALLOCATION FOR INSTRUCTION CACHE SIMULATION
```

```
    echo ""
    echo "Include (Uopt) global code optimization?"
    echo -n "Your choice [y/n] "
    if ($< =~ y*) then
        echo Current options for Uopt are: -regs:0
        echo -n "Please add further options: "
        set options = $<
        echo ""
        echo $uoptawb:t -regs:0 $options $bench
        $uoptawb -regs:0 $options $bench
        if ($status) then
            echo ""
            echo $error Unexpected result from uoptawb
            exit 1
        endif
        set switches = ( -u $switches )
    else
        echo ''
        echo cp $bench.u $bench.z
        cp $bench.u $bench.z
    endif
    echo ''
    echo mv -f $bench.{z,z.i}
    mv -f $bench.{z,z.i}
endif


### STEP 5: Static and trace file generation

if ($mach == apollo) set machext = "a"
if ($mach == sun)    set machext = "o"

if ($ext == c) then
    set tracedlib = $awb/lib/$mach/clibdtrace.$machext
else if ($ext == f) then
    set tracedlib = $awb/lib/$mach/flib.$machext
else if ($ext == p) then
    set tracedlib = $awb/lib/$mach/traceplib.$machext
    set traceilib = $awb/lib/$mach/plib.$machext
    if ($sixteen == 1) set hllref = ( $hllref -sixteen )
endif

set kind = data
set help = "z.d"
set tracelib = $tracedlib
if ($what == i) then
    set kind = instr
    set help = "z.i"
    set tracelib = $traceilib
endif

again:
echo ""
echo Current options for hllref are:
echo -n "Please add further options: "
set options = $<
echo ""
echo hllref $options $bench.$help
$hllref $options $bench.$help
if ($status) then
    echo ""
    echo $error Unexpected result from hllref $status
    exit 3
endif

set flag = ''
if ($ext == c) set flag = " -c "
echo ""
echo Current options for $utrans:t are: $flag
echo -n "Please add further options: "
set options = $<
echo ""
echo $utrans:t $flag $options $bench.x
$utrans $flag $options $bench.x
if ($status) then
    echo ""
```

```
    echo $error Unexpected result from $utrans:t
    exit 4
endif

echo ""
if ($mach == apollo) then
    echo mv -f $bench.a $bench.asm
    mv -f $bench.a $bench.asm
else if ($mach == sun) then
    echo mv -f $bench.s $bench.asm
    mv -f $bench.s $bench.asm
endif

echo ""
echo as $bench.asm
as $bench.asm
if ($status) then
    echo ""
    echo $error Unexpected result from assembler
    exit 5
endif

if ($mach == sun) then
    echo ''
    echo mv -f a.out $bench.bin
    mv -f a.out $bench.bin
endif

echo ""
echo ld -o $bench $bench.bin $tracelib
ld -o $bench $bench.bin $tracelib
if ($status) then
    echo ""
    echo $error Unexpected result from loader
    exit 6
endif

# Create trace file (.tr) by running instrumented benchmark:
echo ""
if (-f $bench.in) then
    echo "./$bench < $bench.in > $bench.out"
    ./$bench < $bench.in > $bench.out
else
    echo "./$bench > $bench.out"
    ./$bench > $bench.out
endif

if ($kind == data) then
    echo ""
    echo mv -f $bench.{st,st.d}
    mv -f $bench.{st,st.d}
    echo mv -f $bench.{tr,tr.d}
    mv -f $bench.{tr,tr.d}
    if ($what == id) then
        set kind = instr
        set help = "z.i"
        set tracelib = $traceilib
        goto again
    endif
else
    echo ""
    echo rm -f $bench.st
    rm -f $bench.st
    echo mv -f $bench.{tr,tr.i}
    mv -f $bench.{tr,tr.i}
endif

if ($what != d) then
    echo ""
    if ($simulator =~ dca*) then
        echo dcastat $switches $bench.z.i
        $dcastat $switches $bench.z.i > $bench.st.i
    else if ($simulator =~ reg*) then
        echo regstat $switches $bench.z.i
        $regstat $switches $bench.z.i > $bench.st.i
    else if ($simulator =~ gen*) then
```

```
        echo genrstat $switches $bench.z.i
        $genrstat $switches $bench.z.i > $bench.st.i
    else if ($simulator =~ sta*) then
        echo stackstat $switches $bench.z.i
        $stackstat $switches $bench.z.i > $bench.st.i
    else
        echo $error "Unsupported simulator ($simulator)"
        exit
    endif
endif


### STEP 6: Initialize cache.conf

if ($what != i) then
# Remove old symbolic link
    find . -type l -name cache.conf -exec rm -f {} \;
    if ( -f cache.conf ) then
        echo ""
        echo $error "Cannot run with cache.conf in $cwd"
        exit 1
    endif
# Get cache.conf
    askconfig:
    if !($?config) then
        echo ""
        echo Data architectures:
        ls $CONFIGDIR | pr -5 -t -11
        echo ""
        echo -n "Please select one: "
        set config = $<
    endif
    if !( -f $CONFIGDIR/$config ) then
        echo ""
        echo $error "Invalid data architecture ($config)"
        unset config
        goto askconfig
    endif
    echo ""
    echo ln -s $CONFIGDIR/$config cache.conf
    ln -s $CONFIGDIR/$config cache.conf
endif


### STEP 7: Select buffer or do I cache simulation

# Level 1 simulation

if ($what != d) then
    echo ''
    echo cp $bench.tr.i $bench.tr
    cp $bench.tr.i $bench.tr
    echo cp $bench.st.i $bench.st
    cp $bench.st.i $bench.st
    echo ""
    echo -n "Do you want to generate a dynamic trace [y,n] "
    set options = $<
    if ($options =~ y*) then
        echo ""
        echo -n "Please specify options for Simblockref: "
        set options = $<
        echo $simblockref:t $options $bench
        $simblockref $options $bench > /dev/null
    endif
    echo ""
    echo -n "Do you want to simulate an instruction cache? [y,n] "
    set options = $<
    if ($options =~ y*) then
        asksize:
        if !($?size) then
            echo ""
            echo 'What cache sizes would you like to simulate?'
            echo "  [sm]  128 to 4K"
            echo "  [reg] 512 to 16K"
            echo -n "Pick one [sm,reg] "
            set size = $<
```

```
        endif
        if ($size =~ s*) then
            set cachesim = $simsmcache
        else if ($size =~ r*) then
            set cachesim = $simcache
        else
            echo ""
            echo $error "Invalid selection ($size)"
            unset size
            goto asksize
        endif

        askref:
        if !($?ref) then
            echo ""
            echo 'Select reference architecture (or "self")'
            egrep '^arch' $PARAMETERS | awk '{ print $2 }' | pr -5 -t -l1
            echo ""
            echo -n "Your choice : "
            set ref = $<
        endif
        egrep "^arch[ ]*$ref " $PARAMETERS
        if (($status) && ($ref != self)) then
            echo ""
            echo $error Invalid selection.
            unset ref
            goto askref
        endif

        if ($ref == self) rm -f $bench.mt
        if ($ref != self) then
            if (-f $ref/$bench.sta) then
awk '/^Instruct/ {print substr($6,2)}' $ref/$bench.sta > $bench.mt
            else
                echo ''
                echo "$error $ref/$bench.sta was not found"
                echo Assuming self
                exit 0
            endif
        endif

        echo ''
        echo $cachesim:t $bench
        $cachesim $bench > /dev/null
        echo mv $bench.sta $bdst/$bench.sta
        mv $bench.sta $bdst/$bench.sta
    endif
endif
if ($what == i) then
    exit 0
endif

if ($what != i) then
    askbuffer:
    echo ""
    echo Select data buffer structure:
    echo "  [srs] Single Register Set"
    echo "  [mrs] Multiple Register Set"
    echo "  [buf] Stack Buffer"
    echo -n "Your choice [srs,mrs,buf] "
    set buffer = $<
    echo ''
    echo Select data buffer simulator:
    echo '  [sim] Simulator'
    echo '  [sub] Simulator, on-chip cache has subblocks'
    echo -n "Your choice [sim,sub] "
    set flag = $<

    if ($flag =~ si*) then
        if ($buffer =~ s*) then
            set buffer = $srssim
        else if ($buffer =~ m*) then
            set buffer = $mrssim
        else if ($buffer =~ b*) then
            set buffer = $bufsim
        endif
```

```
    else if ($flag =~ su*) then
        if ($buffer =~ s*) then
            set buffer = $srssub
        else if ($buffer =~ m*) then
            set buffer = $mrssub
        else if ($buffer =~ b*) then
            set buffer = $bufsub
        endif
    else
        goto askbuffer
    endif

    echo ''
    echo mv $bench.tr.d $bench.tr
    mv $bench.tr.d $bench.tr
    echo mv $bench.st.d $bench.st
    mv $bench.st.d $bench.st
    echo ''
    set options = ' '
    echo Current options for $buffer:t are: $options
    echo -n 'Please enter additional options: '
    set options = ( $options $< )
    echo ''
    echo $buffer:t $options $bench
    $buffer $options $bench
endif


### STEP 8: IDsim

if ($what == id) then
    set options = ""
    if ($sixteen == 1) set options = " -sixteen "
    echo ""
    echo mv -f bench.{at,atx}
    mv -f bench.{at,atx}
    echo ''
    echo Current options for Idsim are: $options
    echo -n 'Please enter additional options : '
    set options = ( $options $< )
    echo ''
    echo cp $bench.st.i bench.st
    cp $bench.st.i bench.st
    echo $idsim:t $options $bench < $bench.sti
    $idsim $options $bench < $bench.sti
endif


### STEP 9: D and ID cache simulation

set switches = ""
if ($what == id) set switches = " -instr "
if ($sixteen == 1) set switches = ( $switches -sixteen )

askcache1:
echo ""
echo Select type of associativity:
echo "  [f] Fully-associative cache"
echo "  [s] Set associative cache (or direct mapped)"
echo -n "Your choice [f,s] "
set choice = $<

if ($choice =~ f*) then     #ASSCCH
    echo ""
    echo Current options for asscch are: $switches
    echo -n 'Please enter additional options: '
    set switches = ( $switches $< )
    echo $asscch:t $switches $bench
    $asscch $switches $bench
    echo ""
    if !($status) then
        echo ''
        echo "Cache performance results are in $cwd/$bench.as.cch"
    endif
    exit 0
endif
```

```
echo ""
echo -n "Include subblocks in cache? [y,n] "
set sbic = $<

if ($sbic =~ n*) then     #SETCCH
    echo ""
    echo Select set size:
    echo "  [2] Two"
    echo "  [4] Four"
    echo "  [all] 1, 2 and 4 (LRU only)"    # -r:0 (default)
    echo -n "Your choice [2,4,all] "
    set size = $<
    if !($size =~ a*) then
        echo ""
        echo Replacement strategy:
        echo "  [f] FIFO"
        echo "  [r] Random"
        echo -n "Your choice [f,r] "
        set method = $<
        if ($method =~ r*) then
            set x = 1
        else
            set x = 3
        endif
        if ($size == 4) @ x = $x + 1
        set switches = ( $switches -r:$x )
    endif
    echo ""
    echo Current options for setcch are: $switches
    echo -n 'Please enter additional options: '
    set switches = ( $switches $< )
    echo $setcch:t $switches $bench
    $setcch $switches $bench
    if !($status) then
        echo ''
        echo "Cache performance results are in $cwd/$bench.set.cch"
    endif
    exit 0
endif

echo ""
echo -n "Prefetching of subblocks? [y,n] "
set pfsb = $<
if ($pfsb =~ y*) then
    echo -n "How many subblocks to prefetch? [#] "
    set switches = ( $switches -pref:$< )
else if ($pfsb =~ n*) then
    echo -n "Load forwarding? [y,n] "
    if ($< =~ y*) then     #SETREP
        echo ""
        echo Maximum associativity:
        echo "  [1] One"
        echo "  [2] Two"
        echo "  [4] Four"
        echo -n "Your choice [1,2,4] "
        set maxass = $<
        set switches = ( $switches -a:$maxass )
        if !($maxass == 1) then
            echo ""
            echo Replacement strategy:
            echo "  [f] FIFO"
            echo "  [l] LRU"
            echo "  [r] Random"
            echo -n "Your choice [f,l,r] "
            set method = $<
            if ($method =~ f*) then
                set x = 2
            else if ($method =~ l*) then
                set x = 0
            else
                set x = 1
            endif
            set switches = ( $switches -r:$x )
        endif
        echo ""
```

```
        echo Current options for setrep are: $switches
        echo -n 'Please enter additional options: '
        set switches = ( $switches $< )
        echo $setrep:t $switches $bench
        $setrep $switches $bench
        if !($status) then
            echo ''
            echo "Cache performance results are in $cwd/$bench.set.rep"
        endif
        exit 0
    else
        set switches = ( $switches -forw )
    endif
endif

#SETSUB
echo ""
echo "Maximum (LRU) associativity:"
echo "  [1] One"
echo "  [2] Two"
echo "  [4] Four"
echo -n "Your choice [1,2,4] "
set sets = $<
set switches = ( $switches -amax:$sets )
if !($sets == 1) then
    echo ""
    echo Minimum associativity:
    echo "  [1] One"
    echo "  [2] Two"
    echo -n "Your choice [1,2] "
    set switches = ( $switches -amin:$< )
endif
echo ""
echo Current options for setsub are: $switches
echo -n 'Please enter additional options: '
set switches = ( $switches $< )
echo $setsub:t $switches $bench
$setsub $switches $bench
if !($status) then
    echo ''
    echo "Cache performance results are in $cwd/$bench.set.sub"
endif

exit 0
```

APPENDIX 3: Cache.conf

The configuration file is located in the directory '/awb/interface/config'. The buffer configuration has a default data path to/from cache of 4B and is structured as in the following example:

```
2 3 4 4 32 1024 0 1 0 0 1
1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
```

or:

```
l c i o s n g a f z u
r m
r8
w8
```

in which:

l level (1=first, 2=second, 3=third, 4=fourth)

c configuration (1=buffer, 2=onchip, 3=offchip, 4=main memory)

l,c 2,1 Not allowed

2,2 Like 2,4 but with fully associative cache in second level

2,3 Like 2,4 but with cache simulator in second level

2,4 Normal single/multiple register set or stack buffer

3,3 Like 2,2 but with cache simulator in third level

i datapath from memory in bytes

o datapath to memory in bytes

s blocksize

n blocknumber

g ignore writes (flag)

a allocate on write misses (flag)

f subflag (flag)

z subblocksize

u cachestatus (flag)

r count regular access

m count maintenance access

r8 reads from cache include static, global, stack, structure, heap, frame, io and immediate references.

w8 writes to cache include static, global, stack, structure, heap, frame, io and immediate references.

APPENDIX 4: Gethllref & ucode.h

```
procedure gethllref(var f:text; var r: hllrefrec);
var    ch: char;
    xflag: boolean;
begin
    with r do begin
        repeat read(f,ch) until ch <> ' ';
        indir := direct; mem := zmt; dtype := zdt;
        bytecnt := 0; xflag := false;
        if ch = 'X' then begin
            read(f,regno);
            read(f,ch);
            xflag := true;
        end;
        if ch in ['&','@']  then begin
            if ch = '&' then indir := addressof
            else indir := indirect;
            read(f,ch);
        end;
        if ch in ['E','R'] then begin
            if ch = 'E' then reftype := evalstackref
            else reftype := runtimeref;
            rw := getrwtype(f);
            dtype := getdatatype(f);
            if dtype in [mdt,sdt] then read(f,bytecnt);
        end (* E R *)
        else if ch in ['L','N','G'] then begin
            rw := getrwtype(f);
            rtflag := false;
            if ch = 'L' then reftype := localref
            else if ch = 'G' then reftype :=globalref
            else begin reftype := nlocalref; read(f,block); end;
            mem := getmttype(f);
            read(f,obj);
            if mem in [mmt,fmt,tmt] then obj := -obj;
            dtype := getdatatype(f);
            if dtype in [mdt,sdt] then read(f,bytecnt);
        end (* L N G *)
        else if ch = 'I' then begin
            reftype := immediateref;
            dtype := getdatatype(f);
            read(f,bytecnt);
        end (* I *)
        else if ch = 'S' then begin
            reftype := iostartref;
            rw := getrwtype(f);
            dtype := getdatatype(f);
            if dtype in [sdt,mdt] then read(f,bytecnt);
        end (* S *)
        else if ch = 'F' then
            reftype := frameref
        else err('gethllref: bad reftype',ord(ch));
        if xflag then begin reftype2 := reftype; reftype := extended; end;
    end; (* R *)
end; (* gethllref *)
```

```
(* -- UCODE.INC -- *)

  (* This file contains all types that define U-code *)
const
  (* set constant representation in Ucode *)
  maxoperands = 10;   (* maximum number of operands in u-code instruction + 1 *)
  maxinstlength = 30; (* maximum size of a b-code instruction, in host words, =
                         max (size of largest set constant (in bits), size of
                         largest string constant (in bits)) div wordsize+ 2;  *)
type
  datatype = (adt,    (*address (pointer)*)
              bdt,    (*boolean*)
              cdt,    (*character*)
              edt,    (*procedure entry point*)
              hdt,    (*pointer pointing to heap only*)
              idt,    (*integer, double word*)
              jdt,    (*integer, single word*)
              ldt,    (*non-negative integer, single word*)
              mdt,    (*array or record*)
              pdt,    (*procedure, untyped*)
              qdt,    (*real, double word*)
              rdt,    (*real, single word*)
              sdt,    (*set*)
              zdt);   (*undefined*)


memtype = (
    zmt,    { undefined }
    pmt,    { parameters }
    tmt,    { temporaries }
    smt,    { Statically allocated memory }
    mmt,    { complex variables }
    fmt,    { simple variables }
    rmt,    { registers }
    vmt     { space for arrays/record - tjiang 21-11-85 }
);

  (*constants*)
  valuptr = ^valu;
  stringtext = array [1..strglgth] of char;
  strg = record
       len: 0..strglgth;
       chars: stringtext
     end;
  valu =

  record                            (*describes a constant value*)
    case datatype of
      adt,bdt,cdt,hdt,ldt,jdt: (ival: integer);
      mdt,qdt,rdt,sdt,idt:
    (len: 0..strglgth;
     chars: array [1..strglgth] of char);
  end;

  (*ucode instructions*)

{ Add all new opcodes at the end just before Unop - tjiang 21/11/85 }
  uopcode =
  (uabs,uadd,uadj,uand,ubgn,ubgnb,uchkf,uchkt,uchkh,uchkl,uchkn,uclab,ucomm,
   ucup,ucvt,ucvt2,udata,udead,udec,udef,udif,udiv,udsp,udup,uend,uendb,uent,
   uequ,uexpv,ufjp,ugeq,ugrt,ugoob,uicup,uiequ,uigeq,uigrt,uileq,uiles,
   uilod,uimpm,uimpv,uineq,uinc,uinit,uinn,uinst,uint,uior,uistr,uixa,ulab,
   ulca,ulda,uldc,uldp,uleq,ules,ulex,uloc,ulod,umin,umax,umod,umov,umpy,umst,
   umus,uneg,uneq,unew,unot,unstr,uodd,uoptn,upar,uplod,upop,upstr,uregs,uret,
   urlod,urnd,urstr,usdef,usgs,usqr,ustp,ustr,usub,uswp,usym,utjp,uujp,uuni,
   uvequ,uvgeq,uvgrt,uvles,uvleq,uvmov,uvneq,uxjp,uxor,uzero,ueof,
   uvdef,
   unop);

  bcrec = packed record
     case boolean of false:
    (opc: uopcode;       (* 7 bits *)
     dtype : datatype;   (* 4 bits *)
     mtype : memtype;    (* 2 bits *)
     lexlev : 0..15;     (* 4 bits *)
     i1: 0..65535;       (* 16 bits *) (* used for labels and block numbers *)
```

```
    case uopcode of
         ucvt: (dtype2: datatype);
         uent: (pname: identname;
                case uopcode of
                   ucup: (pop,push,extrnal: 0..127);
               );
         uchkl: (checkval: valu);
         uiequ: (length: integer;
                 case uopcode of
                    uldc: (constval: valu);
                    udata:(areaname: identname);
                    uiequ:(offset: integer;
                           case uopcode of
                              usym: (vname: identname);
                              uinit: (offset2:integer; initval: valu);
                              uxjp: (label2: 0..35535)
                     )
               )
         );
         true: (intarray: array[1..maxinstlength] of integer);
    end (*record*);

  (* source line buffer *)
  sourceline = array [1..strglgth] of char;

  opcstring = array [1..4] of char; (* string representation of an
                                       Uopcode *)
  (* different types of operands in a u-code inustrtion *)

  uoperand = (sdtype, smtype, slexlev, slabel0, slabel1, sblockno,
              sdtype2, spname0, spname1, spop, spush, sexternal, scheckval,
              slength, sconstval, scomment, sareaname, soffset,
              svname0, svname1, soffset2, sinitval, slabel2,
              send);
  (* describes the order and type of operands in a u-code inustrtion *)
  uops = array [1..maxoperands] of uoperand;

  utabrec = record
     format: uops;      (* operands *)
     opcname: opcstring; (* opcode name table *)
     hasconst: boolean; (* true if instruction requirs constant *)
     instlength: 1..maxinstlength; (* length of instruction *)
     end;
```