Report Number: WUCSE-2003-79

2003-12-10

The SimplePipe Toolset Manual

Vinayak Joshi and Mark A. Franklin

SimplePipe is a simulation framework/tool for analyzing performance effects of alternative task allocations in network processors having multiple pipelines where pipeline stages are either processors or dedicated hardware functions. Tasks are defined in terms of sequence of separate C program executions with each sequence representing the functional requirements of a flow, where a flow is defined as the set of packets having the same processing requirements. The assignment of tasks to pipeline stages, selection of number of stages, and determination of processor cache sizes are important designing decisions impacting performance.
... Read complete abstract on page 2.

The SimplePipe Toolset Manual

Vinayak Joshi and Mark A. Franklin

Complete Abstract:

SimplePipe is a simulation framework/tool for analyzing performance effects of alternative task allocations in network processors having multiple pipelines where pipeline stages are either processors or dedicated hardware functions. Tasks are defined in terms of sequence of separate C program executions with each sequence representing the functional requirements of a flow, where a flow is defined as the set of packets having the same processing requirements. The assignment of tasks to pipeline stages, selection of number of stages, and determination of processor cache sizes are important designing decisions impacting performance.

2003-79

The SimplePipe Toolset Manual

Authors: Vinayak Joshi and Mark A. Franklin

December 10, 2003

Abstract: SimplePipe is a simulation framework/tool for analyzing performance effects of alternative task allocations in network processors having multiple pipelines where pipeline stages are either processors or dedicated hardware functions. Tasks are defined interms of sequence of seperate C program executions with each sequence representing the functional requirements of a flow, where a flow is defined as the set of packets having the same processing requirements.

The assignment of tasks to pipeline stages, selection of number of stages, and determination of processor cache sizes are important designing decisions impacting perfromance.

The SimplePipe Toolset Manual (Version 1.0)[1]

Vinayak Joshi and Mark A. Franklin
Computer and Communications Research Center
Washington University in St.Louis, Missouri 63130
{vjoshi,jbf}@ccrc.wustl.edu

1 Introduction

Network processors (NPs) are being increasingly used on router line cards to provide flexibility and programmability in supporting networking functions. These network processors are often connected in a pipelined fashion with each stage of the pipeline consisting of a processor. Having multiple such pipelines further increases performance. Figure 1 illustrates a generic NP architecture that consists of three pipelines, each having three stages. The notation S(i,j) indicates the j^{th} stage in the i^{th} pipeline. Each pipeline executes a set of tasks associated with processing the incoming communications traffic. The incoming traffic consists of a number of *flows* where a *flow* is defined as a set of packets that have the same processing requirements (i.e., one flow might require IP lookup and encryption while another flow might require IP lookup, transcoding and compression). Flows are associated with one or more of the pipelines, and the packets for the flow proceed down the pipeline with each stage successively executing functions required for the associated packet protocol. Each flow is partitioned into an *ordered* set of tasks. These tasks are called *phases*.

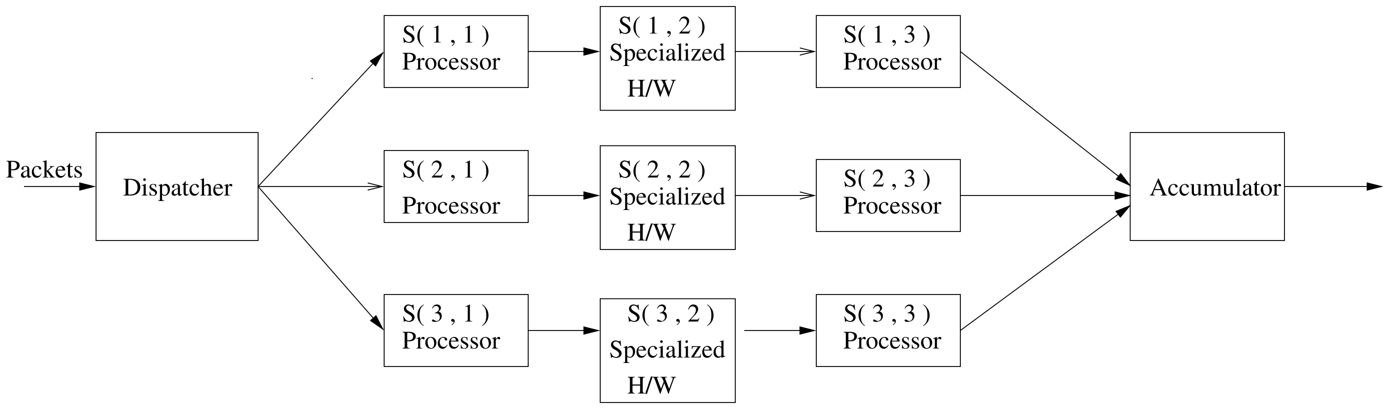

Figure 1: Network Processor Design

Note that a pipeline stage can consist of a general RISC processor (microengine) or can consist of specialized logic hardware. The dispatcher has the decision logic to select pipelines for each flow and packet. The accumulator is the collection point of processed packets.

Network applications are partitioned into phases and, to obtain the highest throughput, these phases need to be assigned to pipeline stages in an optimal or near optimal manner. *SimplePipe* is a simulation tool that permits the modelling and simulation of pipeline of processors with tasks of various flows assigned to them. *SimplePipe* is built upon *SimpleScalar*[2], which is a processor modelling and simulation toolset and the **sim-outorder** tool of *SimpleScalar* toolset is utilized. More information on the motivation and use

[1]This work has been supported in part by NSF grant CCR-0217334

of *SimplePipe* can be found in the report titled "*SimplePipe* : A Simulation Tool for Task Allocation and Design of Processor Pipelines with Application to Network Processors."

2 Overview

An overview of structure of *SimplePipe* is shown in Figure 2.

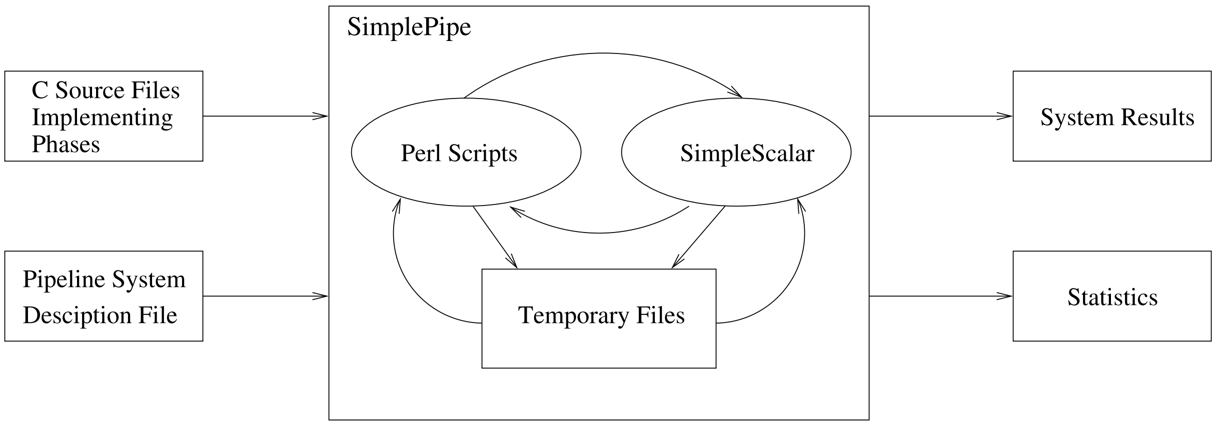

Figure 2: Overview of *SimplePipe*

As indicated, *SimplePipe* takes two inputs. The first input consists of C source files implementing the phases associated with different flows. Inter-phase communication is implemented with a simple file read-process-write mechanism. That is, a phase P_i, after processing the data, writes the information needed by phase P_{i+1} into a temporary file that is subsequently read by P_{i+1} during its execution. The file operations are implemented using the standard I/O library functions. The source code associated with each phase must be compatible with the *xgcc* compiler. Timings for software based phases are based on the performance results of *SimpleScalar* execution of the associated phase C code. Timings for hardware based phases are based on the internal generation of timing by the C code itself. In some cases this may simply be a timing delay that is associated with the hardware implementation of the phase. In other cases the timing may be the result of execution of a model of the hardware (that just happens to be run on the *SimpleScalar* processor). The second input is a text file describing the pipeline stage characteristics, assignment of phases to stages and interstage delay specifications. The format of this file is described in detail in later sections.

3 Installation

To obtain *SimplePipe* e-mail your requests to vjoshi@ccrc.wustl.edu.

SimplePipe is written in the Perl scripting language and it runs on Perl interpreter Version 5 and above. *SimplePipe* is built upon *SimpleScalar* Version 3.0 with some enhancements. *SimplePipe* itself does not need any special procedure for installation. However, it needs installation of *SimpleScalar* along with *xgcc* compilation tool. For instructions of installations of the above, please refer to [1].

4 Usage

4.1 Command Line Format

Typing the following command at the command-line runs *SimplePipe*:

```
$ perl simplepipe.pl
```

SimplePipe accepts the following command line arguments:

–h	print the help message
–conf $\langle file \rangle$	read in and use a configuration (pipeline system description) file. If this option is not specified, *SimplePipe* looks for default configuration file named *'spconfig'* in the current working directory.
–gccd $\langle path \rangle$	directory where *xgcc* compiler is located. Defaults to current working directory.
–simd $\langle path \rangle$	directory where the **sim-outorder** tool of *SimpleScalar* is located. Defaults to current working directory.
–utild $\langle path \rangle$	directory where the **simple-util** utilities are located. Defaults to current working directory.
–srcd $\langle path \rangle$	directory where C source files are located. Defaults to current working directory.
–rf $\langle file \rangle$	file into which simulation results are written. Defaults to console.
–ic	output the total number of instructions executed
–lds	output the total number of loads executed.
–str	output the total number of stores executed.
–cy	output the total CPU cycles executed.
–cpi	output CPI per simulation of one source file.
–et	output the execution time statistics. Also output latency statistics.
–il1m	output L-1 instruction cache miss rate.
–dl1m	output L-1 data cache miss rate.
–ul2m	output L-2 unified cache miss rate.
–clk $\langle frequency \rangle$	clock frequency, in MHz for execution time calculation. Defaults to 2 Ghz. Useful only with -et option.
–all	all output messages generated by **sim-outorder** will be dumped into result output file.

By default -ic, -lds, -str, -cy, -cpi and -et options are turned on. Explicitly specifying one of these options turns it on and turns off the rest.

4.2 Input File Format

The configuration file, specified as part of the *–conf* argument above, contains the description of the pipeline setup and information about how different phases are allocated to various stages of the pipeline. The entries in the configuration file are *not* case sensitive. All characters after '#' in a line are treated as comments.

The file contains six sections; a pipeline description section, a processor description section, a flow specification section, a phase-to-stage assignment section, a stage-to-stage overhead section and source files sections. The sections must be ordered as in the prior sentence. A section begins with the keyword *begin* followed by a colon and the keyword that identifies the section (section identifier). End of a section is indicated with the keyword *end.* Configurations are entered in each section as entity and entity specification pairs in the following format:

```
BEGIN : ⟨ SECTION IDENTIFIER ⟩

   ⟨ENTITY ⟩  : ⟨SPECIFICATION ⟩
   ⟨ENTITY ⟩  : ⟨SPECIFICATION ⟩
                 ...
                 ...
   ⟨ENTITY ⟩  : ⟨SPECIFICATION ⟩

END
```

An entity may specify a part of the system, such as a stage or a phase. A specification may be a numerical value, or a filename, or another entity-specification pair, or a sequence of pairs. If the specification consists of multiple entity-specification pairs, each of those pairs should be entered in a new line.

Any entity that can be identified by a unique number is called a single dimensional entity and it is denoted by an entity name followed by character '-' and an integer index.Indices begin with 1. For example, *PIPELINE-10* denotes the tenth pipeline. Pipelines, flows and processors are single dimensional entities. Entities which require more that one dimension are denoted by multiple single dimensional entities separated by commas. For example, *"PIPELINE-1 , STAGE-1"* denotes the first stage in the first pipeline and, *"FLOW-1 , PHASE-2"* denotes the second phase in the first flow. Character '*' can be used as a short-hand notation to indicate multiple values of indices. For example, *"FLOW-1 , PHASE-*"* denotes all phases in flow 1. The short-hand notation is permitted only with single dimensional entities and the second dimension of two dimensional entities. For example, the notation *"FLOW-* , PHASE-1"* is invalid because short-hand notation is not allowed with the first dimension.

Short-hand notations can be used to denote certain entities. Table 1 lists the entities and their short-hand notations.

Entity	Short-hand notation
flow	F
phase	P
pipeline	PIPE
processor	PROC
stage	S

Table 1: Short-hand notations

Unless otherwise mentioned, a source file can be specified in the following three ways.

1. Specify a C source file name. Whenever the file needs to be executed, it gets compiled with the *xgcc* compiler with the default compiler options. This option is specified with the keyword SRC.
2. Specify the name of a makefile and the name of the executable file to be generated. The makefile is used by *SimplePipe* to obtain the executable file. This option is specified with the keyword MAKE.
3. Specify the name of a precompiled executable file itself. This method avoids repeated compilations. This option is specified with the keyword EXEC.

All the three methods provide a way to pass command line arguments when the source files are executed. The examples shown below describe the three methods.

```
# C Source file example- The file myphase1.c gets compiled with
# xgcc, with the default compiler options. The string "100 200"
# is passed on the command line during execution.
SRC : myphase1.c : 100 200

# Makefile and executable file example- The file "mymakefile" is
# used and results in the executable file named "myexec1".
# The string "xyz" is the command line argument.
MAKE : mymakefile : myexec1 : xyz

# Precompiled executable file example- The executable "myexec2"
# implements -h and -k command line options.
EXEC : myexec2 : -h 20 -k 50
```

4.2.1 Pipeline Description

The pipeline description section specifies the number of pipelines, number of stages in each pipeline and the hardware entities each stage consists of. This section is identified by the keyword *PIPELINE-DESCRIPTION-SECTION.*

The pipeline system in Figure 1 having identical processors and identical specialized hardware elements is described by the following section:

```
BEGIN : PIPELINE-DESCRIPTION-SECTION

        NUMBER-OF-PIPELINES : 3

        NUMBER-OF-STAGES :
                PIPELINE-1 : 3
                PIPELINE-2 : 3
                PIPELINE-3 : 3

        STAGE-DEFINITION :
                PIPELINE-1 :
                        STAGE-1 : SIMPLE-PROCESSOR-1
                        STAGE-2 : SPECIAL-PROCESSOR-1
                        STAGE-3 : SIMPLE-PROCESSOR-1
                PIPELINE-2 :
                        STAGE-1 : SIMPLE-PROCESSOR-1
                        STAGE-2 : SPECIAL-PROCESSOR-1
                        STAGE-3 : SIMPLE-PROCESSOR-1
                PIPELINE-3 :
                        STAGE-1 : SIMPLE-PROCESSOR-1
                        STAGE-2 : SPECIAL-PROCESSOR-1
                        STAGE-3 : SIMPLE-PROCESSOR-1
END
```

4.2.2 Processor Description

The processor description section, identified by the keyword *PROCESSOR-DESCRIPTION-SECTION,* describes the hardware setup of each pipeline stage in the system. If it is a simple processor (i.e., the blocks labeled "processor" in Figure 1) the *SimpleScalar* specification for CPU description should be given. The processor currently simulated by *SimpleScalar Version 3.0* is a RISC processor (DEC ALPHA). Processor characteristics that are configurable include the structure of processor core (e.g., number of available floating point ALUs), the structure of the memory system (e.g., L1 data cache size) and the branch predictor algorithm to be used.

If a special processor (i.e., the blocks labeled "Specialized H/W" in Figure 1) is specified then a source file specification that models the special processor hardware should be entered. When phases allocated to stages that consist of a special processor need to be modelled by *SimplePipe*, it executes the corresponding source file. The model is executed on the native CPU (not on a *SimpleScalar* simulated CPU). Flow number and phase number are passed as the first two command line arguments of the executable that models the special processor. The source file, along with modelling the special processor, also needs to estimate an execution time of the the model. The estimated executed time should be printed on the standard output. The printed value is read by *SimplePipe* from the standard output for system execution time statistics calculations. *SimplePipe* ignores all other messages printed by the model. The execution time should be printed by the model with a C statement (e.g., " printf("\ n SIMPLEPIPE-DATA-EXEC-TIME: %d\n", exec_time);") which results in a single print line with format:

SIMPLEPIPE-DATA-EXEC-TIME : ⟨ Execution time expressed in microseconds ⟩

The *SimplePipe* keyword *SIMPLEPIPE-DATA-EXEC-TIME* distinguishes the execution time message from other messages. The example shown below contains the messages printed by one such model. The message displaying an execution time of 2000 microseconds is processed by *SimplePipe* and the rest are ignored.

```
Starting model 1...
                ...
                ...
SIMPLEPIPE-DATA-EXEC-TIME : 2000
                ...
                ...
```

The example processor description section shown below describes the hardware elements shown in Figure 1. Here the processors have an L1 data cache of 256 sets each with 4-way associativity and line size of 32 bytes. LRU is the cache replacement algorithm. The notation employed here follows that required by *SimpleScalar*. The specialized hardware is implemented by the file *hw.c*. That is, the file contains source code to perform the functions of the specialized hardware. The functions include reading a file to which data is written by phases allocated to stage 1 and writing data to a file which is read by phases allocated to stage 3.

```
BEGIN: PROCESSOR-DESCRIPTION-SECTION
        SIMPLE-PROCESSOR-1 :
                -cache:dl1 dl1:256:4:32:l
        SPECIAL-PROCESSOR-1 :
                SRC : hw.c
END
```

4.2.3 Flow Specification

The flow specification section specifies the total number of flows and the number of phases associated with each flow. This section is identified by the keyword *FLOW-SPECIFICATION-SECTION*. For example, if there are three flows divided into three phases each, the flow specification section should be:

```
BEGIN: FLOW-SPECIFICATION-SECTION
        NUMBER-OF-FLOWS: 3

        NUMBER-OF-PHASES:
                FLOW-1 : 3
                FLOW-2 : 3
                FLOW-3 : 3
END
```

4.2.4 Phase-To-Stage Assignment

The phase-to-stage assignment section describes the assignment of the phases to the stages of different pipelines. This section is identified by the keyword *PHASE-TO-STAGE-ASSIGNMENT-SECTION*. The following example describes a setup where the three phases of the first flow in the example above are assigned to the three stages of the first pipeline, the three phases of the second flow are assigned to the three stages of the second pipeline and so on.

```
BEGIN : PHASE-TO-STAGE-ASSIGNMENT-SECTION

        PIPELINE-1 :
                STAGE-1 :
                        NUMBER-OF-PHASES : 1
                                FLOW-1 , PHASE-1
                STAGE-2 :
                        NUMBER-OF-PHASES : 1
                                FLOW-1 , PHASE-2
                STAGE-3 :
                        NUMBER-OF-PHASES : 1
                                FLOW-1 , PHASE-3

        PIPELINE-2 :
                STAGE-1 :
                        NUMBER-OF-PHASES : 1
                                FLOW-2 , PHASE-1
                STAGE-2 :
                        NUMBER-OF-PHASES : 1
                                FLOW-2 , PHASE-2
                STAGE-3 :
                        NUMBER-OF-PHASES : 1
                                FLOW-2 , PHASE-3

        PIPELINE-3 :
                STAGE-1 :
                        NUMBER-OF-PHASES : 1
                                FLOW-2 , PHASE-1
                STAGE-2 :
                        NUMBER-OF-PHASES : 1
                                FLOW-2 , PHASE-2
                STAGE-3 :
                    NUMBER-OF-PHASES : 1
                          FLOW-2 , PHASE-3
END
```

4.2.5 Stage-To-Stage Overhead

The stage-to-stage overhead section specifies the overhead for each flow incurred in passing data from one pipeline stage to the next stage. This section is identified by the keyword *STAGE-TO-STAGE-OVERHEAD-SECTION*. If the overhead is fixed then delay type is specified with keyword *FIXED* and the delay, in microseconds, is entered. If not, the delay type is specified with the keyword *MODELLED* and a C source file that models the data passing behavior and prints a delay time in microseconds on the standard output. The delay time should be printed in the format similar to the one described in section 4.2.2, however, with one exception; the keyword *SIMPLEPIPE-DATA-EXEC-TIME* should be replaced by the keyword *SIMPLEPIPE-DATA-INTERSTAGE-DELAY*. During the simulation of a flow, the model is executed when data has to be passed across stages. If the delay is not specified for a flow, zero delay is assumed.

The example shown below specifies a fixed delay of 30 microseconds between the first and the second stages of all the three pipelines, for all three flows. The delay between the stages 2 and 3 are not specified and hence is taken to be zero. The delay between the third and the output (a hypothetical stage indicated as “STAGE-OUT”) of all pipelines, for all three flows is modelled by a file *delay.c*. That is, phases executed on the third stage write their output data into an intermediate file. This data is read and processed by the model implemented in the file *d*elay.c.

```
BEGIN: STAGE-TO-STAGE-OVERHEAD-SECTION

      PIPELINE-1 :
            STAGE-1 −⟩ STAGE-2 :
                  FLOW-1 : FIXED : 30
                  FLOW-2 : FIXED : 30
                  FLOW-3 : FIXED : 30
            STAGE-3 −⟩ STAGE-OUT :
                  FLOW-1 : MODELLED : SRC : delay.c
                  FLOW-2 : MODELLED : SRC : delay.c
                  FLOW-3 : MODELLED : SRC : delay.c

      PIPELINE-2 :
            STAGE-1 −⟩ STAGE-2 :
                  FLOW-1 : FIXED : 30
                  FLOW-2 : FIXED : 30
                  FLOW-3 : FIXED : 30
            STAGE-3 −⟩ STAGE-OUT :
                  FLOW-1 : MODELLED : SRC : delay.c
                  FLOW-2 : MODELLED : SRC : delay.c
                  FLOW-3 : MODELLED : SRC : delay.c

      PIPELINE-3 :
            STAGE-1 −⟩ STAGE-2 :
                  FLOW-1 : FIXED : 30
                  FLOW-2 : FIXED : 30
                  FLOW-3 : FIXED : 30
            STAGE-3 −⟩ STAGE-OUT :
                  FLOW-1 : MODELLED : SRC : delay.c
                  FLOW-2 : MODELLED : SRC : delay.c
                  FLOW-3 : MODELLED : SRC : delay.c

END
```

4.2.6 Source Files

The source files section lists the C source files implementing the phases that are allocated to stages with simulated processors. This section is identified by the keyword *SOURCE-FILES-SECTION*.

The example source files section shown below lists the source files implementing the phases in the above example.

```
BEGIN: SOURCE-FILES-SECTION

        PIPELINE-1 :
                STAGE-1 :
                        FLOW-1 , PHASE-1 : f11.c
                STAGE-3 :
                        FLOW-1 , PHASE-3 : f13.c

        PIPELINE-2 :
                STAGE-1 :
                        FLOW-2 , PHASE-1 : f21.c
                STAGE-3 :
                        FLOW-2 , PHASE-3 : f23.c

        PIPELINE-3 :
                STAGE-1 :
                        FLOW-2 , PHASE-1 : f31.c
                STAGE-3 :
                        FLOW-2 , PHASE-3 : f33.c
END
```

As mentioned earlier, data is passed between phases with a read-process-write mechanism. While the phases may spend considerable execution time doing file operations needed to support this mechanism, to obtain correct system timings, file operation times should be excluded by *SimplePipe* when determining the execution times of phases.

If multiple phases of a flow are assigned to the same stage, those phases need to be combined into a single executable unit by compiling the source files of the phases together. The phases need to be executed as successive steps within the unit so that it can be treated as a single entity in the stage. Instead of using the read-process-write technique, inter phase communication has to be achieved through direct procedure calls.

Figure 3 shows an example where phases $P_{j-1}, P_j, P_{j+1}, \ldots\ P_{j+k+1}$ of a flow are assigned to the three stages of a pipeline. Phase P_{j-1} is assigned to stage *i*-1, phases $P_j, P_{j+1}...P_{j+k}$ are assigned to stage *i* and phase P_{i+k+1} is assigned to stage *i*+1. P_{j-1} has to write its results to a file for the use of phase P_j in stage *i*. After P_j processes the data, it has to call a procedure of P_{j+1} for the next step of processing. The next phase P_{j+1} has to call a procedure of P_{j+2} and so on. Finally when the data reaches P_{j+k} it has to write its results to a file for use by P_{j+k+1} in the following stage.

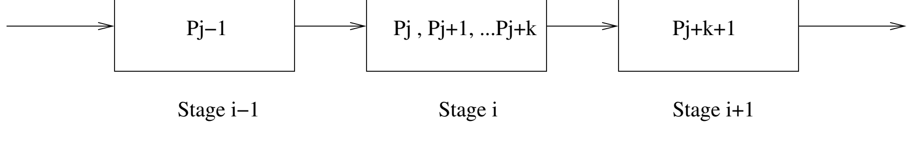

Figure 3: Assignment of Multiple Phases of a Flow to one Pipeline Stage

Earlier in section 4.2 three methods for specifying the source files were presented. However, when aggregating phases to be placed as unit on a single stage different rules must be followed. For this case the precompiled executable and makefile options are not available. *SimplePipe* itself compiles the C source files using *xgcc*.

Additionally,

- Every source file should be divided into three logical sections; an input section, a processing section and an output section, which implement the read-process-write mechanism of the phase. The input section should be placed under the preprocessor directive *#ifdef SIMPIPE_INPUT_i* where *i* is the phase number. Similarly output section should be placed under the preprocessor directive *SIMPIPE_OUTPUT_i*. *SimplePipe* enables the flags as appropriate. If the flag *SIMPIPE_OUTPUT_i* is not enabled then the processing section should call a function in the processing section of the next phase. If not it should call a function in its own output section to write the output.

- The processing section of every phase must be placed between two function calls *SIMPIPE_stat_start()* and SIMPIPE_stat_stop(). That is, before proceeding to the processing section, there should be a call to *SIMPIPE_stat_start()* in the input section. The first statement of the output section should be a call to the function *SIMPIPE_stat_stop()*. *SimplePipe* gathers execution time statistics only for the code that lies between the two function calls. The functions are available in the header file "simplepipe.h" which comes as a part of the *SimplePipe* toolset distribution.

The source files (*phase1.c, phase2.c and phase3.c*) implementing the first three phases of an example flow are given in Figure 4. For this simple example, each phase reads an integer from a file (input), increments it (processing) and writes it back to the file (output). If all three phases are assigned to the same stage they are compiled together resulting in one executable. The preprocessor flags SIMPIPE_INPUT_1 and SIMPIPE_OUTPUT_3 are enabled and SIMPIPE_OUTPUT_1, SIMPIPE_INPUT_2, SIMPIPE_OUTPUT_2 and SIMPIPE_INPUT_3 are disabled during compilation. The flowchart in Figure 5 shows the sequence of functions (along with the filenames and the line numbers where they are defined) called when the final executable runs.

Similarly, if phase 1 and phase 2 are assigned to the same stage, the preprocessor flags SIMPIPE_INPUT_1 and SIMPIPE_OUTPUT_2 are enabled and the flags SIMPIPE_OUTPUT_1 and SIMPIPE_INPUT_2 are disabled during compilation.

4.3 Output file format

The output file contains execution time statistics for individual phases and the system as a whole. CPU related statistics are displayed only for those phases that are simulated on SIMPLE PROCESSOR. An individual phase here is denoted by a six tuple consisting of flow number, phase number, pipeline number stage number, processor number and source file name. For example, the tuple [F-1,P-1,Pipe-1,S-1,Proc-1,f11.c] indicates phase 1 of flow 1 allocated to stage 1 of pipeline 1 which is of processor type 1 and the phase is implemented by source file f11.c.

4.3.1 Execution Time Statistics

Execution time statistics generated by *SimplePipe* are useful in analyzing the the performance of pipelines. Performance of the most heavily loaded stage (bottleneck stage) in a pipeline determines the performance of the entire pipeline. *SimplePipe* identifies the bottleneck stage using two methods; the maximum of maximums method and the maximum of averages method.

4.3.2 Maximum of Maximums Method

This method is suitable for performance analysis of pipelines for the worst case behavior. That is, the maximum possible delay experienced by a packet moving down the pipeline is calculated with this method. Let $\{S_1, S_2, S_3, ..., S_N\}$ denote the N stages of a pipeline P. Let $\{F_1, F_2, F_3, ..., F_M\}$ be the M distinct flows assigned to stage S_i where $1 \leq i \leq N$. That is, one or more phases of each of the M flows are executed on stage S_i. As described in the source files section, if more than one phase of a flow are assigned to a stage, the phases are combined together to form a single executable.

Let T_{ij}^{exe} be the execution time of the phases of flow F_j assigned to stage S_i, where $1 \leq j \leq M$. Let T_{ij}^{mps} be the interstage message passing delay between stages S_i and S_{i+1}, for flow F_j.

The maximum delay experienced by a packet belonging to any of the M flows in stage S_i is given by:

$$D_i^{max} = \max_{j=1}^{M} \; [T_{ij}^{exe} + T_{ij}^{mps}] \tag{1}$$

Hence the maximum delay experienced by a packet moving down pipeline P (bottleneck delay) is given by:

$$D_P^{max} = \max_{i=1}^{N} \; [D_i^{max}] \tag{2}$$

4.3.3 Maximum of Averages Method

This method is useful in analyzing the average performance of pipelines. Assuming the average packet arrival rates of M flows to be equal, the average delay experienced by a packet in stage S_i is given by:

$$D_i^{avg} = \frac{\sum_{j=1}^{M} \; [T_{ij}^{exe} + T_{ij}^{mps}]}{M} \tag{3}$$

Hence the maximum average delay experienced by a packet moving down pipeline P in this case is given by:

$$D_P^{avg} = \max_{i=1}^{N} \; [D_i^{avg}] \tag{4}$$

5 Example

An example of generation of execution time statistics is shown below. Statistics are generated for a single pipeline with three stages which are identical processors. All CPUs have an instruction clock frequency of 500 MHz. A single flow is divided into four phases and the four phases are implemented in source files *phase1.c, phase2.c, phase3.c* and *phase4.c* respectively. For this example, phases just perform dummy functions each having a different execution time. The first two phases are allocated to the first stage and the remaining phases are allocated to the second and the third stages respectively. The processors have an L1 data cache of 256 sets each with 4-way associativity and line size of 32 bytes. LRU is the cache replacement algorithm. The message passing overhead between stage 1 and stage 2 for the flow is a constant value of 10 microseconds. Similarly the overhead between stage 3 and the output is a constant value of 100 microseconds. The overhead between stage 2 and stage 3 is modelled by a file *delay.c* which outputs an overhead value of 20 microseconds on the standard output.

The configuration file describing the above setup is shown below.

```
BEGIN: PIPELINE-DESCRIPTION-SECTION

        NUMBER-OF-PIPELINES : 1

        NUMBER-OF-STAGES :
                PIPELINE-1 : 3
        STAGE-DEFINITION :
                PIPELINE-1 :
                        STAGE-1 : SIMPLE-PROCESSOR-1
                        STAGE-2 : SIMPLE-PROCESSOR-1
                        STAGE-3 : SIMPLE-PROCESSOR-1
END

BEGIN: PROCESSOR-DESCRIPTION-SECTION
        SIMPLE-PROCESSOR-1 :
                -cache:dl1 dl1:256:4:32:l
END

BEGIN: FLOW-SPECIFICATION-SECTION
        NUMBER-OF-FLOWS: 1

        NUMBER-OF-PHASES:
                FLOW-1 : 4
END

BEGIN: PHASE-TO-STAGE-ASSIGNMENT-SECTION

        PIPELINE-1 :
                STAGE-1 :
                        NUMBER-OF-PHASES : 2
                                FLOW-1 , PHASE-1
                                FLOW-1 , PHASE-2
                STAGE-2 :
                        NUMBER-OF-PHASES : 1
                                FLOW-1 , PHASE-3
                STAGE-3 :
                        NUMBER-OF-PHASES : 1
                                FLOW-1 , PHASE-4
END

BEGIN: STAGE-TO-STAGE-OVERHEAD-SECTION

        PIPELINE-1 :
                STAGE-1 −⟩ STAGE-2 :
                        FLOW-1 : FIXED : 10
                STAGE-2 −⟩ STAGE-3 :
                        FLOW-1 : MODELLED : SRC : delay.c
                STAGE-3 −⟩ STAGE-OUT :
                        FLOW-1 : FIXED : 100
END

BEGIN: SOURCE-FILES-SECTION

        PIPELINE-1 :
                STAGE-1 :
                                FLOW-1 , PHASE-1 : phase1.c
                                FLOW-1 , PHASE-2 : phase2.c
                STAGE-2 :
                                FLOW-1 , PHASE-3 : phase3.c
                STAGE-3 :
                                FLOW-1 , PHASE-4 : phase4.c
END
```

Statistics obtained with "–et" option is shown in Figure 6. In the figure, lines 4-20 show the execution times of individual and combined phases. Starting from line 22 various latency statistics are printed. The latency experienced by a packet on flow 1, without taking message passing overhead into account, is 76.258 microseconds, which is the sum of execution times of all four phases. Considering the message passing delays across all stages, the flow latency is $76.258 + 10.0 + 20.0 + 100.0 = 206.258$ microseconds. Since there is only one flow, the average stage latencies follow from the execution times of individual and combined flows assigned to the stages.

Starting from line 39, the performance statistics of the entire pipeline is printed. Since there is only one flow, the values calculated with the two methods described in the previous section are identical. In both cases, the maximum latency, without considering message passing delays, is 39.2725 microseconds and it appears in stage one. It is the sum of execution times of first two phases. The maximum latency with message passing delays, $15.6535 + 100.0 = 115.6535$ microseconds, appears in stage 3. Hence the true bottleneck of the system is in stage 3.

Since the stage with the maximum latency determines the performance of the pipeline, at steady state the system produces one output per 115.6535 microseconds. Hence, in the worst case, the throughput of the system is $1/115.6535 = 0.01$ outputs per microsecond.

6 Use of *SimplePipe* in Pipeline Design

SimplePipe is useful in evaluating alternative assignments of phases to stages. It provides facilities for executing the phases associated with each of the stages in sequence, permitting control and data information to pass between the stages, and capturing the associated performance (e.g., stage latency, cache miss rates, etc.) of the phases executing on each of the processor stages. This process can be repeated with alternative assignments and the best assignment selected. With many phases, stages and flows the optimal assignment cannot be done in this manner since the problem is within the class of NP hard task scheduling problems. However, basic timing information can be obtained which is useful in applying advanced assignment algorithms.

SimplePipe is now being expanded so that the effects of shared memory contention in various memory and cache architectures can be explored. A companion tool *GreedyPipe*[3] is also available to perform near optimum task/phase to pipeline stage assigments.

References

[1] SimpleScalar Documentation. www.simplescalar.com.

[2] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modelling. In *IEEE Computer*, February 2002.

[3] Seema Datar and Mark A. Franklin. Task Scheduling of Processor Pipelines with Application to Network Processors . Technical Report 60, Dept of Computer Science and Engineering, Washington University in St Louis, Missouri, 2003.

File : phase1.c

```
1:   #include <stdio.h>
2:   #include "./simpipe.h"
3:
4:   /* If this file doesnt write output then
5:    the processing functions of phase 2 is called. */
6:   extern void phase2_process( int x );
7:
8:   /* Processing section of this phase.*/
9:   static void phase1_process( int x )
10:  {
11:    x++;
12:  #ifdef SIMPIPE_OUTPUT_1
13:    phase1_output( x );
14:  #else
15:    phase2_process( x );
16:  #endif
17:  }
18:
19:  #ifdef SIMPIPE_OUTPUT_1
20:  /* Output section of this phase.*/
21:  void phase1_output( int x )
22:  {
23:    FILE *fp_out;
24:
25:    SIMPIPE_stat_stop();
26:
27:    fp_out = fopen( "valuefile", "w" );
28:    fprintf( fp_out ,"%d\n", x );
29:    fclose(fp_out);
30:  }
31:  #endif
32:
33:  #ifdef SIMPIPE_INPUT_1
34:  int main( void )
35:  {
36:    int   value;
37:    FILE *fp_in;
38:
39:    /* Perform input */
40:    fp_in = fopen( "valuefile" , "r" );
41:    fscanf( fp_in , "%d", &value );
42:    fclose(fp_in);
43:
44:    /* Begin statistics gathering */
45:    SIMPIPE_stat_start();
46:    phase1_process( value);
47:  }
48:  #endif
```

File: phase2.c

```
1:   #include <stdio.h>
2:   #include "./simpipe.h"
3:
4:   /*If this file doesnt write output then
5:    the processing function of phase 3 is called.*/
6:   extern void phase3_process( int x );
7:
8:   /*  Processing section of this phase. */
9:   void phase2_process( int x )
10:  {
11:    x++;
12:  #ifdef SIMPIPE_OUTPUT_2
13:    phase2_output( x );
14:  #else
15:    phase3_process( x );
16:  #endif
17:  }
18:
19:  #ifdef SIMPIPE_OUTPUT_2
20:  /*  Output section of this phase. */
21:  static void phase2_output( int x )
22:  {
23:    FILE *fp_out;
24:
25:    SIMPIPE_stat_stop();
26:
27:    fp_out = fopen( "valuefile", "w" );
28:    fprintf( fp_out ,"%d\n", x );
29:    fclose(fp_out);
30:  }
31:  #endif
32:
33:  #ifdef SIMPIPE_INPUT_2
34:  int main( void )
35:  {
36:    int   value;
37:    FILE *fp_in;
38:
39:    /* Perform input */
40:    fp_in = fopen( "valuefile" , "r" );
41:    fscanf( fp_in ,"%d", &value );
42:    fclose(fp_in);
43:
44:    /* Begin statistics gathering */
45:    SIMPIPE_stat_start();
46:    phase2_process( value);
47:  }
48:  #endif
```

File: phase3.c

```
1: #include <stdio.h>
2: #include "./simpipe.h"
3:
4: /* If this phase doesnt write output then
5:  the processing function of phase 4 is called.*/
6: extern void phase4_process( int x );
7:
8: /*  Processing section of this phase. */
9: void phase3_process( int x )
10: {
11:   x++;
12: #ifdef SIMPIPE_OUTPUT_3
13:   phase3_output( x );
14: #else
15:   phase4_process( x );
16: #endif
17: }
18:
19: #ifdef SIMPIPE_OUTPUT_3
20: /* Output section of this phase. */
21: static void phase3_output( int x )
22: {
23:   FILE *fp_out;
24:
25:   SIMPIPE_stat_stop();
26:
27:   fp_out = fopen( "valuefile", "w" );
28:   fprintf( fp_out ,"%d\n", x );
29:   fclose(fp_out);
30: }
31: #endif
32:
33: #ifdef SIMPIPE_INPUT_3
34: int main( void )
35: {
36:   int   value;
37:   FILE *fp_in;
38:
39:   /*  Perform input */
40:   fp_in = fopen( "valuefile" , "r" );
41:   fscanf( fp_in ,"%d", &value );
42:   fclose(fp_in);
43:
44:   /* Start statistics gathering. */
45:   SIMPIPE_stat_start();
46:   phase3_process( value);
47: }
48: #endif
```

Figure 4: Source Files for Phase-1, Phase-2 and Phase-3

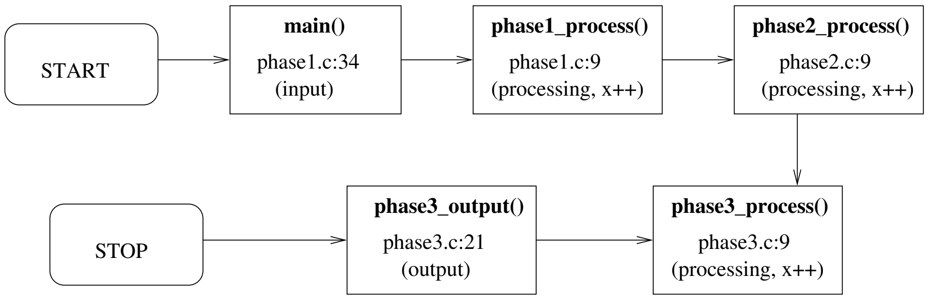

Figure 5: Sequence of Function Calls with all Phases on a Single Stage (phase#.c: line#)

```
==============================================================
=                         SimplePipe Statistics                          =
==============================================================
Pipeline : 1

    Stage : 1

        [F−1,P−1,Pipe−1,S−1,Proc−1,phase1.c] :
        [F−1,P−2,Pipe−1,S−1,Proc−1,phase2.c] :
            Execution Time      = 39.2725 microsecs

    Stage : 2

        [F−1,P−3,Pipe−1,S−2,Proc−1,phase3.c] :
            Execution Time      = 21.332 microsecs

    Stage : 3

        [F−1,P−4,Pipe−1,S−3,Proc−1,phase4.c] :
            Execution Time      = 15.6535 microsecs

Pipeline Latency  :
  Latency Per Flow  :
    Flow   : 1
        WITHOUT Message Passing Overhead : 76.258 microsecs
        WITH Message Passing Overhead    : 206.258 microsecs

  Average Latency Per Stage  :
    Stage   : 1
        WITHOUT Message Passing Overhead : 39.2725 microsecs
        WITH Message Passing Overhead    : 49.2725 microsecs
    Stage   : 2
        WITHOUT Message Passing Overhead : 21.332 microsecs
        WITH Message Passing Overhead    : 41.332 microsecs
    Stage   : 3
        WITHOUT Message Passing Overhead : 15.6535 microsecs
        WITH Message Passing Overhead    : 115.6535 microsecs

Maximum Stage Latency :
  Maximum of Maximums Method:
    WITHOUT Message Passing Overhead : 39.2725 microsecs in Stage 1
    WITH Message Passing Overhead    : 115.6535 microsecs in Stage 3

    Pipeline Throughput = 0.01 outputs/microsec

  Maximum of Averages Method:
    WITHOUT Message Passing Overhead : 39.2725 microsecs in Stage 1
    WITH Message Passing Overhead    : 115.6535 microsecs in Stage 3

    Pipeline Throughput = 0.01 outputs/microsec
```

Figure 6: *SimplePipe* Output