Guillaume Thomassin

Applications for Handhelds: Profiling and Analysis of MediaBench

Student Thesis SA-2001.07
Winter Term 2000/2001

Tutors: Dr. Marco Platzner, Rolf Enzler (IfE)

Supervisor:
Prof. Dr. Lothar Thiele
Abstract

There is a great demand for more powerful handhelds devices: they are asked to be cheap, have long battery life and still be performant. The Zippy project aims are to fulfill these goals by designing a new dynamically reconfigurable processor architecture.

In this term project, the multimedia oriented MediaBench benchmark suite was ported to the SimpleScalar processor simulator tool set. Apart from the PGP application, the port does not need big modifications in the MediaBench programs nor in SimpleScalar. Nine of the eleven Mediabench programs were analyzed with the SimpleScalar simulators and with post-processing scripts. They were simulated on two different architectures: a StrongARM-like and the default SimpleScalar processor architecture, which is a general purpose processor model.

The results are that the programs rely heavily on integer operations, that the branch prediction hit rates are not really high due to data dependent branches, the data cache hit rates greatly vary in function of the program. An estimation of the applications memory footprint was calculated. The runtime intensive function were outlined in all the applications. The different behaviour depending on the processor architecture chosen was also analyzed.
Contents

Introduction ........................................ 3

1 The tools used ..................................... 5
   1.1 SimpleScalar .................................... 5
       1.1.1 The simulators ............................... 5
       1.1.2 The other tools provided with SimpleScalar....... 6
   1.2 MediaBench .................................... 6

2 Simulation methodology ............................ 8
   2.1 Compilation of MediaBench ....................... 8
   2.2 Changes to SimpleScalar ......................... 9
   2.3 The processor architectures used ................... 9
   2.4 Automation and post-processing scripts .......... 10
       2.4.1 Automation scripts .......................... 10
       2.4.2 Post-processing scripts ...................... 11

3 Results ............................................ 16
   3.1 Instruction class mix ............................ 16
   3.2 Branch prediction hit rate ....................... 16
   3.3 Instruction cache hit rate ....................... 16
   3.4 Data cache hit rate ............................. 18
   3.5 Memory footprint ................................ 18
   3.6 Function breakdown .............................. 20

Conclusion .......................................... 28

Bibliography .......................................... 29

A Processor configuration file for SimpleScalar .... 30
B Automation scripts 34
   B.1 benchmark.sh ........................................ 34
   B.2 An example for a specific benchmark: adpcm_benchmark.sh 35

C Postprocessing scripts 36
   C.1 cycle_by_function.pl .................................... 36
   C.2 insn_by_function.pl ..................................... 38

D Extract form loader.c 40
List of Figures

3.1 Instruction class mix ........................................ 17
3.2 Branch prediction hit rate ................................. 17
3.3 Instruction cache hit rate ................................. 18
3.4 Data cache hit rate ......................................... 19
3.5 Program text (code) size (in bytes) ...................... 20
3.6 Program data size (in bytes) .............................. 21
3.7 Total memory pages allocated (in bytes) ............... 21
3.8 Function breakdown (in cycles) for the StrongARM architecture ........................................ 23
3.9 Function breakdown (in cycles) for the default architecture ........................................ 24
3.10 Function breakdown (in instructions) ................... 25

List of Tables

2.1 Command line parameters for the MediaBench programs 12
2.2 Extract from a pipeline trace .............................. 14
3.1 Instruction per cycle for the programs .................. 26
3.2 Instruction per cycle for the most important functions . 27
Introduction

Over the last years, personal computing devices have been more and more common: from PDAs to more and more sophisticated mobile phones. But their current capabilities are still limited (in terms of computing performance and power consumption). The development of new wireless communication technology, like UMTS, which will allow much bigger bandwidth will urge the development of more powerful handheld and wearable devices.

Future generation of such devices will greatly extend current capabilities by combining mobile network access with full audio/video communication, organizing and planning tools, and e-commerce applications.

The most important requirements for embedded processors in these domains are:

• a sufficient performance to run real-time tasks such as audio and video decoding,
• a low power consumption to increase battery life time, and
• low cost, as handhelds and wearables are consumer devices.

Embedded processors for handhelds are typical client processors, where the goal is not to maximize performance but power- and area efficiency.

The aims of the ZIPPY project at ETH (http://www.zippy.ethz.ch/) are to define a new benchmark for embedded application domains of handhelds and wearables and to design a dynamically reconfigurable embedded processor architecture that achieves a performance at least one order of magnitude higher than processor architectures using a comparable amount of silicon area and energy.

This term project concentrated on the first goal: the MediaBench benchmark was analysed with the SimpleScalar processor simulator. The different parts of this term project were:

- The compilation of the programs on Solaris
- The port and compilation of the programs on the SimpleScalar architecture

- Modifications of the simulators to profile the program functions by instruction count and cycle count

- Profiling of the programs to obtain informations such as instruction class mix, cache hit rates, etc., and outlining runtime intensive functions
1 The tools used

1.1 SimpleScalar

SimpleScalar [1] [2] [3] is a tool set to simulate processors. It consists of compiler, assembler, linker, simulation and visualization tools for the SimpleScalar architecture, which uses a close derivative of the MIPS Instruction Set [4]. It is freely available with source code and documentation from http://www.simplescalar.org/. It has been developed by Todd Austin and is now supported by Doug Burger. It was first released in July 1996 and the second release happened in January 1997. It is portable (it runs on most Unix-like machines), the simulator can support multiple Instruction Sets and can be extended.

1.1.1 The simulators

There are six simulators which differ in their speed and the details of the results.

sim-fast and sim-safe are the two fastest simulators and the two simplest. They only do functional simulation, sim-fast does not check alignment and access permission for each memory reference whereas sim-safe does. They allow to test the functionality of the executable that has been compiled.

Two simulators allow functional cache simulation with fully configurable caches (instruction and data, first and second level, associativity, …): sim-cache and sim-cheetah. These simulators are ideal for fast simulation of caches if the effect of cache performance on execution time is not needed.

sim-profile is a functional simulator which is able to produce varied profile information. It can generate detailed profiles on instruction classes and addresses, text symbols, memory accesses, branches and data segment symbols.

The most complicated and detailed simulator is sim-outorder, and as a
consequence also the slowest. This simulator supports out-of-order issue and
evaluation. sim-outorder can, for example, trace the pipeline cycle by cycle.
The simulated processor can be significantly configured: number of ALUs
(integer and floating point), RUU (Register Update Unit) capacity, cache
and memory latency, memory bus width, branch predictor model (taken, not
taken, perfect, bimodal, 2-level adaptive), . . .

The speed difference between the simulators is quite important: for ex-
ample for the same program on the same unloaded machine sim-safe sim-
ulates 2.73 millions instructions per second, sim-profile 682 thousands and
sim-outorder 66590 (sim-outorder runs 40 times slower than sim-safe in this
case).

1.1.2 The other tools provided with SimpleScalar

SimpleScalar comes with all the tools needed to build programs. There is
a port for different compilers: gcc for C programs, f2c for FORTRAN pro-
grams and as the GNU assembler. ar and ranlib ports are also provided to
build libraries. The version 1.09 of the GNU C libraries and version 2.5.2
of the GNU binary utilities are also provided. SimpleScalar also comes with
textprof.pl, a text segment profile viewer written in Perl.

1.2 MediaBench

MediaBench [5] is a benchmark suite composed of multimedia programs. Its
aim is to benchmark architectures for a multimedia utilization, as
opposed to SPEC benchmarks [6] for example. It is available from:
http://www.cs.ucla.edu/~leec/mediabench/. MediaBench is composed
of complete applications coded in high-level languages. All of the applica-
tions are publicly available, making the suite available to a wider user com-
munity. MediaBench 1.0 contains 19 applications culled from available image
processing, communications and DSP applications. The components include:

- JPEG: JPEG is a standardized compression method for full-color and
gray-scale images. JPEG is lossy. Two applications are derived from
the source code: cjepg does image compression and djpeg, which does
decompression.

- MPEG: MPEG2 is the current dominant standard for high-quality dig-
ital video transmission and is also used for DVDs. The important com-
puting kernel is a motion estimation for coding and the inverse discrete
cosine transform for decoding. The two applications used are *mpeg2enc* for encoding and *mpeg2dec* for decoding.

- **GSM**: European GSM 06.10 provisional standard for full-rate speech transcoding, prI-ETS 300 036, which uses residual pulse excitation/long term prediction coding at 13 kbit/s. GSM 06.10 compresses frames of 160 13-bit samples (8 kHz sampling rate, i.e. a frame rate of 50 Hz) into 260 bits.

- **G.721**: Reference implementations of the CCITT (International Telegraph and Telephone Consultative Committee) G.711, G.721 and G.723 voice compressions.

- **PGP**: PGP stands for "Pretty Good Privacy", and permits encrypting and signing data. The signature is computed using a 128-bit cryptographically strong one-way hash function of the message (MD5). To encrypt data PGP uses a block-cipher IDEA, RSA for key management and digital signatures.

- **PEGWIT**: A program for public key encryption and authentication. It uses an elliptic curve over GF(2^{255}), SHA1 for hashing, and the symmetric block cipher square.

- **Ghostscript**: An interpreter for the PostScript language. The single application for Ghostscript is *gs*, which does file I/O but no graphical display.

- **Mesa**: Mesa is a 3-D graphics library clone of OpenGL.

- **RASTA**: A program for speech recognition that supports the following techniques: PLP, Rasta and Jah-RASTA. The technique handles additive noise and spectral distortion simultaneously, by filtering the temporal trajectories of a non-linearly transformed critical band spectrum.

- **EPIC**: An experimental image compression utility. The compression algorithms are based on a bi-orthogonal critically sampled dyadic wavelet decomposition and a combined run-length/Huffman entropy coder. The filters have been designed to allow extremely fast decoding without floating-point hardware.

- **ADPCM**: Adaptive differential pulse code modulation is one of the simplest and oldest forms of audio coding.
2 Simulation methodology

Because the project concentrates on handheld devices, the graphic applications Ghostscript and Mesa were not included in the programs being analyzed. For the Zippy project, it was decided that high-performance graphic applications will not be a target for future handhelds. In order to analyze the other programs they were compiled under SimpleScalar and Sun. The output files of the SimpleScalar and Sun binaries were compared to check if the SimpleScalar binaries are functionally correct.

2.1 Compilation of MediaBench

The compilation of the Sun binaries went without any problem, the provided Makefiles needed only to be adapted to change include paths. The compilation for SimpleScalar was much less smooth. The Makefiles of all the programs were modified to use *ssbig-na-sstrix-gcc* as compiler and linker, and the *ar* port provided with SimpleScalar for the programs for which libraries have to be built.

For MPEG2 the SimpleScalar binaries were not compiled with the X11 library.

For Rasta the SPHERE libraries had to be recompiled for SimpleScalar (mediabench comes with precompiled version of the libraries for SUN). The source code of the SPHERE libraries can be found here: [ftp://jaguar.ncsl.nist.gov/pub/sphere_2.6a.tgz](ftp://jaguar.ncsl.nist.gov/pub/sphere_2.6a.tgz).

For PGP, the line 60 of the Makefile must be uncommented: `BYTEORDER= -DHIGHFIRST` and `$\texttt{(BYTEORDER)}$ added in the CFLAGS variable of the *mips-ultrix* build (line 409). The corresponding CFLAGS is therefore:

```
CFLAGS="$(RSA1NCDIR) $(DBG) -DUNIX -DPORTABLE $(BYTEORDER) \\
-DMPORTABLE -DUSE_SELECT -DIDEA32"
```

The binary is then compiled with *make mips-ultrix*. 
2.2 Changes to SimpleScalar

SimpleScalar needs a small change to run PGP, for the other programs it doesn’t need any change. PGP uses some system calls which are not implemented in SimpleScalar. SimpleScalar’s normal behaviour in this case is to issue an error message (invalid/unimplemented system call encountered, code %d) and stop the execution of the program. The system calls are handled in the syscall.c file. By commenting out the line 2010 of syscall.c (for the PISA architecture), unimplemented system calls don’t cause the end of the execution of the program anymore and the PGP functionality benchmarked\(^1\) still works.

2.3 The processor architectures used

The programs were simulated for two different processor architectures. The first one is an StrongARM-like processor [7]. It has following features:

- 1 integer ALU
- 1 integer multiplier/divider
- 1 floating point ALU
- 1 floating point multiplier/divider
- a 32-way set-associative 16 Kbyte instruction cache\(^2\)
- a 32-way set-associative 16 KByte data cache
- a memory latency of 8 cycles for the first chunk and 2 for the rest of a burst access
- a 4 bytes wide memory access bus
- a bimodal branch predictor

(see Appendix A for the configuration file with all the options).

The second one is the default processor from SimpleScalar:

\(^1\)Encrypting and decrypting a text file. Importing keys in a key ring doesn’t work for example

\(^2\)The cache size is 16 KByte but SimpleScalar instructions are 64 bit long, the 32 first for the instruction and the 32 last bits for flags and information for the simulator. Therefore only 8 KBytes are left for the ”real” instructions. This means our StrongARM-like processor model has an actual instruction cache of 8 Kb, which is half the size of current StrongARM SA-1110 processor [7]
• 4 integer ALU
• 1 integer multiplier/divider
• 4 floating point ALU
• 1 floating point multiplier/divider
• a direct-mapped 16 Kbyte L1 instruction cache
• a 4-way set-associative 16 KByte L1 data cache
• a 4-way set-associative 256 KByte L2 unified cache
• a memory latency of 18 cycles for the first chunk and 2 for the rest of a burst access
• a 8 bytes wide memory access bus
• a bimodal branch predictor

Further informations about the default processor can be found by running sim-outorder without any options.

2.4 Automation and post-processing scripts

Several scripts were written to help run the benchmarks and do some post-processing. The automation scripts do not test for missing files and won’t tell if a benchmark didn’t work.

2.4.1 Automation scripts

The command lines for the simulators are quite difficult to understand and are quite long. Shell scripts were written to run the simulations in a simpler way. There is a central script: benchmark.sh (see Appendix B.1) which is used to launch all the benchmarks (either all at once or only one). The syntax of the command line is:

```
./benchmark.sh all  to run all the benchmarks or
./benchmark.sh adpcm to run the adpcm benchmark for example.
```

In this script all the common arguments are set:

• path to the benchmark programs (variable BENCH_HOME),
• path to the post-processing scripts (see 2.4.2) (CYCLE_SCRIPT and INSN_SCRIPT),
• processor architecture used and other common simulator options (PROC_PARAM),
• file extension for the simulator outputs (EXTENSION).

This script then runs the script corresponding to the benchmark wanted (the variables are exported to the scripts).

The scripts for each benchmark program contains the command line to run the simulator with the options of MediaBench, and the command lines for the post-processing scripts.

The command line used for each programs is given in table 2.1. The command line options for the simulators are:

    sim-profile -iclass -redir:sim <simulation output file> \ 
    <program command line>

and

    sim-outorder $PROC_PARAM -pcstat sim_cycle -pcstat \ 
    sim_num_insn -redir:sim <simulation output file> <program command line>

2.4.2 Post-processing scripts

Some of the information useful for the project about the benchmark programs could not be given by SimpleScalar, like for example the number of instructions and cycles that each function took. sim-profile is capable to give the number of instructions for a function if it is not defined statically (if a function is defined statically the number of instructions the function uses is added to the calling function). Unfortunately, several programs in the benchmark have statically defined functions (cjpeg, djpeg, mpeg2enc and mpeg2dec) which do represent a big proportion of the total calculation, like for example the dist1 function in mpeg2enc. The perl scripts cycle_by_function.pl (see Appendix C.1) and insn_by_function.pl (see Appendix C.2) respectively give, a list of functions ordered by number of cycles used and number of instructions used. They both work the same way, they are just analyzing a different part of the SimpleScalar output. sim-outorder can give informations for each code address. sim-outorder is told which information is wanted with the command line option -pcstat <statistic name>. The statistic analyzed by cycle_by_function.pl is sim_cycle and sim_num_insn for insn_by_function.pl. The format of the output file (for the part which is of interest) is as follows:

    <statistic name>_by_pc
    <statistic name>_by_pc.count = xxxx
    <statistic name>_by_pc.total = xxxx
    <statistic name>_by_pc.imin = xxx
<table>
<thead>
<tr>
<th>Benchmark name</th>
<th>Binary</th>
<th>Options</th>
<th>Input file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adpcm decode</td>
<td>rawdaudio</td>
<td>-o &lt;output file name&gt;</td>
<td>&lt; clinton.adpcm</td>
</tr>
<tr>
<td>Adpcm encode</td>
<td>rawcaudio</td>
<td>-o &lt;output file name&gt;</td>
<td>&lt; clinton.pcm</td>
</tr>
<tr>
<td>Epic epic</td>
<td>epic</td>
<td>-o &lt;output file name&gt;</td>
<td>&lt; test_image.pgm</td>
</tr>
<tr>
<td>Epic unepic</td>
<td>unepic</td>
<td>-o &lt;output file name&gt;</td>
<td>&lt; test.image.pgm.E</td>
</tr>
<tr>
<td>g721 decode</td>
<td>decode</td>
<td>-4 -l -f clinton.g721</td>
<td></td>
</tr>
<tr>
<td>g721 encode</td>
<td>encode</td>
<td>-4 -l -f clinton.pcm</td>
<td></td>
</tr>
<tr>
<td>gsm decode</td>
<td>untoast</td>
<td>-fpl clinton.pcm.run.gsm</td>
<td></td>
</tr>
<tr>
<td>gsm encode</td>
<td>toast</td>
<td>-fpl clinton.pcm</td>
<td></td>
</tr>
<tr>
<td>jpeg decode</td>
<td>djpeg</td>
<td>-dct int -ppm</td>
<td>testing.jpg</td>
</tr>
<tr>
<td>jpeg encode</td>
<td>cjpeg</td>
<td>-dct int -progressive -opt</td>
<td>testing.ppm</td>
</tr>
<tr>
<td>MPEG2 decode</td>
<td>mpeg2dec</td>
<td>-r -f -b mei16v2.m2v</td>
<td></td>
</tr>
<tr>
<td>MPEG2 encode</td>
<td>mpeg2enc</td>
<td>options.par &lt;output file name&gt;</td>
<td>&lt; my.sec</td>
</tr>
<tr>
<td>Rasta decode</td>
<td>rasta</td>
<td>-z -A -J -S 8000 -n 12</td>
<td>&lt; ex5_e1.wav</td>
</tr>
<tr>
<td>Pegwit decode</td>
<td>pegwit</td>
<td>-d pegwint.enc &lt;output file name&gt;</td>
<td></td>
</tr>
<tr>
<td>Pegwit encode</td>
<td>pegwit</td>
<td>-e my.pub pgptest.plain</td>
<td>&lt; encryption_junk</td>
</tr>
<tr>
<td>PGP decode</td>
<td>pgp</td>
<td>-fdb -zbillms</td>
<td>&lt; pgptext.pgp</td>
</tr>
<tr>
<td>PGP encode</td>
<td>pgp</td>
<td>-fes Bill -zbillms -u Bill</td>
<td>&lt; pgptest.plain</td>
</tr>
</tbody>
</table>

\(^a\)The \textit{options.par} file contains the options for encoding the MPEG2 stream and \textbf{absolute paths} that needs to be adjusted to make it work.
\(^b\)The test key has first to be included into keyrings (this must not be done with the pgp built for SimpleScalar):

1. Create a sub-directory named .\texttt{pgp} in a home directory.
2. \texttt{pgp -ka billms prv.pgp} \$(HOME)/.\texttt{pgp/secring.pgp}
3. \texttt{pgp -ka billms pub.pgp} \$(HOME)/.\texttt{pgp/pubring.pgp}

Table 2.1: Command line parameters for the MediaBench programs
AAAAAA is the text address and CCCCCCCC is the value of the counter in hexadecimal for this text address.

The script starts by parsing the output file until it gets to the beginning of statistic. Then for each line it calls the function add_cycles_to_function with the text address and the value of the counter as parameters until it reaches the end of the statistic. The function add_cycles_to_function parses the disassembly file to look to which function this text address belongs. The format of the disassembly file is as follows: each text address is on a single line: the text address (preceded by 00), then there is the function name with the offset in brackets ($<function+offset>$) and then the assembler instruction. The function extracts the name of the function by taking the string which is between the first < and the first > or + whichever comes first. So you have for example:

```
004003e0 <adpcm_coder+60> addiu $t9[25],$t9[25],4592
```

The value of the counter is then added to the value for this function which is stored in a hash. Then the hash is ordered and printed on the screen.

The value of the counter for the sim_cycle statistic is a little bit special. The problem is to count how many cycles an instruction takes to be executed when you have a pipeline and so you can’t add the differences cycle(commit)-cycle(fetch) for each instruction each time the instruction is executed. Counting this way would give numbers without any interesting meaning. SimpleScalar solves this problem by counting the number of cycles between this instruction fetch and the previous instruction fetch for each instruction: if the instruction A is fetched at cycle 1000 and instruction B is fetched at cycle 1002, the counter of B is incremented by 2.

---

3The counter is always written with 8 digits whereas some text addresses are shorter than 6 digits.
4The disassembly file is obtained with objdump:

```bash
objdump -dl <executable file> > <disassembly file>
```
<table>
<thead>
<tr>
<th>@ 2560</th>
<th>ae = '0x004002e0: [internal ld/st]'</th>
<th>af = '0x004002e8: jal 0x4001f0'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[IF] [DA] [EX] [WB] [CT]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>afad</td>
<td></td>
</tr>
<tr>
<td>@ 2561</td>
<td>[IF] [DA] [EX] [WB] [CT]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>aead+</td>
<td></td>
</tr>
<tr>
<td>@ 2562</td>
<td>[IF] [DA] [EX] [WB] [CT]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>af/ad+</td>
<td></td>
</tr>
<tr>
<td>@ 2563</td>
<td>[IF] [DA] [EX] [WB] [CT]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>af/ad</td>
<td></td>
</tr>
<tr>
<td>@ 2564</td>
<td>[IF] [DA] [EX] [WB] [CT]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>af/ad</td>
<td></td>
</tr>
<tr>
<td>@ 2565</td>
<td>[IF] [DA] [EX] [WB] [CT]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>af/ad</td>
<td></td>
</tr>
<tr>
<td>@ 2566</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@ 2567</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@ 2568</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@ 2569</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@ 2570</td>
<td>ag = '0x004001f0: lw r2,0(r4)'</td>
<td></td>
</tr>
<tr>
<td>@ 2571</td>
<td>[IF] [DA] [EX] [WB] [CT]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ag*</td>
<td></td>
</tr>
<tr>
<td>@ 2572</td>
<td>ah = '0x004001f0: [internal ld/st]'</td>
<td></td>
</tr>
<tr>
<td>@ 2573</td>
<td>ai = '0x004001f0: lw r6,8(r4)'</td>
<td></td>
</tr>
<tr>
<td>@ 2574</td>
<td>[IF] [DA] [EX] [WB] [CT]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ah</td>
<td></td>
</tr>
<tr>
<td>@ 2575</td>
<td>ah</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Extract from a pipeline trace
For example in the pipeline trace from Table 2.2, the state of the pipeline is given for each cycle (the cycle counter is the number after the @ sign). sim-outorder would add 10 cycles to the instruction 0x004001f0 and one to 0x004001f8.

With this method, if you add the counters of all instructions you obtain the execution time in cycles of the whole program. But the counters don’t give a pertinent information for a particular instruction, because the number of cycles that will be counted for this instruction depend on the instructions before (the number of cycles of cache misses happening just before this instruction will be added to this instruction). This problem induces an error in the number of cycles calculated by the script, but the error is marginal since only the errors of the first and the last instruction of the function are counted (a function will “lose” cycles if it finishes with an instruction taking a lot of cycles to be executed, and “gain” cycles if it is calling such a function).

5@ is the cycle counter, [IF] the instruction fetch stage, [DA] the decode, [EX] execution, [WB] writeback and [CT] the commit.
3 Results

3.1 Instruction class mix

Figure 3.1 shows the instruction class mix for each program (the mean is an arithmetic mean calculated with the number of instructions of each type for each program, the SPEC mean data have been taken from [8]). Integer operations are by far the most important operations: they account for at least 50% and up to 70% of all the instructions executed. Floating point operations are only present in 5 programs (epic, unepic, mpeg2dec, mpeg2enc and rasta) and in small proportions (less than 20%). The MediaBench programs perform, in proportion, much less store operations than SPEC programs.

3.2 Branch prediction hit rate

In the Figure 3.2 the branch prediction hit rate is shown (the mean is a geometric mean). Branch prediction hit rate for the MediaBench programs is not very high, with a mean of 90% and a minimum of 76.9% for mpeg2enc. This can be explained by a big proportion of data dependent branches. The only difference for the branch predictor between the default architecture and the StrongARM-like one is that the default has a bimodal predictor table which is 4 times bigger. This bigger table brings only a very small improvement (∼ +0.15%).

3.3 Instruction cache hit rate

The Figure 3.3 shows the instruction cache hit rate (only the hit rate for the L1 instruction cache for the default processor). Both instruction caches are "8" KByte (see 2.3) and we can see that the direct-mapped cache of the default processor has, as expected, lower hit rates than the set-associative one from the StrongARM-like processor.
Figure 3.1: Instruction class mix

Figure 3.2: Branch prediction hit rate
3.4 Data cache hit rate

The Figure 3.4 shows the data cache hit rate of the L1 cache. There aren’t big differences between the two architectures. The hit rates are low for Pegwit (smaller than 90%) and unepic (~95%), but quite high for the other programs.

3.5 Memory footprint

The program code size is shown in Figure 3.5. The Figure 3.6 gives the data part of the static memory consumption (the value displayed is the value of ld_data_size given by the simulator). This value can also be obtained through nm\(^1\) by adding the size of the symbols. The rasta program has a huge static structure (mapping_param) which is approximately 8 MByte big. The Figure 3.7 gives the size of the memory pages allocated.

The values returned by SimpleScalar must be taken with great care: SimpleScalar does not decrement its counter of memory pages allocated when a page is freed and no pages are allocated for the bss and sbss sections at the

\(^1\)nm is part of the GNU binary utilities and lists symbols from an object file
start² (see Appendix D on the lines 549 to 553: no mem_bcopy is being made and therefore the memory pages counter is not incremented, as opposed to sdata on line 542 for example). When the data from the bss or sbss sections are used the number of pages allocated is increased. Therefore it is not really easy to have the memory footprint of the applications:

1. SimpleScalar’s output gives static and dynamic information. The size of program text (code), the size of the .data and .bss sections for the static part, and the total size of pages allocated for the dynamic part.

2. The total size of pages allocated is greater than what it should be because the counter of pages allocated is not decremented when pages are freed

3. Referenced data from the .bss section is counted in the total size of pages allocated

²The bss section is used for local common variable storage. You may be allocat address space in the bss section, but you may not dictate data to load into it before your program executes. When a program starts running, all the contents of the bss section are zeroed bytes.
For handheld devices as for embedded systems, it is important to know before running a program if there is enough free memory to run it. It is therefore important to have a memory footprint representing the case where all the data from the .bss section is referenced. But adding the size of the pages allocated and the program data size gives a wrong result because the referenced data from the .bss section is counted twice. Therefore the sum of the total size of pages allocated and the program data size gives an overestimation of the memory footprint.

3.6 Function breakdown

Figure 3.8 and 3.9 show the breakdown of the different functions in terms of cycles and Figure 3.10 the breakdown of the functions in terms of instructions. The functions are ordered, on the left the functions which use the most cycles and the black part on the right is the sum of all the remaining functions. Function names are given under the bar. These graphics are based on the output of the cycle_by_function.pl post-processing script. By comparing the weight of a function on the 2 figures, it can be seen whether a function benefits from having more ALUs and more cache. For example, the collapse_pyr function from unepic takes 33% of the program execution.
Figure 3.6: Program data size\textsuperscript{a} (in bytes)

\textsuperscript{a}size of the .data and .bss sections

Figure 3.7: Total memory pages allocated (in bytes)
time with the StrongARM processor and less than 25% with the other processor. For the Rasta program, the most important function is \_printf. In a handheld device the output of such a program would not be written on the standard output or in a file but rather sent to another program. Therefore, the source code should be modified to remove the calls to printf to have a better look of the “useful” functions.

In the tables 3.1 and 3.2 the Instructions per cycle (IPC) are given for each program and for the most important function of each program. The speed increase factor is the ratio between the IPC for the default architecture and the StrongARM-like one. Looking at the speed factor, we can see that some programs really benefit from the additional ALUs\textsuperscript{3} like PGP and Pegwit, whereas for ADPCM the benefit is much smaller. To better understand the speed increase the data flow diagrams should be analyzed. For PGP, the IPC increase factor is greater than four when decoding and this is also the case for the mp\_smul function for encoding and decoding. This is greater than the increase in integer ALUs and is the consequence of the presence of an L2 cache with a latency of 6 cycles for the default architecture. This latency is smaller than the latency of 8 cycles of the StrongARM-like architecture and therefore the default architecture needs to wait two cycles less when there is a L1 cache miss and the data is present in L2.

\textsuperscript{3}four integer and FP ALUs instead of one for the StrongARM-like architecture
Figure 3.8: Function breakdown (in cycles) for the StrongARM architecture
Figure 3.9: Function breakdown (in cycles) for the default architecture.
Figure 3.10: Function breakdown (in instructions)
<table>
<thead>
<tr>
<th>Program</th>
<th>IPC (StrongARM)</th>
<th>IPC (default)</th>
<th>IPC increase factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adpcm decode</td>
<td>0.59</td>
<td>1.45</td>
<td>2.44</td>
</tr>
<tr>
<td>Adpcm encode</td>
<td>0.60</td>
<td>1.43</td>
<td>2.39</td>
</tr>
<tr>
<td>Epic epic</td>
<td>0.54</td>
<td>1.61</td>
<td>2.97</td>
</tr>
<tr>
<td>Epic unepic</td>
<td>0.48</td>
<td>1.38</td>
<td>2.85</td>
</tr>
<tr>
<td>g721 decode</td>
<td>0.60</td>
<td>1.76</td>
<td>2.91</td>
</tr>
<tr>
<td>g721 encode</td>
<td>0.61</td>
<td>1.94</td>
<td>3.20</td>
</tr>
<tr>
<td>gsm decode</td>
<td>0.59</td>
<td>1.79</td>
<td>3.04</td>
</tr>
<tr>
<td>gsm encode</td>
<td>0.54</td>
<td>2.00</td>
<td>3.68</td>
</tr>
<tr>
<td>jpeg decode</td>
<td>0.58</td>
<td>2.12</td>
<td>3.66</td>
</tr>
<tr>
<td>jpeg encode</td>
<td>0.57</td>
<td>1.97</td>
<td>3.45</td>
</tr>
<tr>
<td>mpeg2 decode</td>
<td>0.54</td>
<td>1.72</td>
<td>3.16</td>
</tr>
<tr>
<td>mpeg2 encode</td>
<td>0.55</td>
<td>1.42</td>
<td>2.59</td>
</tr>
<tr>
<td>Pegwit decode</td>
<td>0.43</td>
<td>1.63</td>
<td>3.76</td>
</tr>
<tr>
<td>Pegwit encode</td>
<td>0.43</td>
<td>1.60</td>
<td>3.71</td>
</tr>
<tr>
<td>Pgp decode</td>
<td>0.57</td>
<td>2.34</td>
<td>4.11</td>
</tr>
<tr>
<td>Pgp encode</td>
<td>0.56</td>
<td>2.24</td>
<td>3.97</td>
</tr>
<tr>
<td>Rasta</td>
<td>0.40</td>
<td>1.31</td>
<td>3.25</td>
</tr>
</tbody>
</table>

Table 3.1: Instruction per cycle for the programs
<table>
<thead>
<tr>
<th>Function name</th>
<th>IPC (StrongARM)</th>
<th>IPC (default)</th>
<th>IPC increase factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm decode</td>
<td>0.59</td>
<td>1.45</td>
<td>2.44</td>
</tr>
<tr>
<td>adpcm encode</td>
<td>0.60</td>
<td>1.43</td>
<td>2.39</td>
</tr>
<tr>
<td>internal_filter</td>
<td>0.55</td>
<td>1.68</td>
<td>3.05</td>
</tr>
<tr>
<td>collapse_pyr</td>
<td>0.52</td>
<td>2.03</td>
<td>3.94</td>
</tr>
<tr>
<td>quan</td>
<td>0.62</td>
<td>2.34</td>
<td>3.75</td>
</tr>
<tr>
<td>quan</td>
<td>0.62</td>
<td>2.31</td>
<td>3.70</td>
</tr>
<tr>
<td>Short_term_synthesis_filtering</td>
<td>0.62</td>
<td>1.82</td>
<td>2.94</td>
</tr>
<tr>
<td>Calculation_of_the_LTP_parameters</td>
<td>0.54</td>
<td>2.11</td>
<td>3.95</td>
</tr>
<tr>
<td>jpeg_idct_islow</td>
<td>0.60</td>
<td>2.20</td>
<td>3.64</td>
</tr>
<tr>
<td>encode_mcu_AC_refine</td>
<td>0.57</td>
<td>2.00</td>
<td>3.49</td>
</tr>
<tr>
<td>mpeg2_idctcol</td>
<td>0.54</td>
<td>1.65</td>
<td>3.06</td>
</tr>
<tr>
<td>dist1</td>
<td>0.55</td>
<td>1.34</td>
<td>2.43</td>
</tr>
<tr>
<td>gfAddMul</td>
<td>0.38</td>
<td>1.31</td>
<td>3.44</td>
</tr>
<tr>
<td>gfAddMul</td>
<td>0.37</td>
<td>1.27</td>
<td>3.43</td>
</tr>
<tr>
<td>mp_smul</td>
<td>0.57</td>
<td>2.38</td>
<td>4.20</td>
</tr>
<tr>
<td>mp_smul</td>
<td>0.57</td>
<td>2.38</td>
<td>4.20</td>
</tr>
<tr>
<td>__printf_fp</td>
<td>0.23</td>
<td>0.86</td>
<td>3.71</td>
</tr>
</tbody>
</table>

Table 3.2: Instruction per cycle for the most important functions
Conclusion

The goals of this project have been achieved:

- the MediaBench benchmark suite was ported to SimpleScalar
- MediaBench was analyzed with several SimpleScalar simulators
- the programs were simulated with two different processors architectures: a general purpose architecture and a StrongARM-like one
- the different outputs given by SimpleScalar have been analyzed to check their accuracy
- the runtime intensive functions have been outlined for each program

The simulations showed the great importance of integer operations in multimedia applications. This lets foresee a possibility for SIMD parallelism. The simulation with two different architectures showed the effects of different number of ALUs, cache sizes and organisation. The different results of this study confirm the results of [8], and gives a better understanding of the SimpleScalar output regarding memory utilization. The programs have also been analyzed more precisely, by looking at the function breakdown to find out the runtime intensive functions of each program.

The source code of the runtime intensive functions should be analyzed to find the algorithms used in multimedia programs, and their data flow diagrams to see how the performance can be enhanced by adapting the processor architecture to the needs of the program.
Bibliography


Appendix A
Processor configuration file for SimpleScalar

# SimpleScalar Processor Configuration File
# inspired by STRONGARM 1100 and others
# M. Platzner
# Sept 15, 2000

###############################################################
# SIMULATOR
# load configuration from a file
# -config
# dump configuration to a file
# -dumpconfig
# print help message
# -h false
# verbose operation
# -v false
# enable debug message
# -d false
# start in Dlilte debugger
# -i false
# random number generator seed (0 for timer seed)
# -seed 1
# initialize and terminate immediately
# -q false
# restore EIO trace execution from <fname>
# -chkpt <null>

32
# redirect simulator output to file (non-interactive only)
# -redir:sim <null>

# redirect simulated program output to file
# -redir:prog <null>

# simulator scheduling priority
# -nice 19

# maximum number of inst's to execute
# -max:inst 1000000

# number of insts skipped before timing starts
# -fastfwd 0

# generate pipetrace, i.e., <fname|stdout|stderr> <range>
# -ptrace <null>

# profile stat(s) against text addr's (mult uses ok)
# -pcstat <null>

# operate in backward-compatible bugs mode (for testing only)
# -bugcompat false

#******************************************************************************
#
# PROCESSOR CORE
#
#******************************************************************************

# instruction fetch queue size (in insts)
# -fetch:ifqsize 1

# extra branch mis-prediction latency
# -fetch:mplat 1

# speed of front-end of machine relative to execution core
# -fetch:speed 1

# instruction decode B/W (insts/cycle)
# -decode:width 1

# instruction issue B/W (insts/cycle)
# -issue:width 1

# run pipeline with in-order issue
# -issue:inorder true

# issue instructions down wrong execution paths
# -issue:wrongpath true

# register update unit (RUU) size
# -ruu:size 2

# load/store queue (LSQ) size
# -lsq:size 2

# instruction commit B/W (insts/cycle)
# -commit:width 1
# total number of integer ALU's available
-res:ialu 1

# total number of integer multiplier/dividers available
-res:imult 1

# total number of memory system ports available (to CPU)
-res:memport 1

# total number of floating point ALU's available
-res:fpalu 1

# total number of floating point multiplier/dividers available
-res:fpmult 1

# branch predictor type {nottaken|taken|perfect|bimod|2lev|comb}
-bpred bimod

# bimodal predictor config (<table size>)
-bpred:bimod 512

# 2-level predictor config (<l1size> <l2size> <hist_size> <xor>)
-bpred:2lev 1 1024 0 0

# combining predictor config (<meta_table_size>)
-bpred:comb 1024

# return address stack size (0 for no return stack)
-bpred:ras 8

# BTB config (<num_sets> <associativity>)
-bpred:btb 512 4

# speculative predictors update in {ID|WB} (default non-spec)
# -bpred:spec_update <null>

# l1 data cache config, i.e., {<config>|none}
-cache:d1
   d1:16:32:32:f

# l1 data cache hit latency (in cycles)
-cache:d1lat 1

# l2 data cache config, i.e., {<config>|none}
-cache:d12 none
# l2 data cache hit latency (in cycles)
-cache:dl2lat 8

# l1 inst cache config, i.e., {<config>|d1l|d1l2|none}
-cache:il1 il1:16:32:32:f

# l1 instruction cache hit latency (in cycles)
-cache:il1lat 1

# l2 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2 none

# l2 instruction cache hit latency (in cycles)
-cache:il2lat 8

# flush caches on system calls
-cache:flush false

# convert 64-bit inst addresses to 32-bit inst equivalents
-cache:icompress false

# memory access latency (<first_chunk> <inter_chunk>)
-mem:lat 8 2

# memory access bus width (in bytes)
-mem:width 4

# instruction TLB config, i.e., {<config>|none}
-tlb:itlb itlb:32:4096:4:f

# data TLB config, i.e., {<config>|none}
-tlb:dtlb dtlb:32:4096:4:f

# inst/data TLB miss latency (in cycles)
-tlb:lat 30
Appendix B
Automation scripts

B.1 benchmark.sh

#!/bin/sh
if [ $# -lt 1 ]; then
    echo >&2 Usage: $0 \[all\|adpcm\|epic\|g721\|gsm\|jpeg\|mpeg2\|pegwit\|pgp\|rasta\]
    exit 1
fi

BENCH_HOME=/home/guthomas/simple/applications/mediabench
CYCLE_SCRIPT=/home/guthomas/src/cycle_by_function.pl
EXTENSION='date '+%d%m%y_%H%M''
PROC_PARAM='-nice 19 -pcstat sim_num_insn'
INSN_SCRIPT=/home/guthomas/src/insn_by_function.pl

export BENCH_HOME
export CYCLE_SCRIPT
export EXTENSION
export PROC_PARAM
export INSN_SCRIPT

if [ $1 = 'all' ]; then
    ./mpeg2_benchmark.sh &
    ./adpcm_benchmark.sh &
    ./epic_benchmark.sh
    ./g721_benchmark.sh &
    ./gsm_benchmark.sh
    ./jpeg_benchmark.sh &
    ./pegwit_benchmark.sh
    ./pgp_benchmark.sh &
    ./rasta_benchmark.sh
    exit 0;
else
    ./$1_benchmark.sh
    exit 0;
fi
B.2 An example for a specific benchmark: adpcm_benchmark.sh

```bash
#!/bin/sh

echo adpcm Benchmark

BENCH_PATH=$BENCH_HOME/adpcm/simple

BENCH_NAME=rawcaudio # the name of the executable file
sim-profile -iclass -iprof -redir:sim $BENCH_NAME.simple.profile.$EXTENSION \ 
$BENCH_PATH/bin/$BENCH_NAME < $BENCH_PATH/data/clinton.pcm \ 
> $BENCH_PATH/data/out.adpcm.$EXTENSION

sim-outorder $PROC_PARAM -pcstat sim_cycle \ 
-redirect:sim $BENCH_NAME.simple.cycle.$EXTENSION $BENCH_PATH/bin/$BENCH_NAME \ 
< $BENCH_PATH/data/clinton.pcm > $BENCH_PATH/results/out.adpcm.$EXTENSION

$CYCLE_SCRIPT $BENCH_NAME.simple.cycle.$EXTENSION \ 
$BENCH_PATH/bin/$BENCH_NAME.dis \ 
>$BENCH_NAME.simple.cycle.function.$EXTENSION

$insn_script $BENCH_NAME.simple.profile.$EXTENSION \ 
$BENCH_PATH/bin/$BENCH_NAME.dis \ 
>$BENCH_NAME.simple.profile.function.$EXTENSION &

BENCH_NAME=rawaudio # the name of the executable file

sim-profile -iclass -iprof -redir:sim $BENCH_NAME.simple.profile.$EXTENSION \ 
$BENCH_PATH/bin/$BENCH_NAME < $BENCH_PATH/data/clinton.adpcm \ 
> $BENCH_PATH/data/out.pcm.$EXTENSION

sim-outorder $PROC_PARAM -pcstat sim_cycle \ 
-redirect:sim $BENCH_NAME.simple.cycle.$EXTENSION $BENCH_PATH/bin/$BENCH_NAME \ 
< $BENCH_PATH/data/clinton.adpcm > $BENCH_PATH/results/out.pcm.$EXTENSION

$CYCLE_SCRIPT $BENCH_NAME.simple.cycle.$EXTENSION \ 
$BENCH_PATH/bin/$BENCH_NAME.dis \ 
>$BENCH_NAME.simple.cycle.function.$EXTENSION

$insn_script $BENCH_NAME.simple.profile.$EXTENSION \ 
$BENCH_PATH/bin/$BENCH_NAME.dis \ 
>$BENCH_NAME.simple.profile.function.$EXTENSION &
```
Appendix C
Postprocessing scripts

C.1 cycle_by_function.pl

#!/tiklopt1/bin/perl

# This source file is distributed "as is" in the hope that it will be
# useful. It is distributed with no warranty, and no author or
# distributor accepts any responsibility for the consequences of its
# use.
#
# Everyone is granted permission to copy, modify and redistribute
# this source file under the following conditions:
# This tool set is distributed for non-commercial use only.
# Please contact the maintainer for restrictions applying to
# commercial use of these tools.
# Permission is granted to anyone to make or distribute copies
# of this source code, either as received or modified, in any
# medium, provided that all copyright notices, permission and
# nonwarranty notices are preserved, and that the distributor
# grants the recipient permission for further redistribution as
# permitted by this document.
# Permission is granted to distribute this file in compiled
# or executable form under the same conditions that apply for
# source code, provided that either:
# A. it is accompanied by the corresponding machine-readable
# source code,
# B. it is accompanied by a written offer, with no time limit,
# to give anyone a machine-readable copy of the corresponding
# source code in return for reimbursement of the cost of
# distribution. This written offer must permit verbatim
# duplication by anyone, or
# C. it is distributed by someone who received only the
# executable form, and is accompanied by a copy of the
# written offer of source code that they received concurrently.
# In other words, you are welcome to use, share and improve this
# source file. You are forbidden to forbid anyone else to use, share
# and improve what you give them.
# parse commands
if (@ARGV != 2)
{
  print STDERR
  "Usage: cycle_by_function.pl <simulator_output> <disassembly_file>\n".
  "\n".
  " where <disassembly_file> is a disassembly file, generated with\n".
  " the command \"objdump -d <binary>\", <simulator_output> is the\n".
  " sim-outorder output containing a text-based profile for pcstat sim_cycle.\n".
  "\n".
  " Example usage:\n".
  "\n".
  " objdump -d test-math >! test-math.dis\n".
  " sim-outorder -pcstat sim_cycle test-math >! test-math.out\n".
  " cycle_by_function.pl test-math.out test-math.dis\n".
  "\n";
  exit -1;
}
$output_filename = shift @ARGV;
open(SIM_OUTPUT, $output_filename) || die "$output_filename could not be opened\n";
$dis_filename = shift @ARGV;
open(DIS, $dis_filename) || die "$dis_filename could not be opened\n";
sub add_cycles_to_function
{
  $ins_address = shift;
  $count = shift;

  while (<DIS>) {
    if (/^00$ins_address\s<([\[\]+].*>/) {
      $func_count{$1} += $count;
      $global_counter += $count;
      return;
    }
  }
  seek(DIS,0,0)
}

while (<SIM_OUTPUT>) {
  if (/sim_cycle_by_pc.start_dist/) {
    last;
  }
}

while (<SIM_OUTPUT>) {
  if (/0x([0-9a-fA-F]{6})([0-9a-fA-F]+)/) {
    add_cycles_to_function($1,hex($2));
  } else {
    if (/sim_cycle_by_pc.end_dist/) {
      last;
    }
  }
}

print "Total of cycles: $global_counter\n";

for (sort {$func_count{$b} <=> $func_count{$a}} keys %func_count) {
  print "$_ : $func_count{$_} cycles ",100*$func_count{$_}/$global_counter,"\n";
}
C.2  insn_by_function.pl

#!/tik1opt1/bin/perl

# This source file is distributed "as is" in the hope that it will be
# useful. It is distributed with no warranty, and no author or
# distributor accepts any responsibility for the consequences of its
# use.
#
# Everyone is granted permission to copy, modify and redistribute
# this source file under the following conditions:
#
# This tool set is distributed for non-commercial use only.
# Please contact the maintainer for restrictions applying to
# commercial use of these tools.
#
# Permission is granted to anyone to make or distribute copies
# of this source code, either as received or modified, in any
# medium, provided that all copyright notices, permission and
# nonwarranty notices are preserved, and that the distributor
# grants the recipient permission for further redistribution as
# permitted by this document.
#
# Permission is granted to distribute this file in compiled
# or executable form under the same conditions that apply for
# source code, provided that either:
# A. it is accompanied by the corresponding machine-readable
#    source code,
# B. it is accompanied by a written offer, with no time limit,
#    to give anyone a machine-readable copy of the corresponding
#    source code in return for reimbursement of the cost of
#    distribution. This written offer must permit verbatim
#    duplication by anyone, or
# C. it is distributed by someone who received only the
#    executable form, and is accompanied by a copy of the
#    written offer of source code that they received concurrently.
#
# In other words, you are welcome to use, share and improve this
# source file. You are forbidden to forbid anyone else to use, share
# and improve what you give them.
#
# Parse commands

if (@ARGV != 2)
{
    print STDERR
    "Usage: insn_by_function.pl <simulator_output> <disassembly_file>\n".
    "\n".
    " Example usage:\n".
    " \n".
    " objdump -d test-math >\! test-math.dis\n".
    " sim-outorder -pcstat sim_num_insn test-math >\! test-math.out\n".
    " insn_by_function.pl test-math.out test-math.dis\n".
    "\n".
    exit -1;
}
$output_filename = shift @ARGV;
open(SIM_OUTPUT, $output_filename) || die "$output_filename could not be opened\n";

$dis_filename = shift @ARGV;
open(DIS, $dis_filename) || die "$dis_filename could not be opened\n";

sub add_cycles_to_function {
    $ins_address = shift;
    $count = shift;

    while (<DIS>) {
        if (/^00$ins_address\s<([\[\]>]+).*>/) {
            $func_count{$1} += $count;
            $global_counter += $count;
            return;
        }
    }
    seek(DIS,0,0)
}

while (<SIM_OUTPUT>) {
    if (/sim_num_insn_by_pc.start_dist/) {
        last;
    }
}

while (<SIM_OUTPUT>) {
    if (/0x([0-9a-fA-F][0-9a-fA-F]+)/) {
        add_cycles_to_function($1,hex($2));
    } else {
        if (/sim_num_insn_by_pc.end_dist/) {
            last;
        }
    }
}

print "Total of instructions: $global_counter\n";

for (sort @{$func_count} <=> @{$func_count} keys %func_count) {
    print "$_ : @{$func_count}{$_} instructions ",100*$func_count{$_}/$global_counter,"\n";
}
Appendix D
Extract form loader.c

Here is an extract (lines 476 to 555) from the file loader.c from the SimpleScalar source code. This code is responsible for loading the executable in the SimpleScalar memory.

```c
switch (shdr.s_flags) {
    case ECOFF_STYP_TEXT:
        ld_text_size = ((shdr.s_vaddr + shdr.s_size) - MD_TEXT_BASE) + TEXT_TAIL_PADDING;
        p = calloc(shdr.s_size, sizeof(char));
        if (!p) fatal("out of virtual memory");
        if (fseek(fobj, shdr.s_scnptr, 0) == -1) fatal("could not read '.text' from executable", i);
        if (fread(p, shdr.s_size, 1, fobj) < 1) fatal("could not read text section from executable");
        /* copy program section into simulator target memory */
        mem_bcopy(mem_access, mem, Write, shdr.s_vaddr, p, shdr.s_size);
        /* create tail padding and copy into simulator target memory */
        mem_bzero(mem_access, mem,
                   shdr.s_vaddr + shdr.s_size, TEXT_TAIL_PADDING);
        /* release the section buffer */
        free(p);
    #if 0
        Text_seek = shdr.s_scnptr;
        Text_start = shdr.s_vaddr;
        Text_size = shdr.s_size / 4;
        /* there is a null routine after the supposed end of text */
        Text_size += 10;
        Text_end = Text_start + Text_size * 4;
        /* create_text_reloc(shdr.s_relptr, shdr.s_nreloc); */
    #endif
    break;
    case ECOFF_STYP_RDATA:
        /* The .rdata section is sometimes placed before the text */
        /* section instead of being contiguous with the .data section.
        */
```

```c
#define Rdata_start = shdr.s_vaddr;
#define Rdata_size = shdr.s_size;
#define Rdata_seek = shdr.s_scnptr;
#endif

/* fall through */
#endif

Data_seek = shdr.s_scnptr;
#endif

/* fall through */
#endif

Sdata_seek = shdr.s_scnptr;
#endif

p = calloc(shdr.s_size, sizeof(char));
if (!p)
    fatal("out of virtual memory");

if (fseek(fobj, shdr.s_scnptr, 0) == -1)
    fatal("could not read '.text' from executable", i);
if (fread(p, shdr.s_size, 1, fobj) < 1)
    fatal("could not read text section from executable");

/* copy program section into simulator target memory */
mem_bcopy(mem_access, mem, Write, shdr.s_vaddr, p, shdr.s_size);

/* release the section buffer */
free(p);

break;

case ECOFF_STYP_BSS:
break;

break;

} }
```