Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

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 analized 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 run-time intensive function were outlined in all the applications. The different behaviour depending on the processor architecture chosen was also analyzed.

Contents

List of Figures

List of Tables

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 execution. 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 example for the same program on the same unloaded machine sim-safe simulates 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 programs 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 applications are publicly available, making the suite available to a wider user community. 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 digital video transmission and is also used for DVDs. The important computing 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 fullrate speech trancoding, 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 functionaly correct.

2.1 Compilation of MediaBench

The compilation of the Sun binaries went without any problem, the provided Makefiles needed only to be adpated 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` .

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

```
CFLAGS="$(RSAiNCDIR) $(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 misssing 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 usefull 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 funtion 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
```

Benchmark name	Binary	Options	Input file
Adpcm decode	rawdaudio		< clinton.adpcm
Adpcm encode	rawcaudio		< clinton.pcm
Epic epic	epic	-o <output file name>	< test_image.pgm
Epic unepic	unepic	-o <output file name>	< test.image.pgm.E
g721 decode	decode	-4 -l -f clinton.g721	
g721 encode	encode	-4 -l -f clinton.pcm	
gsm decode	untoast	-fpl clinton.pcm.run.gsm	
gsm encode	toast	-fpl clinton.pcm	
jpeg decode	djpeg	-dct int -ppm -outfile <output file name>	testimg.jpg
jpeg encode	cjpeg	-dct int -progressive -opt -outfile <output file name>	testimg.ppm
MPEG2 decode	mpeg2dec	-r -f -b mei16v2.m2v -oO tmp%d	
MPEG2 encode[a]	mpeg2enc	options.par <output file name>	
Pegwit decode	pegwit	-d pegwint.enc <output file name>	< my.sec
Pegwit encode	pegwit	-e my.pub pgptest.plain <output file name>	< encryption_junk
PGP decode[b]	pgp	-fdb -zbillms	< pgptext.pgp
PGP encode[b]	pgp	-fes Bill -zbillms -u Bill	< pgptest.plain
Rasta	rasta	-z -A -J -S 8000 -n 12 -f map_weights.dat	< ex5_c1.wav

[a]The `options.par` file contains the options for encoding the MPEG2 stream and **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 `.pgp` in a home directory.
2. `pgp -ka billms_prv.pgp $(HOME)/.pgp/secring.pgp`
3. `pgp -ka billms_pub.pgp $(HOME)/.pgp/pubring.pgp`

Table 2.1: Command line parameters for the MediaBench programs

```
<statistic name>_by_pc.imax = xxxxxxxx
<statistic name>_by_pc.average = xxxxxxxxx
<statistic name>_by_pc.std_dev = xxxxxxxxx
<statistic name>_by_pc.overflows = x
# pdf == prob dist fn, cdf == cumulative dist fn
# index count pdf
<statistic name>_by_pc.start_dist
0xAAAAAACCCCCCCC xxxxxxxxxxxxxxxxxx x.xx
...
0xAAAAAACCCCCCCC xxxxxxxxxxxxxxxxxx x.xx
<statistic name>_by_pc.end_dist
```

AAAAAA is the text address and CCCCCCCC is the value of the counter in hexadecimal for this text address[3].

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_funtion* 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_funtion* parses the disassembly file to look to which function this text address belongs. The format of the disassembly file[4] 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 wich 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 fectched at cycle 1000 and instruction B is fetched at cycle 1002, the counter of B is incremented by 2.

[3]The counter is always written with 8 digits whereas some text addresses are shorter than 6 digits.

[4]The disassembly file is obtained with *objdump*:
`objdump -dl <executable file> > <disassembly file>`

```
@ 2560
ae = '0x004002e0: [internal ld/st]'
af = '0x004002e8: jal       0x4001f0'
 [IF]      [DA]      [EX]      [WB]      [CT]
  af        ad
            ae
@ 2561
 [IF]      [DA]      [EX]      [WB]      [CT]
            ae        ad+
            af/
@ 2562
 [IF]      [DA]      [EX]      [WB]      [CT]
            af/                ad
                               ae
@ 2563
 [IF]      [DA]      [EX]      [WB]      [CT]
                      af                 ad
                                         ae
@ 2564
 [IF]      [DA]      [EX]      [WB]      [CT]
                                af
@ 2565
 [IF]      [DA]      [EX]      [WB]      [CT]
                                         af
@ 2566
@ 2567
@ 2568
@ 2569
@ 2570
ag = '0x004001f0: lw        r2,0(r4)'
 [IF]      [DA]      [EX]      [WB]      [CT]
  ag*
@ 2571
ah = '0x004001f0: [internal ld/st]'
ai = '0x004001f8: lw        r5,8(r4)'
 [IF]      [DA]      [EX]      [WB]      [CT]
  ai        ag
            ah
@ 2572
aj = '0x004001f8: [internal ld/st]'
 [IF]      [DA]      [EX]      [WB]      [CT]
            ah        ag+
            ai
            aj
```

Table 2.2: Extract from a pipeline trace

For example in the pipeline trace from Table 2.2 [5], 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 "loose" 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 *mepg2enc*. 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 ($\sim$ +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.

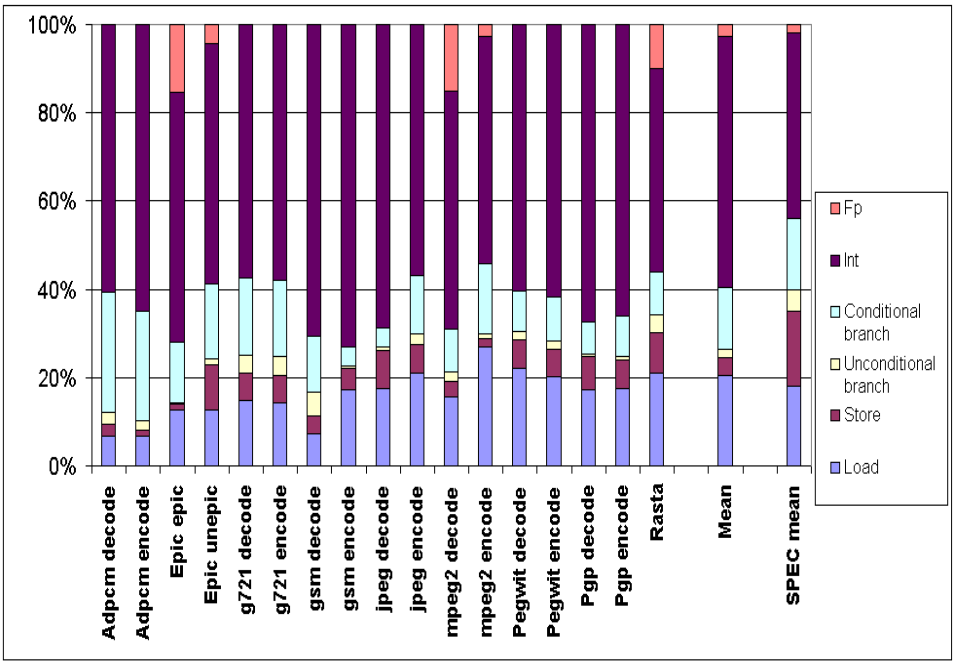

Figure 3.1: Instruction class mix

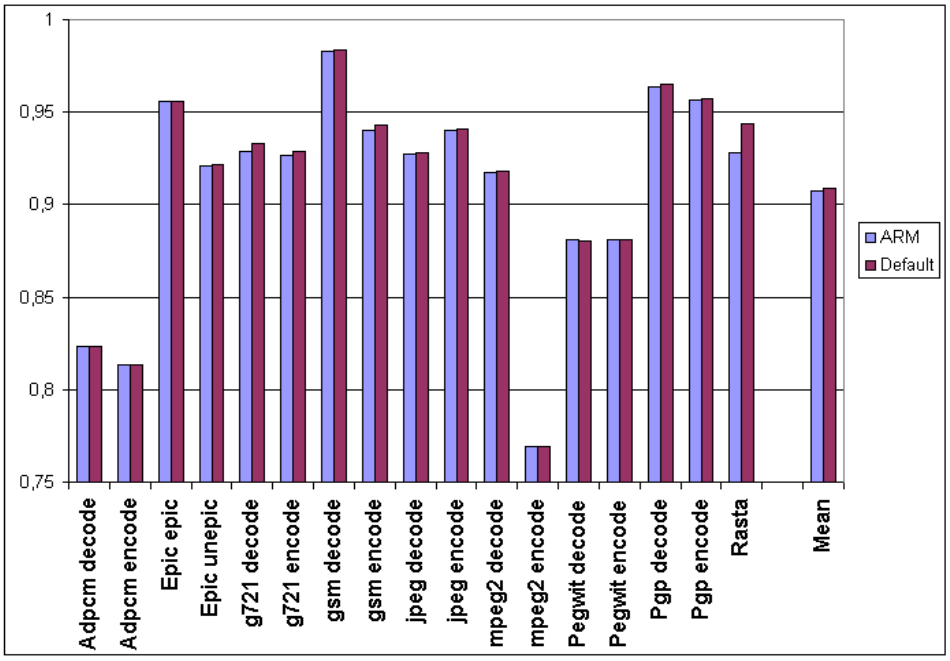

Figure 3.2: Branch prediction hit rate

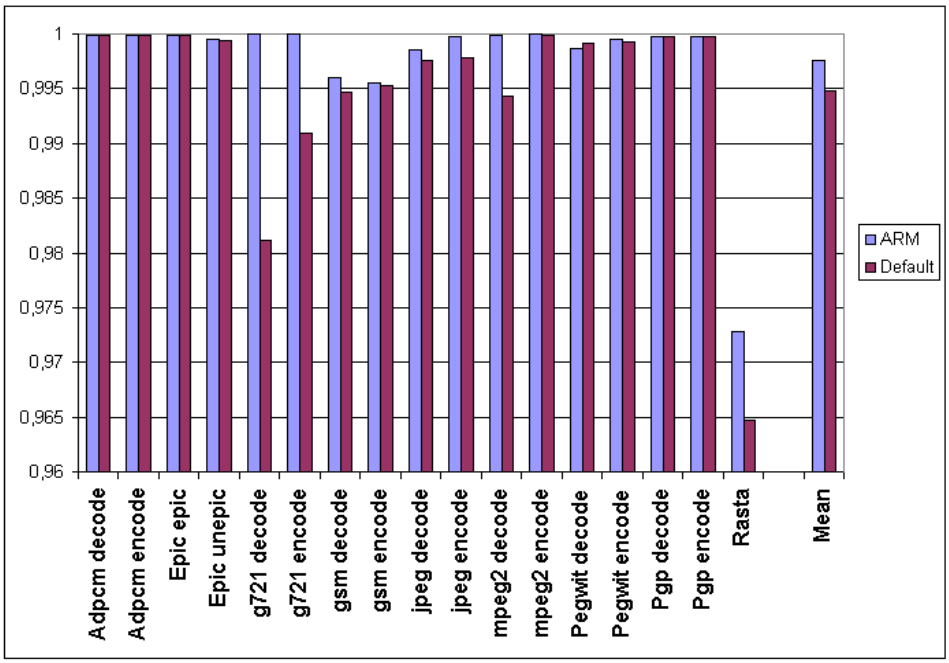

Figure 3.3: Instruction cache hit rate

3.4 Data cache hit rate

The Figure 3.4 shows the data cahce 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 aproximately 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

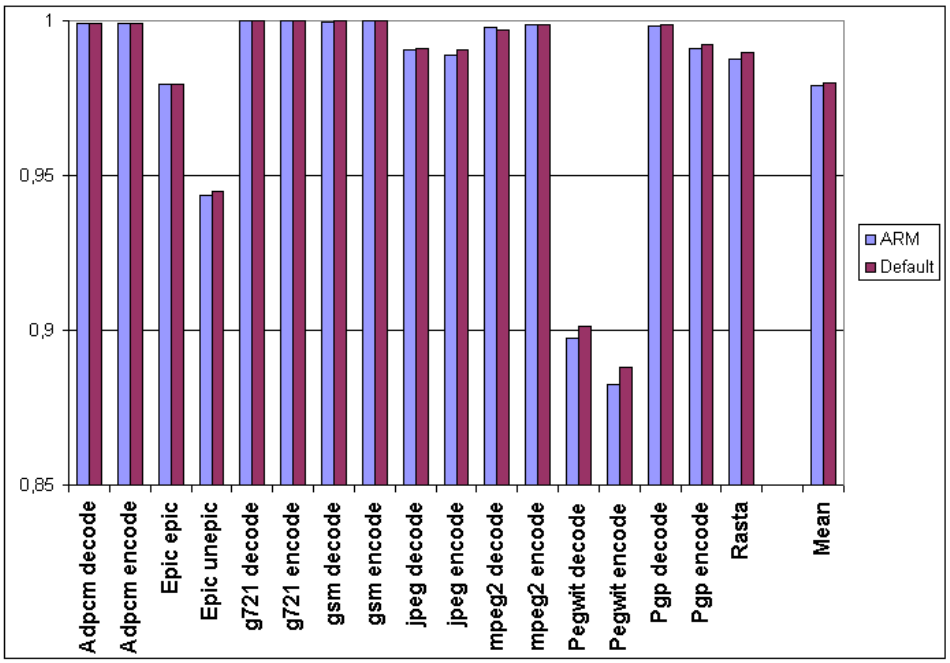

Figure 3.4: Data cache hit rate

start[2] (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

[2]The 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.

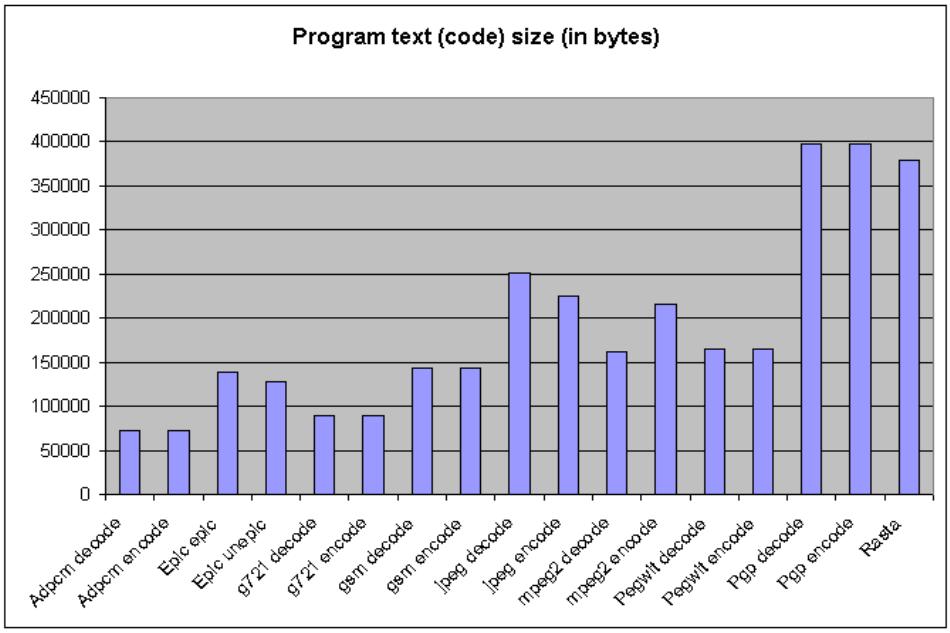

Figure 3.5: Program text (code) size (in bytes)

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_funtion.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

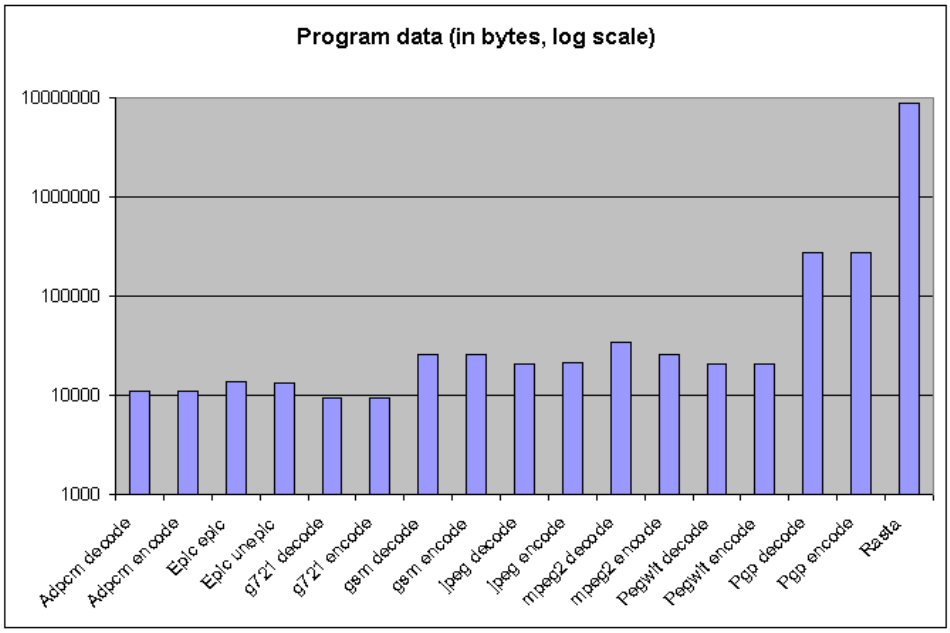

Figure 3.6: Program data size[a] (in bytes)

[a]size of the .data and .bss sections

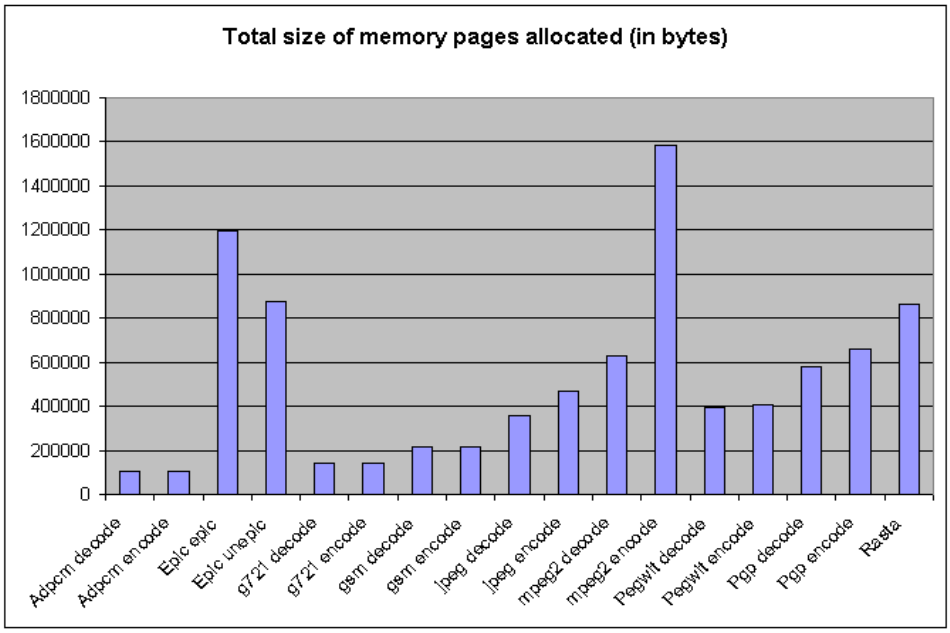

Figure 3.7: Total memory pages allocated (in bytes)

time with the StrongARM processor and less than 25% with the other processor. For the Rasta program, the most important function is *__printf_fp*. 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[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.

[3]four integer and FP ALUs instead of one for the StrongARM-like architecture

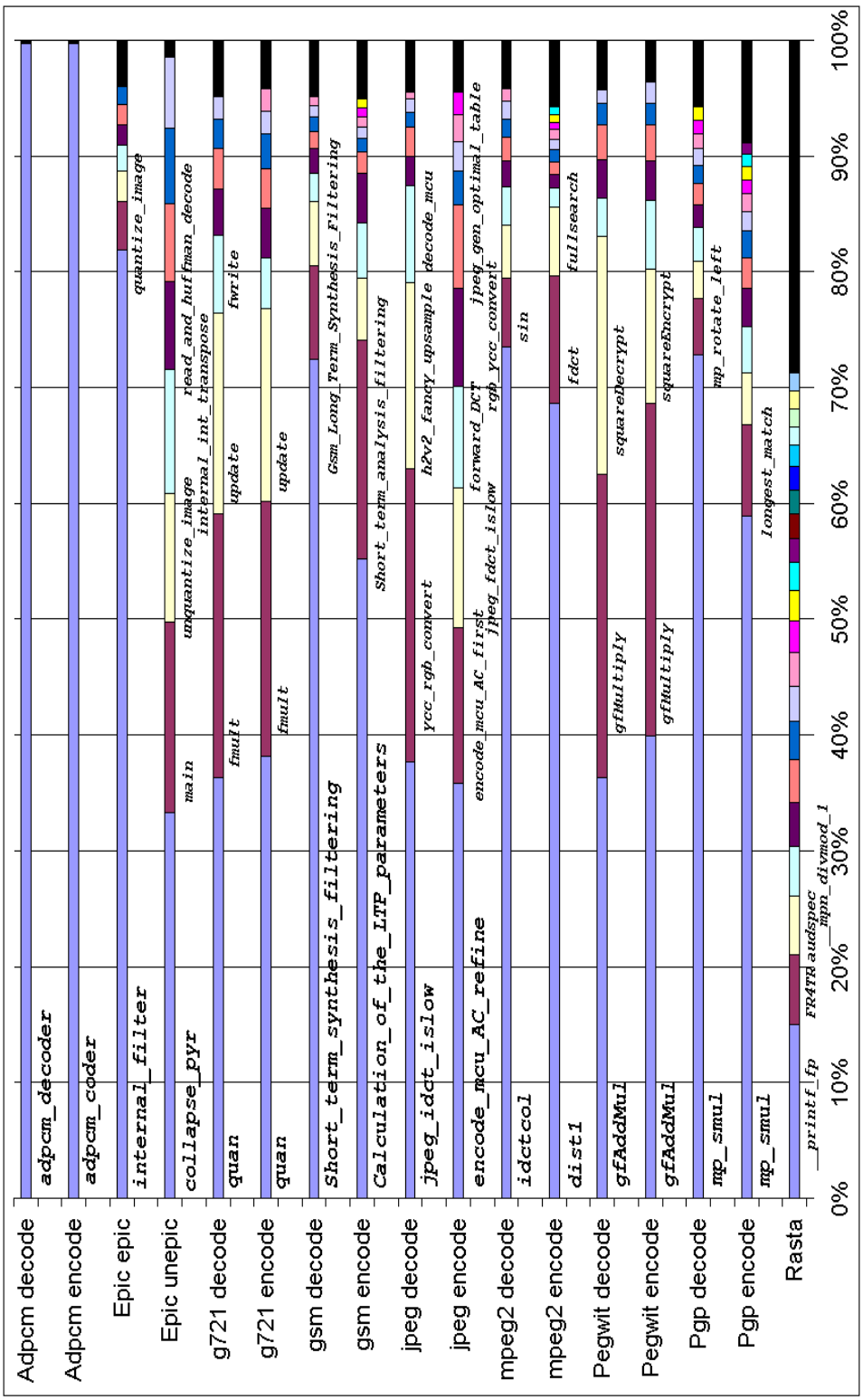

Figure 3.8: Function breakdown (in cycles) for the StrongARM architecture

Figure 3.9: Function breakdown (in cycles) for the default architecture

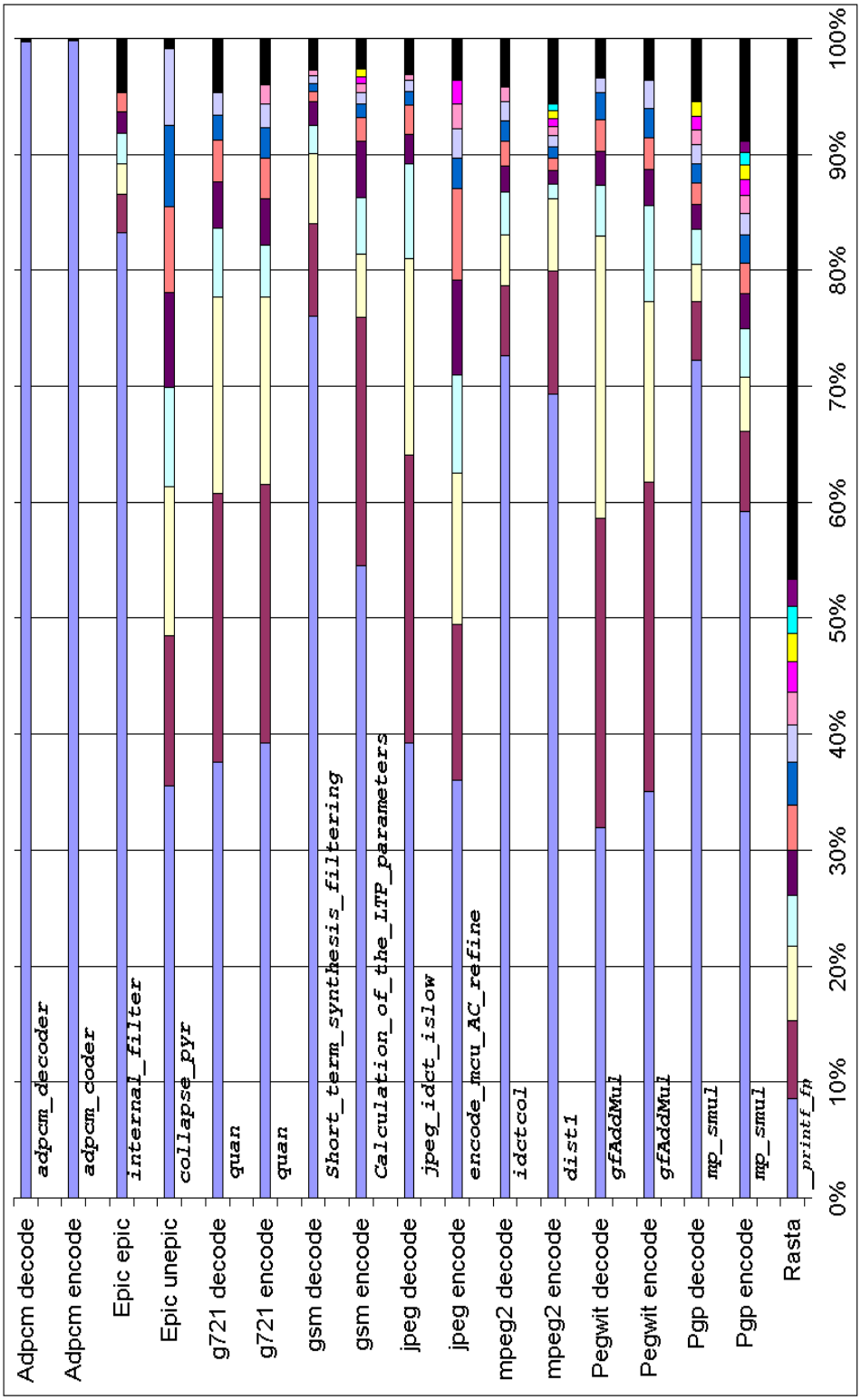

Figure 3.10: Function breakdown (in instructions)

	IPC (StrongARM)	IPC (default)	IPC increase factor
Adpcm decode	0.59	1.45	2.44
Adpcm encode	0.60	1.43	2.39
Epic epic	0.54	1.61	2.97
Epic unepic	0.48	1.38	2.85
g721 decode	0.60	1.76	2.91
g721 encode	0.61	1.94	3.20
gsm decode	0.59	1.79	3.04
gsm encode	0.54	2.00	3.68
jpeg decode	0.58	2.12	3.66
jpeg encode	0.57	1.97	3.45
mpeg2 decode	0.54	1.72	3.16
mpeg2 encode	0.55	1.42	2.59
Pegwit decode	0.43	1.63	3.76
Pegwit encode	0.43	1.60	3.71
Pgp decode	0.57	2.34	4.11
Pgp encode	0.56	2.24	3.97
Rasta	0.40	1.31	3.25

Table 3.1: Instruction per cycle for the programs

	Function name	IPC (StrongARM)	IPC (default)	IPC increase factor
Adpcm decode	adpcm_decoder	0.59	1.45	2.44
Adpcm encode	adpcm_coder	0.60	1.43	2.39
Epic epic	internal_filter	0.55	1.68	3.05
Epic unepic	collapse_pyr	0.52	2.03	3.94
g721 decode	quan	0.62	2.34	3.75
g721 encode	quan	0.62	2.31	3.70
gsm decode	Short_term_synthesis_filtering	0.62	1.82	2.94
gsm encode	Calculation_of_the_LTP_parameters	0.54	2.11	3.95
jpeg decode	jpeg_idct_islow	0.60	2.20	3.64
jpeg encode	encode_mcu_AC_refine	0.57	2.00	3.49
mpeg2 decode	idctcol	0.54	1.65	3.06
mpeg2 encode	dist1	0.55	1.34	2.43
Pegwit decode	gfAddMul	0.38	1.31	3.44
Pegwit encode	gfAddMul	0.37	1.27	3.43
Pgp decode	mp_smul	0.57	2.38	4.20
Pgp encode	mp_smul	0.57	2.38	4.20
Rasta	__printf_fp	0.23	0.86	3.71

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 possibilty 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 analized 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

[1] Todd M. Austin. A User's and Hacker's Guide to the SimpleScalar Architectural Research Tool Set, 1997.

[2] Doug Burger and Todd M. Austin. SimpleScalar Tutorial

[3] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical report, University of Winsconsin-Madison, Computer Science TR #1342, 1997.

[4] Charles Price. MIPS IV Instruction Set, revision 3.1. MIPS Technologies, Inc., Mountain View, CA, 1995

[5] Chunho Lee, Miodrag Potkonjak, and William H. Magione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the IEEE/ACM Int'l Symposium on Microarchitecture*, 1997.

[6] Standard Performance Evaluation Corporation.
`http://www.spec.org/`

[7] Intel® StrongARM* Processors.
`http://developer.intel.com/design/strong/quicklist/processor/`

[8] Benjamin Bishop, Thomas P. Kelliher, and Mary Jane Irwin. A Detailed Analysis of MediaBench. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, 1999.

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 Dlite 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>
```

```
# 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 PREDICTION
#
##########################################################################

# 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 8 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>



##########################################################################
#
#  MEMORY HIERARCHY
#
##########################################################################

# l1 data cache config, i.e., {<config>|none}
-cache:dl1             dl1:16:32:32:f

# l1 data cache hit latency (in cycles)
-cache:dl1lat                   1

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

```
# l2 data cache hit latency (in cycles)
-cache:dl2lat                    8

# l1 inst cache config, i.e., {<config>|dl1|dl2|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 1>&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
   ./rasta_benchmark.sh &
   ./pgp_benchmark.sh
   exit 0;
else
   ./$1_benchmark.sh
   exit 0;
fi
```

B.2 An example for a specific benchmark: adpcm_benchmark.sh

```
#!/bin/sh

echo adpcm Benchmark

BENCH_PATH=$BENCH_HOME/adpcm/simple
cd $BENCH_PATH/results

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 < $BENCH_PATH/data/clinton.pcm \
  > ../data/out.adpcm.$EXTENSION
sim-outorder $PROC_PARAM -pcstat sim_cycle \
  -redir: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=rawdaudio  # 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 \
  -redir: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

```
#!/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: 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\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".
"   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_num_insn.\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]{6})([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\n";

for (sort {$func_count{$b} <=> $func_count{$a}} 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.

```
476         switch (shdr.s_flags)
477           {
478           case ECOFF_STYP_TEXT:
479             ld_text_size = ((shdr.s_vaddr + shdr.s_size) - MD_TEXT_BASE)
480               + TEXT_TAIL_PADDING;
481
482             p = calloc(shdr.s_size, sizeof(char));
483             if (!p)
484               fatal("out of virtual memory");
485
486             if (fseek(fobj, shdr.s_scnptr, 0) == -1)
487               fatal("could not read `.text' from executable", i);
488             if (fread(p, shdr.s_size, 1, fobj) < 1)
489               fatal("could not read text section from executable");
490
491             /* copy program section into simulator target memory */
492             mem_bcopy(mem_access, mem, Write, shdr.s_vaddr, p, shdr.s_size);
493
494             /* create tail padding and copy into simulator target memory */
495             mem_bzero(mem_access, mem,
496                       shdr.s_vaddr + shdr.s_size, TEXT_TAIL_PADDING);
497
498             /* release the section buffer */
499             free(p);
500
501 #if 0
502             Text_seek = shdr.s_scnptr;
503             Text_start = shdr.s_vaddr;
504             Text_size = shdr.s_size / 4;
505             /* there is a null routine after the supposed end of text */
506             Text_size += 10;
507             Text_end = Text_start + Text_size * 4;
508             /* create_text_reloc(shdr.s_relptr, shdr.s_nreloc); */
509 #endif
510             break;
511
512           case ECOFF_STYP_RDATA:
513             /* The .rdata section is sometimes placed before the text
514              * section instead of being contiguous with the .data section.
515              */
```

```
516 #if 0
517             Rdata_start = shdr.s_vaddr;
518             Rdata_size = shdr.s_size;
519             Rdata_seek = shdr.s_scnptr;
520 #endif
521             /* fall through */
522           case ECOFF_STYP_DATA:
523 #if 0
524             Data_seek = shdr.s_scnptr;
525 #endif
526             /* fall through */
527           case ECOFF_STYP_SDATA:
528 #if 0
529             Sdata_seek = shdr.s_scnptr;
530 #endif
531
532             p = calloc(shdr.s_size, sizeof(char));
533             if (!p)
534               fatal("out of virtual memory");
535
536             if (fseek(fobj, shdr.s_scnptr, 0) == -1)
537               fatal("could not read `.text' from executable", i);
538             if (fread(p, shdr.s_size, 1, fobj) < 1)
539               fatal("could not read text section from executable");
540
541             /* copy program section it into simulator target memory */
542             mem_bcopy(mem_access, mem, Write, shdr.s_vaddr, p, shdr.s_size);
543
544             /* release the section buffer */
545             free(p);
546
547             break;
548
549           case ECOFF_STYP_BSS:
550             break;
551
552           case ECOFF_STYP_SBSS:
553             break;
554           }
555       }
```