EMULATORS BASED ENVIRONMENT FOR HW/SW PARTITIONING

Mirsad Halimic (Panasonic Broadband Communications Development Laboratory, Wokingham, Berkshire, United Kingdom; Mirsad.Halimic@eu.panasonic.com)
Adnan Al-Adnani (Panasonic Broadband Communications Development Laboratory, Wokingham, Berkshire, United Kingdom; Adnan@eu.panasonic.com)
Yasuo Harada (Panasonic Broadband Communications Development Laboratory, Wokingham, Berkshire, United Kingdom; Yasuo.Harada@jp.panasonic.com)
Dick Arvind (University of Edinburgh, Edinburgh, United Kingdom; dka@dcs.ed.ac.uk)
Janek Mann (University of Edinburgh, Edinburgh, United Kingdom; janekm@gmail.com)

ABSTRACT

This paper describes two parts of the environment termed a Partitioner. The Partitioner is utilised for partitioning a functional designs (software modules) onto heterogeneous hardware platforms. In this study the heterogeneous platform is presented by three different commercially available development environments representing a RISC processor, DSP and FPGA. The Partitioner is designed to take a set of basic code modules and in the process of reconfiguring an SDR based terminal find a partitioning (assignment to different processing units) and schedule for these modules that will optimised the performance of the unit.

Parts of the Partitioner presented here are a Threading analyser, which analyses the threadability of a module of non-threaded code and a tool that performs analysis of the power consumption on each of the available platforms.

1. INTRODUCTION

Mobile computing in the future will demand processing platforms that provide high-performance computation at low power consumption and which support multi-standard wireless protocols. One of the enablers for fulfilling these requirements is an entity that maps software onto heterogeneous hardware platform in an optimal way. In this study that entity is termed the Partitioner.

The Partitioner is designed to take a set of basic code modules and in the process of reconfiguring an SDR based terminal find a partitioning (assignment to different processing units) and schedule for these modules that will optimised the performance of the unit. There is a GUI that allows entering the information on all the software (functional) modules, their dependencies and different mappings. Several different mappings, even onto the same hardware part, can take place for each module in the design. The GUI also allows hardware and design parameters to be set, such as the clock speed, for each hardware module.

Respective hardware platforms are represented by their development environments, which provide support for running a simulation of the partitioned code on the vendor-supplied simulators. The RISC processor platform is represented by the Armulator, a cycle-accurate simulator for the ARM cores available from ARM Ltd. The DSP platform is represented by VisualDSP++, the integrated development environment from Analog Devices for a cycle-accurate simulator for the TigerSHARC processor. The FPGA platform was represented by ModelSim, Xilinx simulation environment for Virtex II.

Parts of the Partitioner presented in this paper are a Threading analyser and a tool for Timing and Power analysis.

2. THREADING ANALYSER

The purpose of the “Threading Analyser” is to take input in the shape of a C file and produce output of a number of “modules” as understood by the rest of the Partitioner environment consisting of C files with associated dependency and communication channel information. There is no conditional control flow between these modules, and each module needs to be executed only once.

2.1. Target Environment

The target environment for the threading efforts is the existing Partitioner environment as described in [1], which consists of an emulation environment and a partitioning tool with scheduling support.

The environment is illustrated to show what requirements it imposes on the threading code.

2.2. Partitioning Tool

The partitioning tool is designed to take a set of basic code modules and find a partitioning (assignment to different processing units, such a processors/FPGAs) and schedule for these modules that will optimise the performance of the unit.

2.3. Emulation Environment

The emulation environment provides support for running a simulation of the partitioned code on the vendor-supplied simulators for the different kinds of processing units supported. It was decided early in the project that it was desirable to use the vendor-supplied simulators in preference to a custom or third-party simulation environment.

While investigating the different simulators to be used it was determined that they did not universally provide support for the degree of access to the internal simulation state, or the possibility of integrating their memory models that would be required to allow parallel simulation of code on several of these simulators at the same time. It would also not be possible to allow the code running on one of the simulators, such as the armulator, to influence the execution of code running on one of the other simulators.

As a result, it was necessary to create a framework that would allow the parallel schedule determined for the different modules of code to be simulated sequentially on the different simulators, executing one module at a time on the simulators but keeping track of the global time in the overall simulation framework to simulate parallel execution.

The model used for exposing parallelism to the Partitioner is essentially a data-flow model between blocks of code that contain sequential code within them. This model was chosen as it provided the only feasible way of allowing parallel code to be written for the simulation environment that can be supported on the simulators we use, as we can support neither shared-memory, message passing or other forms of synchronisation. The only time we can allow the passing of data, as well as synchronisation between blocks of code, is between executions of the simulators.

This model is reminiscent of HeNCE [2], which provides a graphical environment for specifying procedures as the nodes of a graph with the edges between them signifying their dependencies.

The task of the threading analysis is to extract such blocks of code from a sequential program and identify the data-flow between these blocks.

This is in marked contrast to the traditional task of parallelising sequential code, which targets conventional parallel computer systems. These tend to be either shared-memory or distributed-memory systems [3].

While a data-flow model of parallel computation has been used extensively in the literature, especially as an intermediate stage in the parallelisation of functional programming languages (a data-flow model can be seen as analogous to a functional model, and the translation between them is easy), this has been traditionally at much finer granularity rather than as data-flow between bigger blocks of sequential code.

2.4. Compiler Framework

The threading analysis requires a compiler framework that supports source-to-source translation, since we are required to use the compilers provided by the vendors of the different execution environments. Providing backends for all the different CPUs used currently or in the future would have been unfeasible. Further it was desired to have access to a framework that allows compiler passes to be written in a high-level language to allow rapid development of such passes.

A number of compiler frameworks where considered for the threading analysis:

- GCC
- LCC [4]
- SUIF [5]

GCC is perhaps the most popular compiler framework available today as it is supported on every common platform and has a very large number of backends. However, GCC does not support source-to-source translation well as most of its transformations take place on very low-level representations of the code. GCC is also written in C and is of such complexity that any significant changes to its codebase would take considerable time to implement.

SUIF was designed as an environment for compiler research. It is written in pre-ISO C++ and supports a wide range of analyses. However, the only C frontend available for SUIF is a commercial one which requires a special license to be obtained for commercial use, and in fact it has lately been very difficult to obtain this frontend at all. Also, it appears that SUIF has not been updated since 2001 and there are strong doubts about its future viability [6].

LCC is a relatively simple compiler framework written in Ansi-C. While by itself it does not support the features we required, it has in recent versions gained support for writing out the intermediate representation in ASDL format [7] which allowed it to be imported into our own compiler framework, allowing LCC to be used as a compiler frontend.

It was decided to implement our own compiler framework in python, using the LCCs ASDL output to gain access to the intermediate representation. This allowed progress to be made relatively quickly, however some limitations in LCCs intermediate representation have been identified that make extension to more complete support of the C language difficult.

2.5. Parallelisation

A large body of work exists on the problem of parallelisation of sequential programs [8, 9, 3]. However, traditionally most of this work has focused on the problem of extracting fine-grained parallelism, such as vectorization, as it was thought that the biggest scope for extracting parallelism was in the parallelization of inner loops [3]. Recently the focus has shifted somewhat to the problem of coarse-grained parallelization [8].

The algorithm designed for this study operates as follows:

The entire program is read into the compiler framework (currently only a single .c file is supported). The main function in the program is identified and all function calls are inlined where possible. This allows the following analysis steps to be carried out across function-call boundaries.

The code of the program is analysed in a sequential manner. First each access to a variable is investigated to decide whether it is a write or read access (a define or a use). In the intermediate representation currently used, only an ASGN node represents a write access to a variable.

Next the program is analysed to find basic blocks (a common compiler analysis step [10]), that is a block of code that does not contain any branches to other blocks other than at its end, and that has no branches from other parts of the code terminating inside it. This analysis is performed by starting a new basic block wherever a label is encountered or a previous block was closed and ended wherever a branching instruction is encountered.

In the next step, Zones are formed from these basic blocks. A Zone is a set of basic blocks that has no control flow with the basic blocks contained in any other Zone other than basic sequential flow, that is no loops or conditional branches. Such a Zone is formed by merging any blocks with those blocks already in a Zone that can reach a block in the Zone by such a loop or branch. This is repeated until there are no further blocks that can be added to the Zone.

As a final analysis step, Zones can be combined in order to generate fewer Zones (each Zone is translated into a separate module for consideration by the Partitioner at the end of the analysis).

After all analysis steps have been completed, each Zone is written into a separate module for consideration by the Partitioner and is passed to the Partitioner together with information about its communication channels and dependencies.

2.6. Limitations

Currently only a subset of C is understood by the Threading Analyser. Understood are the int type, functions without returns, all the basic arithmetic operations and conditional operations. External function calls are not implemented yet but will be implemented next, as will the return statement. Some of the other data types, in particular structures, arrays and pointers may take some extra time to implement.

3. TIMING AND POWER ANALYSIS

In general, power analysis requires estimates of power consumption of the individual component parts of the architecture, such as ISA Processor, Digital Signal Processor, FPGA and buses, to be calculated either during or after a simulation run. Different approaches have been proposed in the literature depending on the type of architectural components and the simulation environment.

3.1. Processor

Two approaches have been proposed for estimating the energy consumption in processors, other than the prohibitively expensive (in terms of processing and memory requirements) circuit-level simulations at the netlist level using SPICE-like tools.

3.1.1. Instruction-Level

Where a cycle accurate simulation of the processor with tracing or plugin support is available, then it can be used to look up pre-determined energy values on a per-instruction basis. This approach was first applied to simulations of a 486DX2 and Fujitsu SPARClite 934 by Tiwari et al. [11, 12]. It has since been applied successfully to a number of other processor designs including ARM7 [13], StrongARM [14], and an unspecified Fujitsu DSP [15]. Although Tiwari et al. applied their analysis statically, this technique can be readily adapted to simulation traces.

3.1.2. Architectural-Level

In an architectural power analysis approach, the system is decomposed into a set of architectural units with individual power models. Such decomposition for a processor could consist of array structures, memories, combinational logic/wires, and clocking [16]. A big improvement over previous results has been claimed for approaches based on the dual bit type method [17]. In this approach, the energy contribution of the most significant bits in each word is considered separately from that of the least significant bits. This takes account of the considerable impact of the most significant bits when using two's complement arithmetic (as the most significant bits can be seen as indicating the probability of a change in sign). An architectural-level power analysis framework for use with the SimpleScalar simulation framework is available [16], and the approach has also been applied to simulation of the energy costs of co-processors in ARM-based SOC designs [18].

Architectural-level power analysis requires an architectural model of the processor as well as detailed power models for each component of the processor. While this information can be obtained for custom designs using circuit-level simulation tools such as SPICE it is generally not possible to obtain this information or simulation models for off-the-shelf processors.

3.2. Memory and Caches

Memory is usually modelled using a per-access cost model [16]. Where detailed information about the memory chips used is available, it is possible to consider the impact of page faults. The dual bit type method can also be applied to memory if a sufficiently detailed model of the memory is available [17].

3.3. Buses

Buses can be modelled on the basis of the number of capacitance switches on the lines involved during each bus transaction [14]. If the necessary information about the data transferred and the design of the buses are not available during power analysis, then a per-access power model can be used with limited accuracy.

3.4. Conclusions

An instruction-level simulation approach based on Tiwari et al's method was chosen for the Partitioner as it is a good match for the simulators and architectural information available for the processors used in the project.

This approach is also easy to adapt to any alternative processor architectures that may be chosen in the future.

Per-access models were chosen for memory and bus accesses as no detailed information for these components was available.

3.5. Timing and Power Analysis in Partitioner

Timing and Power analysis is performed differently depending on processing unit architecture, due to differences in development methodology and simulation environments. For the Xilinx architecture, a combination of cycle counting and extraction of power data from the XPower tool is used. For the ARM an analysis of the simulator trace is performed, combined with stop-instructions to distinguish the code under test from testbench code. For the TigerSharc architecture a test harness that uses the (simulated by the VisualDSP simulator) processor cycle counter is used, together with an estimate of average current draw.

3.5.1. Xilinx

As the Xilinx architecture is a FPGA architecture, a common timing method is not assumed but it is common to use a clock signal to time a design. However, there is no instruction trace (as there are not generally any instructions executed) and the clock signal is not easily identifiable. To nonetheless provide an environment for gathering timing statistics from the simulated design, a counter needs to be set up in the testbench for the design and cycle counts are extracted using this counter.

Power Analysis for the Xilinx architecture is performed by the XPower tool that is part of the Xilinx toolsuite. The current draw for the device is automatically extracted from the output file of this tool.

3.5.2. ARM

The ARM processor architecture is based around single-pipeline processors. The ARM7TDMI processor design was studied during this work, but the methodology can be easily adopted to other ARM processor designs as they share a common development environment and simulator design.

The ARM simulator (ARMULATOR) has a provision for writing a simulation trace file during simulation of the module under test. This file provides a trace of all instructions executed during the simulation run and further provides information on all the memory accesses performed. Unfortunately, no cycle counts are provided for the instructions executed, however the cycle duration of virtually all ARM instructions is constrained by the memory accesses involved during the instruction execution. In the case where an instructions cycle count is not thus constrained an average number of extra cycles executed can be provided to the Partitioner environment through the “arminst.txt” file.

Current draw for the different instructions encountered during the simulation needs to be provided by the user of the Partitioner environment in the “arminst.txt” file.

The format of the “arminst.txt” file is as follows:

```
I default 0.32 0
I skipped 0.12 0
I MUL 0.62 3
I MOV 0.20 0
```

where I indicates a line providing information about an instruction, followed by the name of the instruction or one of a small number of special code words, followed by the average current draw during execution of this instruction and further an estimate of the “extra” cycles likely to be encountered during execution of this instruction. For the MUL instruction a conservative estimate of 3 extra cycles was used, though in actual code it is unlikely that 3 extra cycles would be encountered as this number of extra cycles would only occur during uncommon boundary conditions. Nonetheless it may be appropriate to provide a pessimistic estimate of the cycle count.

The special code words implemented currently are “default” and “skipped”, providing the information for an instruction for which no specific information was provided (appropriate for instructions that do not occur in the inner loops of a design) and for an instruction that was skipped (conditionally not executed) respectively.

In order to distinguish instructions in the trace that belong to the code under test from those that are part of the testbench, “stop instructions” are inserted into the assembled code for the module under test. These “stop instructions” are constructed as effectively noop instructions built up from a combination of arithmetic operations using integer constants that can be identified by the Partitioner but are very unlikely to occur in the code under test.

3.5.3. TigerSharc

The TigerSharc (TS101) processor has a 4-instruction pipeline, meaning that it can execute up to 4 instructions in parallel. Parallelisation across the 4 pipelines is performed statically at compile time by the VisualDSP compiler. The VisualDSP framework provides a graphical view of the execution flow of instructions through the processor pipelines during simulation. Unfortunately it has not been possible to find an automatic method for extracting this information in text form from the VisualDSP framework. However as it is possible that such a method may be found / made available by the VisualDSP designers an analysis of the possible use of this information for power and timing analysis was performed.

The pipeline view in the VisualDSP framework provides two views on the information, either as a decompiled instruction trace showing the arithmetic-style assembly codes for each instruction or in the binary instruction coding as used within the processor. The assembly format of the TigerSharc architecture does not easily lend itself to analysis as a fairly complex parsing would be required. It is suggested that the (easily decodable) binary instruction representation be used instead.

The EX bit specifies whether the instruction is last in an instruction line. This is used in case fewer than 4 instructions are scheduled in one instruction line by the compiler to indicate a shorter than maximum instruction line and avoid the use of extraneous NOP instructions.

The CC bit specifies that the instruction is conditional. In the TS101 instruction set nearly all instructions can be specified as being conditional in execution on the state of a conditional execution flag that can be set by one of a number of evaluative instructions. The X and Y bits specify whether the X and Y compute block or both of the processor are used by the instruction. The remaining bits of the instructions header specify a multiply instruction, and the header is followed by a specification of the operands of the instruction which are of little use in timing and power analysis.

Should a method of extracting the pipeline trace from the VisualDSP environment be found it should be possible to trace the activity of each compute block of the processor in each instruction cycle, thus providing a very detailed power analysis.

In the absence of an accessible pipeline trace a cycle count for the TigerSharc architecture is obtained by taking advantage of the cycle count register available in the TigerSharc processors. Preprocessor macros are used to insert code that calculates the number of cycles executed within the regions of code under test. This methodology is useful as it is very quickly adaptable to other processor architectures as long as either a cycle count register or a high resolution timer is simulated within the simulated environment. This is the case for both the ARM and TigerSharc simulators and is likely the case for most industrial-strength simulators.

4. INTER-PROCESSOR COMMUNICATION

Inter-processor communication is simulated using a shared-memory model. The shared memory is accessed by buses from each of the processing units.

The Partitioner can extract information about the number of bytes exchanged through each of the channels between modules. The channel name needs to be the same in the Partitioner as well as in the testbench code.

For each processing unit in the simulated system, both an access time / energy draw for initiating a transmission as well as a per-byte transfer time and energy draw can be specified for its bus connection. Equally the time and energy draw for accessing the memory can be specified.

The time required for a communication between two modules on different processing units is the sum of the access times for the two bus interfaces and the memory and the per-byte transfer times for each component involved in the communication multiplied by the number of bytes transferred.

5. REFERENCES

[1] M. Halimic, D. Bourse, and E. Nicollet, “Optimal Functional Mapping onto End-to-End Reconfigurable (E^2R) Equipment Hardware Platform,” SDR Forum Technical Conference Proceedings, Orlando, USA, 2006.

[2] A. Beguelin et al. HeNCE: a heterogeneous network computing environment. Scientific Programming , 3(1):49-60, Spring 1994.

[3] T.G. Lewis et al. Introduction to Parallel Computing, Prentice-Hall, 1992

[4] C.W. Fraser and D.R. Hanson. A retargetable compiler for ANSI C. Technical Reports CS-TR-303-91, Princeton, N.J., 1991.

[5] R. Wilson. An overview of the SUIF compiler system.

[6] S.I. Lee et al. Cetus – an extensible compiler infrastructure for Source-To-Source Transformation. In L. Rauchwerger, editor, Languages and compilers for Parallel Computing: 16th International Workshop, LCPC 2003, volume 2958 of lecture Notes in Computer Science, pages 539-553, College station, TX, USA, 2 October 2003, Springer

[7] D.R. Hanson. Early experience with ASDL in lcc. Software – Practice and Experience, 29(5):417-435, 1999.

[8] K. Kenedy. Compiler technology for machine-independent parallel programming, International Journal of Parallel Programming, 22(1):79-98, 1994.

[9] M. Wolfe. High Performance Compilers for Parallel Computing. Addison-Wesley, 1996.

[10] A.W. Appel. Modern Compiler Implementation in Java. Cambridge University Press, 2nd edition, November 2002.

[11] V. Tiwari et al. Power Analysis of Embedded Software: A First Step Towards Software Power Minimisation.

[12] V. Tiwari et al. Instruction level power analysis and optimization of software.

[13] F. Menichelli et al. A Simulation-Based Power-Aware Architecture Exploration of a Multiprocessor System-on-Chip Design

[14] T. Simunic et al. Cycle-accurate simulation of energy consumption in embedded systems.

[15] M.T. Lee et al. Power analysis and minimization techniques for embedded DSP software.

[16] D. Brooks et al. Wattch: a framework for architectural-level power analysis and optimizations.

[17] P. Landman and J. Rabaey. Architectural Power Analysis: The Dual Bit Type Method.

[18] D. Crisu et al. High-Level Energy Estimation for ARM-Based SOCs.