USING ISMENE TO DEBUG AND PREDICT THE PERFORMANCE OF AN EMBEDDED SYSTEM DEVICE DRIVER

GAIUS MULLEY

School of Computing, University of Glamorgan
Treforest, Mid Glamorgan, CF37 1DL, UK

Abstract: This paper reports on performance results gained while modelling an embedded system, how the associated workload was constructed and finally documents the results measured on the actual embedded system. The main hardware components of the embedded system consisted of a Pentium 32 bit microprocessor, 32 Mbyte of memory and a ne2000 10/100 Mbit/sec 802.3 network interface card. The software for the embedded system was built from lightweight processes which are coordinated through a preemptive executive running in a microkernel. This paper reports of the measured and simulated performance of the device driver and the simulated throughput results were found to be accurate to within 8%.

Keywords: m2f, gdb, binutils, embedded system, simulation, pentium, device driver.

1. INTRODUCTION

In the life cycle of an embedded system, performance analysis is often undertaken after a commitment has been make to key components. Sometimes performance analysis and benchmarking are undertaken after a product has been completed and the opportunity to alter significant aspects of design are thus denied. To compound this problem it is not easy to extrapolate the final performance figures from individual component specifications. Furthermore the interrelated nature of devices, executive timeouts, DMA cycle stealing, interrupt overhead, application code, device driver code means that optimizing any one entity might not deliver the required performance improvement.

The embedded systems of interest are constructed from a uniprocessor based microkernel controlling specialist hardware through device drivers implemented as a set of interrupt driven processes[Comer, 1984]. The technique adopted in this research is discrete event simulation[Fishman, 1978] as it exploits the correspondence between the implementation of next event simulation and the multitasking executive[Mulley, 1988]. The event generation method is a model program operating on an abstract machine termed model processing element (MPE). The MPE consists of a model processor (is32) and model hardware devices[Mulley, 1995].

2. REVIEW OF PREVIOUS AND SIMILAR WORK

In the discussion of previous work it is useful to extend the *exo* and *endo* architectural classifications given by Dasgupta[Dasgupta, 1989a] to include a *system* architecture classification. Essentially an *exo* architecture definition consists of the hardware specifications required by an assembler programmer. The *endo* architecture definition describes the register transfer level or circuit/gate level. The review presented here extends these classifications by including a *system* architecture level which is used to describe the architecture at a higher level than the *exo* architecture. Typically this would describe processes within the computer system and their interaction.

Perhaps the most influential hardware design notation was Instruction Set Processor Specifications (ISPS), which appeared in the early 1970's. This notation described the architectural schema of a computer system. Instruction Set Processor (ISP) was the first notation which described the *exo* architecture of a system rather than the *endo* architecture. This notation formed the basis for what eventually became ISPS which was designed and implemented by Barbacci[Barbacci, 1981]. ISPS was a very important architecture design language as an interpreter was written for it. ISPS also provided the basis for other design tools. For example it was used to derive compiler code generators[Cattell, 1980] and ISPS was later extended[Parker, 1981] to describe interface hardware such as computer the UNIBUS[ANSI, 1979]. SLIDE is at a higher level of abstraction

than ISPS and uses *P* and *V* semaphores[Dijkstra, 1968] for process synchronisation; which in SLIDE indicate hardware synchronisation such as handshaking, bus requests and bus grants.

Although ISPS was ground breaking it was really designed to describe and analyse the performance of the processor rather than a complete computer system. Fundamentally the simulator was modelling the *endo* architecture[Dasgupta, 1989b] and thus it suffers a reduction in speed of several orders of magnitude during execution as the simulator is interpreting below register transfer level. This reduction in speed could, for example, preclude the investigation of complex high level language experiments.

More recently the SimpleScalar[Austin, 2002] tool set has been constructed and it provides an infrastructure for simulation and architectural modeling. It has been used to model a variety of processors ranging from the StrongArm[Intel, 2001], Alpha[Compaq, 1999], and Pentium[Intel, 1995]. This tool set, written in C[Kernighan, 1988], is highly modular and allows various components to be replaced or extended. The SimpleScalar simulator has been exploited to simulate an iPAQ although a number of devices are missing. Nevertheless sufficient has been implemented to allow the Linux kernel to boot.

The SimpleScalar tool set allows for very accurate performance modelling and also enables designers to examine power consumption, cache performance and memory accesses. It is an *exo* architecture simulator which also allows very detailed device modelling.

Adding new devices to SimpleScalar requires new modules to be implemented and integrated into the tool set. Modelling a different processor requires a large recoding and reconfiguration effort within the simulator.

3. ISMENE AND THE MODEL PROCESSING ELEMENT

The approach taken with Ismene is to produce an abstract machine simulator which includes processor, devices, microkernel executive, interrupts and DMA. Modelling a different processor within Ismene requires a new mapping between real processor instruction costs and the Ismene abstract machine processor (is32).

Ismene implicitly models the executive which allows users to examine the amount of time spent in key system procedures. For example it can determine the amount of processor time spent in each process or in the *P* and *V* procedures.

Ismene is a next event simulator embedded systems design tool and it is used in conjunction with a number of components: Modula-2 workload generator (`m2f`), GNU binutils and GNU debugger (`gdb`). The key component interaction is shown in figure 1.

The goal of Ismene is not only to determine the performance of an embedded system before it is constructed but also to provide an environment in which debugging such a system is advantageous.

Ismene allows different solutions to be explored together with their performance implications. The user expresses the software model in Modula-2 which is later automatically transformed into an is32 workload description. As the design progresses the Modula-2 model becomes closer to final production code[Wirth, 1971]. Finally the model and production code are the same therefore it is possible to debug the production embedded system through Ismene. Ismene has many advantages when debugging the embedded system:

(i) ability to *suspend* time while the system state is examined.

(ii) ability to single step *all* code: interrupt service routines, device drivers and application code.

(iii) ability to *rewind* time and when using a modified `gdb`; single stepping the source code in reverse; allowing the user to answer the ‘‘how did we get here?’’ question.

(iv) break points may be set on hardware and software events. For example start of DMA activity, timer interrupt, `iret` instruction. All break points interact seamlessly with `gdb`.

4. EVENT GENERATION THROUGH A MODEL MACHINE

Ismene uses the technique of execution driven simulation to generate the appropriate events. The is32 code is loaded into Model Processing Element (MPE) and interpreted, each instruction generates at least one software event (the next instruction) and it may create a new hardware event (by manipulating a device control register). Each MPE contains a model of a processor and devices. It is intended that these models contain many of the important features found in

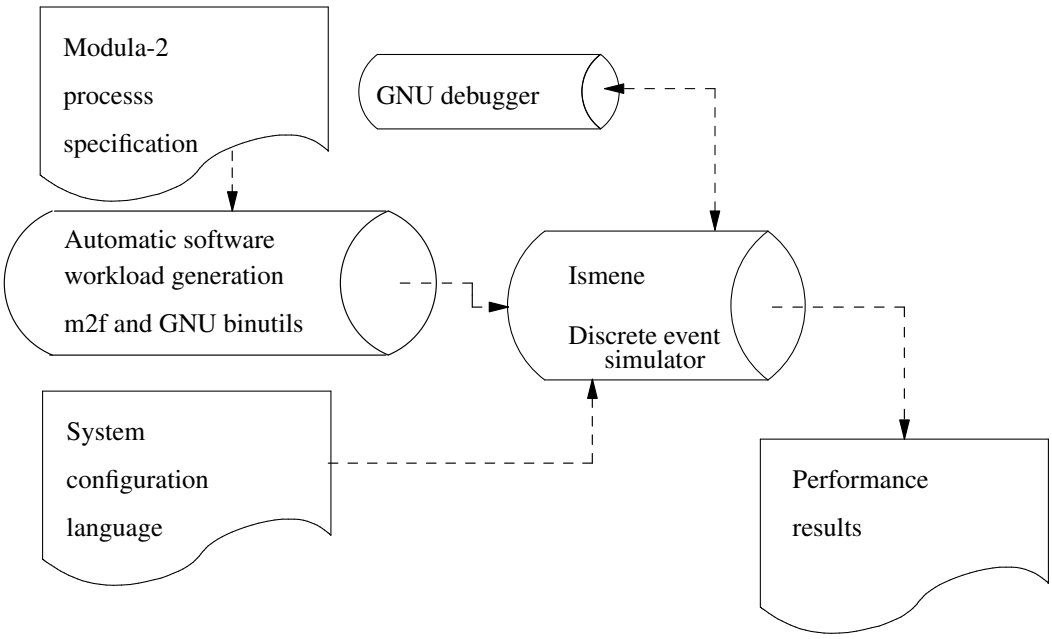

Figure 1: Ismene and related components

traditional processors and devices.

Detailed modelling of an embedded system implies representing conditional behaviour of processes. Thus the process representation must allow for examination of conditional variables. Any manipulation of synchronisation primitives by a process will have a direct or indirect affect on the remaining processes. Furthermore device workload upon the processor can be closely modelled if the concept of interrupt mask register, DMA cycle stealing, processor cycles, interrupt priority and interrupt activation schema are present.

4.1. SOFTWARE EVENT GENERATION

It is desirable that software loads from a variety of different microprocessors can be mapped onto the processor (is32) within the MPE. The software workloads are allocated to the is32. A detailed model of software can be built to faithfully replicate functionality and execution cost[Mulley, 1988], alternatively, a higher level model can be produced which represents software in skeleton form (processes are briefly described in terms of fundamental interprocess communication primitives and an aggregate execution cost).

The simulator advances the current process one is32 instruction at a time. A simulation event is associated with the beginning of the execution of an is32 instruction which, will usually correspond to one or more instructions on a real processor. The events associated with the current instruction are placed onto the event queue and the next event simulation technique effectively interprets a process. A time of completion (or time of new event) must be calculated for each instruction.

The is32 instruction set can be categorised into executive calls `wait`, `signal`, `initprocess`, `killprocess` and `initsemaphore`; assignment, iteration and conditional primitives; device control such as `in`, `out`, `iret`, `disable`, `enable`; request for the processor `pr`.

The request for the processor, `pr`, represents an average load of a block of *real* machine code instructions whose functionality is not modelled in detail at the current refinement level.

4.2. HARDWARE EVENT GENERATION

The MPE must be configured with real system overheads. The MPE hardware characteristics such as processor speed, instruction speed and rate of device transmission are specified in the system configuration language (SCL). Within the SCL a basic unit of a device is termed a port. A port is specific to one MPE and its performance characteristics remain constant throughout a simulation experiment. This language describes a system in terms of a collection of interconnected MPEs and each MPE is individually described in terms of machine clock frequency, speed of

processor, memory configuration and ports. The description of a port determines whether it is a model character device, clock device or block DMA device, the amount of cycles DMA steals from the processor, duration of DMA burst, priority of DMA and the speed of operation. Ports can contain links to other ports and therefore more complex device models are created by combining these units. The SCL and MPE has been described in detail[Mulley, 1989, Mulley, 1995] elsewhere.

The ports are accessible to the is32 by explicit input/output instructions and the processor communicates to a model device controller by sense status polling or a ready interrupt.

5. EMBEDDED SYSTEM TO BE MODELLED

The embedded system to be modelled is an inexpensive Pentium based unit running at 133 Mhz containing a 10/100 Mhz 802.3 ne2000 compatible network interface card. Previous research using Ismene has shown that hand coded software workloads can produce results accurate to within 4%. However these were extremely tedious to produce and they were modelling simple microprocessors[Zilog, 1977, Motorola, 1984] which had no instruction pipelining or instruction cache. Modelling the execution time of instructions running on the Pentium presents a challenge as this microprocessor has an instruction pipeline, data/instruction cache, address generation interlocks, opcode pairing rules, operand conflicts and cache conflicts[Intel, 1997].

6. SOFTWARE WORKLOAD CALCULATION

Modelling the software workload accurately is very important if reliable network performance figures are to be obtained. An increase in processor performance can very often yield the same increase in network data throughput[Tanenbaum, 1997]. The Pentium contains a U and V pipeline. All instructions may enter the U pipeline but only a subset of the instructions may enter the V pipeline simultaneously. Furthermore operand conflicts between the instructions in the U and V pipelines will incur wasted cycles. There exist a number of different reasons a pipeline might stall and some of these are outlined in table 1. The instructions which appear in bold face in table 1 are causing a pipeline stall.

As an experiment the `m2f` compiler was modified to include an extra command line option `-Oworst` which generated the same Pentium instructions but in the worst pipeline order. The best ordering was seen to execute in 60% of the time compared to the worst. Clearly the software workload must incorporate this knowledge.

One solution considered was to modify Ismene to contain an accurate Pentium model, this is the approach taken by the SimpleScalar project[Austin, 2002]. However this would be a large software undertaking and the downside is that Ismene would become processor specific.

Another solution is to modify the compiler to issue timings for each and every basic block[Aho, 1986]. The compiler could be improved to understand the exact performance implication of the U and V pipeline, instruction caching, data caching and operand conflicts. For every basic block the compiler firstly calls the is32 code generator to model the code behaviour and then secondly calls the Pentium code generator to work out the execution cost. The execution cost is inserted into the is32 generated list of instructions using a `pr` instruction. The Pentium code generator is a real production code generator but in this instance it does everything other than actually emit the Pentium instructions. This solution has the advantage that after the embedded system is simulated and the satisfactory results are obtained the same compiler is used (with a different set of command line options) to generate the real Pentium instructions for the target system. The components of the `m2f` compiler are shown in figure 2.

This solution was adopted and initial tests showed that this algorithm was accurate to within 8%. These tests consisted of timed programming constructs such as while loops, assignment, array accesses, bubble sort, string manipulation and pointer manipulation examples. The results shown in table 2 were obtained by compiling the test code for the Pentium and executing the code under the GNU/Linux tool `perfmon`[Goda, 1999]. The same code was then compiled for the is32 processor with the compiler predicting the execution time on the Pentium for each and every basic block. Finally the is32 code was issued to Ismene and the total simulated execution times obtained.

The results were satisfactory, the 8% error was probably due to imperfect U/V pipeline knowledge when control jumped from one basic block to another.

U pipe	V pipe	Explanation
`addl 4,%esp`	`popl %esi`	Address generation interlock (AGI) stall between the U and V pipes, as both are attempting to utilise `%esp`.
`movl 123, %eax`	`inc %eax`	Operand conflict between the U and V pipe when assigning `%eax` and incrementing `%eax`.
`movl 4096+4, %eax` `movl 4096*2+9, %ebx`	`nop` `nop`	Cache conflict when attempting to utilise the same cache line.
`movl %eax,0(%esi)`	`movl %ebx,32000(%esi)`	Imperfect pairing due to cache bank conflict
`push %ax`	`push %bx`	Not all combinations of instructions are pairable, the two `push` instructions are manipulating the same 4 byte word.
`nop`	`adc 123, %eax`	Some instructions will only pair if they are issued to the U pipe.
`pop %eax`	`pop %ebx`	Some instructions will pair even if they are both manipulating the stack.

Table 1: Some examples of Pentium pipeline rules

Test module	Simulated ÷ Actual time × 100
func	104.4
ptr	102.5
simplestr	96.1
sort	92.8
testbec44	102.1
testbec45	100.4
testparam	102.1
teststr	108.6
vector	95.4

Table 2: Pentium software workload accuracy

7. DEBUGGING SUPPORT WITHIN ISMEME

There are two interfaces available when debugging with Ismene; the Ismene front panel and the `gdb` interface. Normally users will invoke `gdb` via a development environment such as `emacs` or `ddd`. The Ismene front panel includes an execute forwards and execute backwards button which allows the user to simulate the embedded system, reach a bug and ask the question, ‘‘how did we reach this point?’’. At this stage the user can click Ismene execute backwards button and `gdb` can be told to single step in reverse. Eventually the question will be answered. This mechanism works well as it allows users to interrogate the complete system during execution in reverse. For example users can iterate backwards through loops, step backwards into called functions and print variable values which come back into scope. For this to work it is necessary to restore the data, BSS and stack region contents at the appropriate time.

The debugging support implemented within Ismene was achieved by keeping a circular history of the last n instructions, the register contents prior to the execution of this instruction, the memory addresses it modified and previous memory contents at these addresses. This allows Ismene, when single stepping in reverse, to view memory through a history of patches, thus giving the correct memory and register state at that particular time.

Providing there is a clean procedural interface between the implementation of the processor model and the memory model then this mechanism requires minimal implementation effort. Debugging in reverse became an important feature of the Ismene simulation tool.

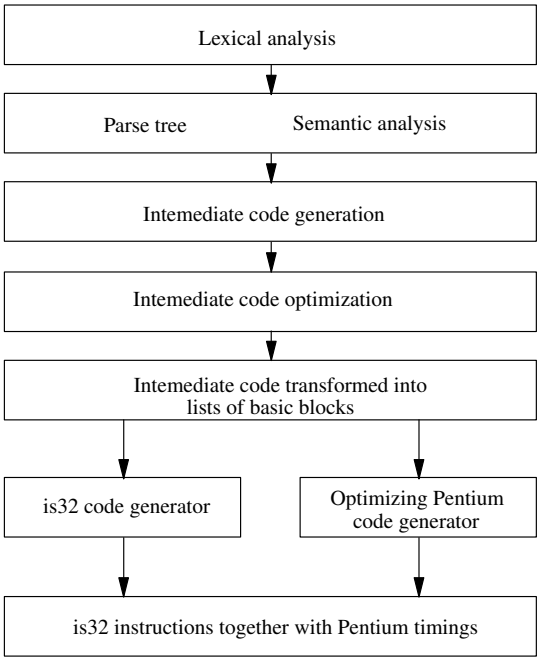

Figure 2: Components of the `m2f` compiler

8. PERFORMANCE OF AN EMBEDDED SYSTEM

A Pentium based embedded system running at 133 Mhz controlling a ne2000 network interface card operating at 10 Mhz was simulated, built and measured. The software consisted of a microkernel, device driver and minimal application. The ne2000 specific code was taken from the Linux kernel and translated into Modula-2. For the purpose of these tests the application is an infinite source which continuously transmits fixed size frames.

The software profiling within the simulation tests showed that the device driver was processor bound (95% polling) waiting for the transmission of large blocks to complete. Whereas when the system was transmitting smaller block sizes the interrupt handling, device delay and copying the frame into the network interface card (out/in) matched the cost of executing the polling routine. Smaller block sizes incurred a noticeable cost in having to wait for the hardware to react to software control (device delay 15%).

It would be interesting to see the effect of running the network interface card at 100 Mbits/sec without the device delay penalty. This simulation experiment was performed and the results are shown in table 4.

The device driver is split into two components: the shared 8390 device code and the ne2000 specific code. To achieve this an object oriented technique was used in the C version inside the Linux kernel and also in Modula-2 on the embedded system. This design allows for software reuse in devices which utilise the 8390 chip-set but it was found to account for 6% of the available processor time when transmitting 64 byte frames.

As this network interface card is transmitting blocks ten times faster than the previous experiment it can be seen that less time is wasted polling for the outgoing packet to complete (poll). Consequently the executive, timers and memory buffer management have more of an impact on the software cost.

9. CONCLUSIONS AND FURTHER WORK

In conclusion the technique of modifying a compiler to generate accurate workloads based on the predicted execution time of each basic block has proved successful even when targeting a processor which employs complex instruction pipelining and caching. The combination of Ismene and `gdb` is attractive as an embedded system can be simulated and debugged at the source code level and the single step in reverse facility is a potent debugging aid.

Frame size (bytes)	Live results	Simulated results				
	Throughput KBytes/sec	Throughput Kbytes/sec	Percentage of processor time			
			Polling	Interrupt	Device delay	Out/In
64	263	328	45	7	15	27
96	n/a	395	56	3	12	23
128	n/a	437	63	3	10	19
256	n/a	525	77	2	6	11
512	634	600	87	1	3	7
1024	703	700	93	0	2	3
1514	731	739	95	0	1	3

Table 3: Comparison between simulated and live embedded system

Simulated results							
Frame Bytes	Tpt Kbytes	Percentage of processor time					
		ne(poll)	Exec	8390	Timers	MBuf	Sys
64	1969	24(19)	18	13	8	12	16
96	2560	33(29)	15	11	7	11	14
128	3011	41(37)	14	11	6	10	13
256	4096	59(57)	9	7	4	7	9
512	4995	75(73)	6	4	3	4	5
1024	5769	86(85)	3	2	2	2	3
1514	5937	90(89)	2	2	2	2	2

Table 4: Simulated embedded system using a 100 Mhz network interface card

In the future it would be interesting to see whether these techniques could be applied to GCC. In particular it would be interesting to see whether GCC can pass the basic block execution timing of other complex processors to Ismene with similar success.

REFERENCES

Aho A.V., Sethi R. and Ullman J.D. 1986, *Compilers: Principles Tools and Techniques,* Addison Wesley.

ANSI 1979, "U.S.A. contribution to ISO TC97/SC13 for small computer to peripheral bus interface standard," American National Standards Institute, Inc., ANSI Technical Committee X3T9, New York.

Austin T., Larson E. and Ernst D. 2002, *SimpleScalar: An Infrastructure for Computer System Modeling,* 35(2), Pp. 59-67, IEEE Computer Society Press, Los Alamitos, CA, USA. ISSN:0018-9162.

Barbacci M.R. 1981, "Instruction Set Processor Specifications (ISPS): the notation and its applications," IEEE Transactions on Computers, C-30(1), Pp. 24-40.

Cattell R.G.G. 1980, "Automatic derivation of code generators from machine descriptions," ACM Transactions on Programming Languages and Systems, 2(2), Pp. 173-190.

Comer D. 1984, *Operating System Design: The XINU Approach,* Prentice-Hall International, Englewood Cliffs.

Compaq 1999, *Alpha 21264 Microprocessor Hardware Reference Manual,* Compaq Computer Corporation.

Dasgupta S. 1989, *Computer Architecture: A Modern Synthesis Vol 1 Foundations,* John Wiley and Sons, New York.

Dasgupta S. 1989, *Computer Architecture: A Modern Synthesis Vol 2 Advanced Topics,* P. 61, John Wiley and Sons, New York.

Dijkstra E.W. 1968, "Cooperating Sequential Processes," Programming Languages, Pp. 43-112, Academic Press, New York.

Fishman G.S. 1978, *Principles of Discrete Event Simulation,* John Wiley and Sons, New York.

Goda M. 1999, *Performance Monitoring for the Pentium and Pentium Pro Under the Linux Operating System.* `http://qso.lanl.gov/~mpg/permon.html`.

Intel 1997, *Intel Architecture Optimization Manual,* Intel Corporation, P.O. Box 7641, Mt Prospect, IL 60056-7641.

Intel 2001, *Intel StrongARM SA-1110 Microprocessor Developers Manual,* Intel, USA.

Intel 1995, *Pentium Processor Family Developer's Manual,* Intel Literature, P.O. Box 7641, Mt. Prospect, IL 60056-7641, USA.

Kernighan B.W. and Richie D.M. 1988, *The C Programming Language,* 2nd Edition (ANSI C), Prentice-Hall.

Motorola 1984, *MC68000 16 bit Microprocessor Users Manual,* Fourth Edition, Motorola Inc.

Mulley G.P.C. 1989, *A design tool for performance prediction of realtime systems,* P. 126, PhD Thesis, University of Reading.

Mulley G.P.C. 1995, "Device Driver Workload Modelling Through an Abstract Machine," Performance Engineering of Computer and Telecommunications Systems, Proceedings of UKPEW 1995, Liverpool John Moores University, UK.

Mulley G.P.C. and Loader R.J. 1988, "The application of a real time systems simulation design tool to protocol engine design" in Procedings of the International Conference on Data Communication Technology, ed. Burkley D, Pp. 240-250, The National Institution for Higher Education.

Parker A.C. 1981, "SLIDE: an I/O hardware descriptive language," IEEE Transactions on Computers, C-30(6), Pp. 423-439.

Tanenbaum A.S. 1997, *Computer Networks,* 3rd Edition, Prentice-Hall.

Wirth N. 1971, "Program development by stepwise refinement," CACM, 14(4), Pp. 221-227.

Zilog 1977, *Z80 - CPU Technical Manual,* Zilog Inc.

AUTHOR BIOGRAPHY

Gaius Mulley is a senior lecturer at the University of Glamorgan. He is the author of GNU Modula-2 and the groff html device driver `grohtml`. His research interests also include performance of microkernels and compiler design.

He obtained a Bsc(Hons) in Computer Science from the University of Reading in 1985 and a PhD in Computer Science from the University of Reading in 1989. He then worked for Meiko Scientific before joining the University of Glamorgan in 1994.