The Use of ASIPs and Customised Co-processors in an Embedded Real-Time System

Jack Whitham and Neil Audsley
Department of Computer Science
University of York, York, YO10 5DD, UK
{jack|neil}@cs.york.ac.uk

July 13, 2005

Abstract

Embedded systems are increasingly being designed in a system-on-chip (SoC) form to save costs. Often, a field programmable gate array (FPGA) device is used to contain the processor, memory and devices used by the system. In this environment, it is possible to make use of an application-specific instruction processor (ASIP) in place of a conventional processor, and it is also possible to add customised co-processors to the device. Both of these technologies can be optimised to accelerate a particular application, potentially reducing hardware costs and power consumption.

However, embedded systems are generally also real-time systems, and the current approaches for determining where optimisations should be applied fail to take this into account, neglecting important real-time properties such as deadlines and response times. We demonstrate that designers may be led to poor optimisation decisions by such approaches, then show how timing properties may be combined with profile data to allow informed optimisation choices to be made.

1 Introduction

Embedded systems designers frequently wish to reduce per-unit cost and power consumption, while at the same time avoiding increasing design costs. The use of Field Programmable Gate Array (FPGA) and Application Specific Integrated Circuit (ASIC) technologies allow per-unit cost, power consumption and overall system size to be reduced by fitting most system components onto a single chip. This is known as a system-on-chip (SoC) architecture.

FPGAs and ASICs allow hardware to be customised to a particular application, which can allow substantial savings to be made. The general approach involves moving frequently executed sections of code into hardware, which can increase the speed of the entire system. A simpler processor or slower system clock frequency can be used, perhaps saving cost, power, physical space, or some combination of the three.

Application Specific Instruction Processors (ASIPs) allow system performance to be optimised at the architectural level by inclusion of custom hardware within the processor to support new instructions (e.g. [20, 4, 3]). Thus, they require modification of an existing processor core. ASIP manufacturers sell this existing core with the tools needed to customise it.

Custom co-processor devices allow customised hardware to be included at the bus level. They can be added to any system with any processor core. They also execute independently of the main processor, so some parallelism is possible.

In both cases, the key design time issue is that of function selection, i.e. which functions of the application should be implemented with custom instructions or using a co-processor. Conventionally, this is achieved by revealing frequently executed areas of code using a profiler [20, 24, 2]. These areas are then translated into either a hardware description language (HDL) for inclusion in a co-processor, or, in the case of an ASIP, into a combination of assembly code and custom instruction descriptions in a HDL. This process can be carried out automatically [2], but it is best to do the work by hand,

because a designer will have a high level understanding of the function of a particular piece of code which cannot be derived from source code, allowing the hardware to be used more efficiently [12].

In this paper, we assume that the platform is such that either an ASIP approach or co-processor approach could be taken, and that a hybrid approach combining elements from both is possible. In practice, the designer may be constrained to one particular choice by hardware or cost restrictions. In this case, the work is still applicable, but some of the choices are restricted.

In a hard real-time system, many distinct processes with different timing characteristics must share processing resources. In this environment, current approaches for function selection are not effective. Profiling individual processes does not reveal where the best system-wide improvement can be made, especially where processes interact with each other. This paper focuses on the need for improvements to existing approaches for function selection, and proposes methods that may be used to improve the existing approaches. These methods are then applied within a case study.

The paper is structured as follows. In section 2 background and previous work is discussed. Discussion of instruction selection issues for real-time systems is given in section 3, and section 4 details the solution that we propose to avoid the problems inherent in current approaches. Section 5 presents the results of a practical case study. Finally, conclusions are given in section 6.

2 Background

2.1 ASIPs

ASIPs are conventional processors that allow additional custom instructions to be added to the standard instruction set architecture (Figure 1(a)). Custom instructions are defined at design time: the hardware to execute them essentially forms a fixed part of the ASIC (or FPGA softcore). Usually, the instructions are incorporated within the RTL for the entire processor [4].

The number of custom instructions allowed is limited primarily by the target hardware, not the ASIP or instruction set. The overhead of incorporating custom instructions, in terms of the additional hardware required, is significant. In [23], “a few additional instructions” need “18,000 additional gates” in one sample application, with “image filtering and color-space conversion” custom instructions needing an “additional 64,100 gates” for another application. The basic processor requires only 25,000 gates [21].

Usually, the types of custom instructions permitted can be quite complex, e.g. single-instruction multiple-data (SIMD) vector processing and multiply-accumulate (MAC) instructions [23]. Often, custom instructions are kept to a single cycle in length, although they may have state, enabling more complex functionality to be implemented [23].

2.2 Customised Co-processors

In some applications, it may be inappropriate or impossible to include custom hardware within the processor data path. For example, the processor may not be user-modifiable, being a hard core within an FPGA, a soft core that is sold without RTL source code, or a prefabricated chip. It is also possible that the designer wishes to exploit the isolation between a co-processor and the main processor and have tasks running in parallel on both of them, or that the RTL for a particular function may require a level of complexity that is not available within the data path, such as a need for extremely wide registers.

In these cases, a customised co-processor may be placed on an external bus (Figure 1(b)). The co-processor is often a memory mapped device, with the ability to generate an interrupt when processing is complete. Co-processors must be self-contained devices, including both the hardware required to carry out the desired function, and control hardware to manage bus accesses, which does mean that they take up more hardware space than the equivalent custom instruction based implementation.

We consider only the case of customised co-processors which are used to speed up processing functions. In practice, it is common for device driving hardware to do some processing itself in order to assist the main processor (such as CRC/parity checks in communications hardware), but as this function is not easily separable from the hardware itself, device driving processors are not considered to be co-processors for the purposes of this paper.

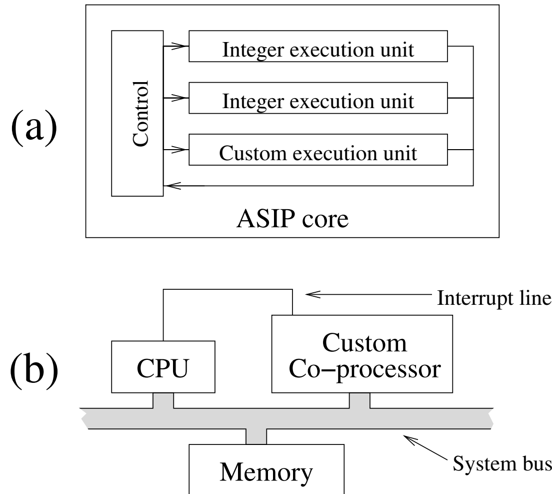

Figure 1: (a): an ASIP core provides customisable execution units on the processor data path. (b): a customised co-processor is generally an additional memory mapped device on the system bus, connected to an interrupt line which is used to signal task completion.

Cascade [2] is a commercial co-processor generation tool providing similar features to ASIP design tools [20]. Unlike custom instructions, co-processors can be arbitrarily complex, and may execute in parallel with the main processor. The number of co-processors that can be introduced is effectively limited only by the size of the hardware platform. However, the communication overhead with a co-processor is substantial. Co-processors have no access to processor registers, and all data must be transferred to and from them via the system bus. For these reasons, co-processors are more suited to complex tasks that take a substantial length of time to process.

2.3 Common Features

The use of either technology may lead to a need for a large FPGA or ASIC, with associated higher costs and power consumption. However, these problems can be reduced by effective selection of code for optimisation. Conventionally, an iterative process is used [20, 24, 2, 11], involving code profiling to find the best places for optimisation, and then implementation of that optimisation using custom instructions or a co-processor. This process is repeated until the code executes sufficiently quickly.

The use of profiling for optimisation selection is similar to the simulation approach used in classic co-design methodologies [13, 17], where profiling is used to decide which application functions are to be implemented in a co-processor. In that case, the implementation is automatic, essentially a direct translation of the software source code to hardware. This highlights an important distinction between co-design and the more recent approaches, in that ASIP approaches rely on the designer to implement the custom instructions, and to make the final decisions about implementation strategies (Cascade makes similar provisions for co-processors). Better optimisations can be made by applying a high-level understanding of the application's requirements, which avoids the sub-optimality of automatic software to hardware translation [12].

However, the profiling approach is limited, as it does not address real-time performance issues arising from multiple interacting processes with complex timing requirements. While [3] does propose tools to profile process performance characteristics within a real-time system, there is no attempt to relate the profiling information to any form of timing analysis in order to guide the selection of custom instructions or co-processor functions.

As the number of custom instructions and co-processors will always be limited by the amount of hardware available, correct selections must be an essential part of optimising system design. The correct choices can minimise hardware cost and power consumption, and may allow the system clock frequency to be reduced. However, making these choices remains an open issue in the context of real-time systems.

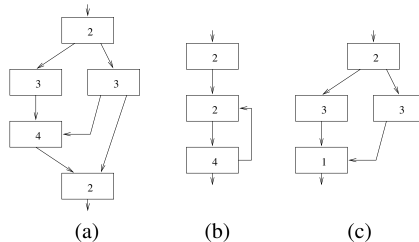

Figure 2: WCET Calculation and Refinement.

3 ASIPs, Co-processors and Real-Time Systems

The correctness of a real-time system is dependent on *when* an application function occurs, as well as the correct operation of that function [7]. Too early and too late are equally bad. Typically, a real-time system consists of a number of application processes, each with specific timing constraints. Hard real-time systems (such as safety-critical systems) contain processes with deadlines that cannot be missed without possibility of catastrophe (e.g. aircraft engine failure).

Hard real-time systems have processes that are either periodic, i.e. invoked at regular (fixed) intervals; or sporadic, i.e. are invoked at random times, but with an enforced minimum time between successive invocations. Note that the effects of sporadic processes are bounded, but the precise times at which they are invoked are not known *a priori*. User input and network traffic are both examples of sporadic processes.

Processes may share resources, such as a software buffer (i.e. memory block) in an exclusive manner, such that blocking may occur when a process wishes to access a shared resource. Also, processes may be affected by interrupts, and operating system overheads (e.g. context switch times).

For hard real-time systems, temporal predictability must be proved prior to system execution - i.e. timing requirements must be shown to be met offline. Testing cannot show the absence of timing faults, hence analytical approaches are used [7]. Timing analysis models the worst-case timing behaviour of the system, given the scheduling (and resource management) policies incorporated in the operating system. For each process, its worst-case response time (i.e. elapsed time between invocation and completion) must be no greater than its deadline. The response time is dependent upon its own computation time, the execution times of other processes, and any time that the process must wait (i.e. whilst blocked) for access to shared resources (e.g. shared memory used in inter-process communication and synchronisation, physical devices). Many such analysis methods have been proposed, for different scheduling policies, e.g. fixed-priority (FP), earliest deadline first (EDF) [7].

Within real-time systems, there lie a number of opportunities for the use of custom instructions and co-processors to improve performance, in terms of process response times. The remainder of this section proposes different strategies that could be used.

3.1 Reducing WCET

The response time of an individual process is dependent upon its own computation time. For hard real-time systems, the worst-case execution time (WCET) is used. This represents the worst-case timing path through executable code of a process [7]. This is calculated by breaking the process code into a graph of basic blocks (single entry/single exit blocks of non-branching code), as given in Figure 2(a). A basic block will always execute in a constant time (assuming conservative pipeline and cache behaviours). The WCET for a process can be computed by finding the path through the graph that maximises the total process time, in this case 11. Note that to calculate WCET requires bounded code, i.e. bounded loops (Figure 2(b) shows a process with undefined WCET, due to its unbounded loop).

ASIP-style custom instructions and co-processors can often replace one or more basic blocks. Conditional statements and loops can be parallelised so that they always execute in a fixed time. Figure 2(a) might be simplified to Figure 2(c) by the addition of a custom instruction which replaces some of the blocks, resulting in a revised WCET of 6. In all cases, if the WCET is known before the

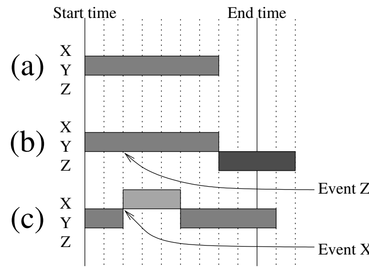

Figure 3: Response Time Example.

use of custom instructions, it will be known afterwards.

3.2 Reducing Main Processor Utilisation

Co-processors can also be used to replace basic blocks, but a co-processor is capable of much more than a single custom instruction. There is potential for parallelism between a co-processor and the main processor: both can execute at once, whereas custom instructions operate just like regular machine instructions - if one takes many clock cycles to execute, the processor pipeline will become stalled and the entire processor will wait for it to complete. Co-processors can take as long as they need to complete without stalling other processes, if the system is designed to take advantage of parallelism. Additionally, it is possible for a co-processor to speed up the system as a whole even if it is no faster than the equivalent software implementation.

The limited parallel model proposed in [5] may be used to model the parallel operation of co-processors. [5] distinguishes between co-processors that are invoked in a master/slave scenario, in which the main processor (master) waits for the co-processor (slave) to complete, and those that are used in a parallel scenario, and describes how co-processors used in the latter scenario can be incorporated into a timing analysis.

However, [5] does not consider how the functions of co-processors should be selected. Nor is context switch and communication time considered. However, all of these factors are of high importance in a practical application. Using a co-processor takes time for communication of data across the memory bus, and will also result in a context switch if parallelism is being utilised, as the main processor will begin executing another task after the co-processor has begun execution. Context switch and communication time are significant unless the co-processor execution time is far greater than both of them.

3.3 Improving Response Times using Custom Instructions

Consider processes X, Y and Z with WCETs of 3, 7 and 4 units, and deadlines of 4, 9 and 25 respectively. Both X and Z are sporadic (with minimum inter-arrival times 6 and 40 respectively), Y is periodic (with period 20). They are assigned priorities such that X $>$ Y $>$ Z.

For a fixed priority scheduling policy, the worst-case response times of X, Y and Z are 3, 15 and 22 respectively. Thus, X and Z meet their deadlines, whilst Y does not ($15 > 9$). At run-time some invocations of Y will meet their deadlines if X is not invoked at its maximum frequency (Figures 3(a) and (b)), but in the worst-case it will miss its deadline if X is invoked (Figure 3(c)).

Conventional profiling may suggest targeting process Y for custom instructions as it has the highest WCET. This is not supported by timing analysis, as even reducing it to 4 units (from 7) leaves the modified response time (10) still exceeding the deadline (9). This is due to the worst-case involving two invocations of X (see Figure 4).

In this case, X is a better target for optimisation, even though it is the shortest process when profiled in isolation. Reducing its WCET to 1 unit reduces the response times of X, Y and Z to 1, 8 and 13 respectively.

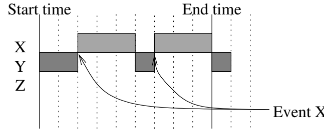

Figure 4: Optimisation of Response Time.

3.4 Reducing Effects of Process Interaction

A further sporadic process (W) is added to those in the example of section 3.3. W has a WCET of 3 units, and both W and Z make use of a resource, R. Each time W runs, it locks R for 3 units, and each time Z runs, it locks R for 1 unit.

Z's WCET (4) is greater than W's (3). Conventionally, Z would be targeted before W. However, W locks a shared resource for a longer period of time than Z. W may block Z by up to 3 units, as Z has to wait for W to release the lock on R. In this situation, it may be beneficial to target W and reduce the amount of time that R is locked for. As before, profiling W and Z in isolation will not reveal this behaviour.

3.5 Summary

Hardware implementations of software functions can speed up the entire system, but the correct function must be chosen for hardware implementation. A method for finding that function is needed. Furthermore, because there are different types of hardware implementation, the method must be also able to suggest which type of implementation would be best: custom instructions, or a co-processor. The method must take the timing properties of processes (e.g. deadlines) into account, along with their run time behaviour (e.g. inter-process interaction).

4 Our Solution to the Real-Time Function Selection Problem

We present a solution to the problem. The solution first describes the selection of a set S of basic blocks from a process P ($S \subset P$), which is the set to be optimised by hardware implementation. Then, a method of selecting the best type of implementation is described.

Function selection can be carried out either by measurement or analysis. An analytical method for finding S would take the known worst-case minimum period between each execution of a process, and use it to calculate the worst-case execution frequency of each basic block in that process. However, this approach omits information about process interaction and is likely to be overly pessimistic, as it must assume the worst case in all cases.

Measurement is a more straightforward alternative, requiring far less analysis and yielding "typical" performance values instead of the worst cases. It is done by profiling a prototype of the system. This will reveal a set of candidates for S, which we will call C. The prototype must be run in conditions that simulate actual operation, as system behaviour may be dependant on input data. The advantage of this measurement approach over analysis is relative simplicity.

This is not conventional profiling of a single process or application. The entire real-time system, including operating system, is profiled, thus implicitly incorporating information about process interaction. We refer to this type of profiling as whole-system profiling.

4.1 Choosing S by Analytical Methods

We now have the set C, containing some sets of basic blocks that are frequently executed. These sets are candidates for S. An analytical method may be used to find the best candidate.

First, calculate the WCET of every process. Next, calculate the direct effect that the choice of each candidate will have on the WCET of each process. Each candidate will affect only one process (the one that it is part of). The effect will be a reduction in the WCET, due to some part of the process being implemented in hardware. Put this data into a table similar to Table 1, which lists resulting WCETs for each candidate choice.

Candidate	Proc. A WCET	B WCET	C WCET
None chosen	15	10	6
Choice 1	15	8	6
Choice 2	10	10	6
Choice 3	9	10	6
Choice 4	15	10	2

Table 1: The effects of each choice of S on the WCET of each process.

Candidate	Proc. A WCRT	B WCRT	C WCRT
None chosen	15	25	56
Choice 1	15	23	29
Choice 2	10	20	26
Choice 3	9	19	25
Choice 4	15	25	27

Table 2: The effects of each choice of S on the WCRT of each process. We have assumed that process A has a period of 30, and B and C have periods of 40.

Using these figures, it is possible to calculate the worst-case response time of each process for each candidate. Table 2 shows an example using the figures from Table 1. The worst-case response times must all be less than or equal to their associated deadlines, so this will allow some candidates to be eliminated. For example, if the deadline for process C was 26, then choices 1 and 4 (and "none") would be eliminated in this example.

The only remaining candidates are "safe" - no matter which one is chosen, deadlines will be met. The best one should be chosen. We consider the best choice to be the one that minimises overall system utilisation, as this may allow the designer to use a lower clock frequency or a cheaper implementation platform. Table 3 shows the utilisation figure for each (remaining) candidate, calculated by dividing the WCET of each process by the period of that process, and adding up the results.

This indicates that the third candidate is the best choice for S. However, there is one aspect of this choice that has not yet been checked: what is the effect on other processes? This aspect is best examined by simulation or prototyping. Attempting to model process interaction is defeated by the same problems that defeat fully static analysis of large programs - there are too many variables. We recommend either implementing S, or implementing a software model of S, and then testing it out. The next section will describe the use of software models as part of an alternative to an analytical approach.

4.2 Choosing S by Non-analytical Methods

Analytical methods are not always appropriate. In the previous section, we made the assumption that the WCET of every process could be calculated. This is not always possible. We also assumed that the set of processes was fixed, and that the system was a hard real-time system in which every process has a precisely defined period or minimum inter-arrival time. In practice, embedded systems are not always so simple. Many are soft real-time, and meeting deadlines is a quality-of-service issue instead of a safety-critical issue. These systems are often under-engineered from a hard real-time perspective - they work perfectly provided that processes don't need much more time than their average execution time.

For such systems, we propose the use of a non-analytical method. One could evaluate each possible candidate for S in a prototype or simulation, and choose the one that works best. This is similar to

Candidate	Utilisation
Choice 2	1.48
Choice 3	1.40

Table 3: Utilisation figures for each remaining candidate.

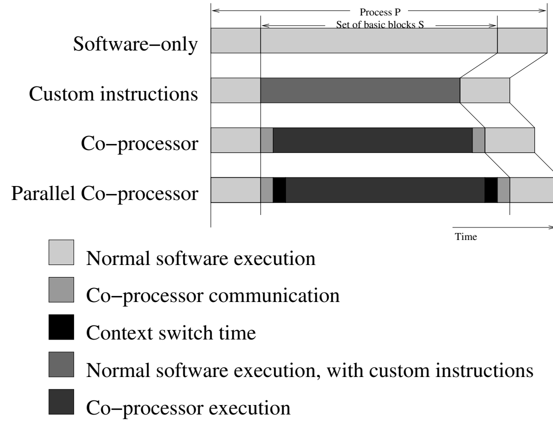

Figure 5: Four different implementations of process P.

the co-design approach to finding the best hardware/software partition, but it is directed only at a few likely candidates rather than all functions.

Evaluating each candidate would normally involve substantial implementation work. Fortunately, this does not have to involve any hardware design, as simulators should allow the use of a software model of some hardware. The software model carries out the same job that the hardware would do. It will generally be based on the original software. It is even possible to use software models that are incomplete implementations, provided that they have the correct timing properties, because this step only examines the effect of a particular choice on the overall timing of the system.

This technique allows each candidate to be modelled in the simulator or in the prototype. The designer can try out all the possibilities with very little implementation effort, and use profiling data from the prototype or simulator to choose the candidate that best fits the requirements. This approach will also show up any problems arising from process interaction. In effect, the period information that was incorporated by analysis in the previous section is incorporated implicitly by this technique.

4.3 Choosing the Implementation Type for S

Having found a suitable set of basic blocks for hardware implementation (S), we consider four possible implementations, which are shown in Figure 5.

Using custom instructions to carry out some of the operations in S reduces the execution time in comparison to software alone, and no extra communication or context switch time is introduced, because the custom instructions are fully inlined in the program and their operations are entirely register-bound.

Using a non-parallelised co-processor introduces some extra communication time, because the data that the co-processor uses must be sent to it, and read back after completion. Co-processors have no access to main processor registers. The software process waits for the co-processor to finish, so no context switching takes place.

Using a parallelised co-processor introduces context switch time, because other processes may execute while the co-processor is operating. Figure 6 shows a process P, which is partly implemented using a co-processor. Processes Q and R, which have lower priorities than P, execute while the co-processor is active. As soon as the co-processor finishes, the operating system returns control to P. As Figure 6 illustrates, two extra context switches are needed to support the parallel operation of the co-processor. Even if there were no lower priority tasks, this would still be the case, as the operating system would switch to the idle process.

Clearly, the different approaches have distinct advantages in different scenarios. Fortunately, a simple timing model can be used to predict which approach is best matched to a particular scenario. Let:

- N be the number of invocations of S during P,
- T_p be the co-processor WCET for S,

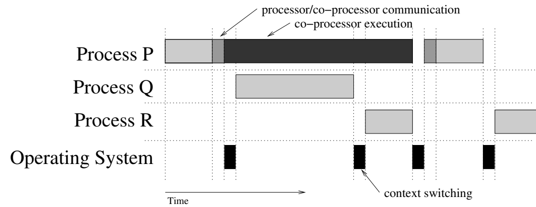

Figure 6: If a parallelised co-processor carries out some of the work of process P, lower priority processes Q and R can execute while the co-processor is active.

Implementation	WCET	Utilisation
Software only	NT_s	NT_s
Custom inst.	NT_i	NT_i
Co-proc.	$N(T_c + T_p)$	$N(T_c + T_p)$
Co-proc. (par.)	$N(T_c + T_w + T_p)$	$N(T_c + 2T_w)$

Table 4: Equations to allow WCET and utilisation time to be calculated for each implementation.

- T_s be the software-only WCET for S,
- T_i be the custom instruction WCET for S,
- T_w be the context switch time, and
- T_c be the overall (bi-directional) co-processor communication time.

For each implementation, Table 4 gives equations to work out the WCET and utilisation time. Utilisation time is equal to WCET in all cases, except for the parallel co-processor case. In that case, the main processor is not in use during co-processor execution, and utilisation time may be considerably less than the process execution time. Designers can use each of these equations to work out the overall utilisation for that implementation and choose the minimum.

We can expect T_i to be approximately equal to T_p. Any optimisation that could be applied to a co-processor to make it more efficient could also be applied to a custom instruction execution unit. We can also expect both T_i and T_p to be less than T_s: there is always a speed gain from moving software components into hardware, provided that the move is carried out correctly. However, no predictions can be made about the relationship between the other figures. The advantage of one approach over another depends critically on the values of T_c and T_w in relation to T_p and T_i.

To summarise this informally, if the context switch time dominates the co-processor execution time, then a custom instruction approach is likely to be better. However, if this is not the case, then significant processor time could be made available to other processes by the use of a parallel co-processor.

5 Case-Study: PDA

This section describes the application of our methods toward effective use of custom instructions and co-processors in a sample real-time platform: a personal digital assistant (PDA) with wireless networking and telephony support.

5.1 Experimental Platform

The Simplescalar [6] ARM simulator was augmented to form a generic ASIP with capacity for 256 custom instructions, accessed using opcodes that are undefined on a standard ARM processor. Custom instructions are implemented in C, and attached to Simplescalar using a plug-in system. Additionally, support for simulated memory-mapped co-processors was added to the simulator. Again, these co-processors are implemented in C, and attached as plug-ins.

	WCET/μs	Deadline/μs	Period/μs
V	59,400	250,000	250,000
N	2160	2160	700
I	60,000	100,000	100,000
J	280,000	1,000,000	1,000,000

Table 5: PDA process timing characteristics.

Due to a limitation in Simplescalar, any custom instruction taking more than one clock cycle to complete is assumed to stall the processor until it completes. Simplescalar does not support execution units that require an instruction-dependent number of clock cycles for their work.

The RTEMS [19] real-time operating system was ported to run within this simulator. RTEMS supports multiple processes with fixed priority scheduling. The port of RTEMS has been extended to make process execution statistics available. The amount of time spent in each process (including the idle process) is accounted.

5.2 Description of the PDA

The PDA has next-generation capabilities: wireless networking support and voice-over-IP (VOIP) support which can be used to place and receive telephone calls. The PDA also has typical features, such as a touch-sensitive screen. The effects of the different approaches towards custom instruction selection can be shown by concentrating upon three high-priority processes of particular interest:

1. V: the periodic process for VOIP processing;
2. N: the sporadic process driving the network interface;
3. I: the sporadic process which responds to user input.

A fourth process is also of interest. J is a graphics rendering process which runs sporadically to support a web browser. Its primary function is decoding JPEG images for display on screen. This process has a lower priority than the other three.

Processes V and N are active during a phone call, because voice encoding and decoding are taking place, and data is sent and received over a network. They must run at a high priority, as a call cannot be interrupted by other applications on the PDA. Process I becomes active when the user enters data into the PDA, by using a stylus on the touch screen. This must be responsive, even during a call.

The functionality of V, N, I and J is described further below. Table 5 shows the timing characteristics of software implementations[1]. The WCET values were discovered by measurement using the same sample data used throughout the case study. As can be seen by the disparity between the WCET and period of N, this is a soft real-time system.

5.2.1 N: Network Subsystem Process

The PDA wireless networking interface requires a software protocol stack, a low-level hardware driver, and an encryption system (for security). Wireless networks generally use WEP (wired equivalent privacy) technology, as it is specified as part of the 802.11b standard. We modelled this encryption system with 3DES, implemented using libdes [10].

The networking subsystem is modelled as a high priority process (N) that executes whenever a network event is signalled by the release of a semaphore. On execution, it copies a 1kb "packet" from one area of RAM to another, and encrypts that data using DES. This model captures both reception and transmission of data.

Network events are signalled by V to model the flow of data in and out of the system, but they are also signalled randomly to simulate traffic generated by other applications or other computers. The frequency of these random events is one of the parameters of the system that can be changed.

[1] 700μs is the minimum possible inter-arrival time for N. It was computed by assuming that packets of 1kb arrive at the maximum rate over an 11Mb/s wireless network.

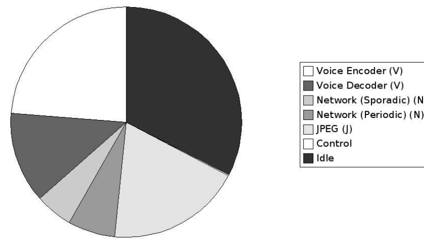

Figure 7: Distribution of CPU time between processes, with no user input or random network events.

5.2.2 V: Voice Subsystem Process

The PDA assumes a voice subsystem consisting of a G.721 codec. G.721 is a predecessor of the GSM standard, used for telephone-quality voice compression. Since the PDA receives and transmits voice data, V is split into a decoding and an encoding process which use the codec.

The MediaBench [16] benchmark suite implementation of G.721 was utilised, as it is ideal for use in an embedded environment. The codec operates on real voice data that is sourced from RAM, at a fixed rate of 4000 bytes per second. The data is handled in blocks of 1000 bytes which arrive every 250ms.

5.2.3 I: Input Subsystem Process

PDAs allow the user to enter data by using a stylus to draw glyphs on the screen. A simple form of handwriting recognition is performed on the glyphs. The input subsystem used here is modelled using the open-source gesture recognition program Wayv [18]. A recording was made using a desktop PC of various gestures being entered into Wayv. This is played back into a Wayv process running within the embedded system to simulate a user entering several symbols per second into the PDA.

5.2.4 J: Graphics Rendering Process

As the PDA has network connectivity, a web browser is provided. The web browser is able to render graphics in JPEG format. Process J handles JPEG decoding for this purpose.

We assume that the PDA decodes one image every second. Images are taken from a set of small pictures, each between 5k and 25k in size, intended to be representative of the types of image found on most web pages. The libjpeg [14] implementation is used.

5.3 Priority Assignment

N is assigned the highest priority. In a real system, the network subsystem would be interrupt driven, and thus would have one of the highest possible priorities.

Process I presents a choice, as priorities could be assigned so that I $<$ V $<$ N, or so that V $<$ I $<$ N. It is difficult to tell which configuration will yield the best results, so we experiment with both.

Process J has the lowest priority of all. Loading images is not a critical task.

5.4 Using a Software-Only Approach

The simplest way to implement the four subsystems is through software alone, with no special ASIP instructions. In this configuration, the system was allowed to run for ten seconds of simulation time: just over 250 million clock cycles with a clock frequency of 25MHz. Throughout execution, a simulated phone call was in progress, and a new JPEG image was loaded every simulated second.

Execution statistics were gathered after simulation. Figure 7 shows the distribution of CPU time between the processes over the execution period with the input process disabled, assuming no sporadic network events occur. Here, V takes up most of the processor's non-idle time: it is split into an encoding and decoding process.

Figure 8 shows the change in CPU idle time as the level of network traffic increases. We measure the network traffic by counting the total number of randomly generated network packets received

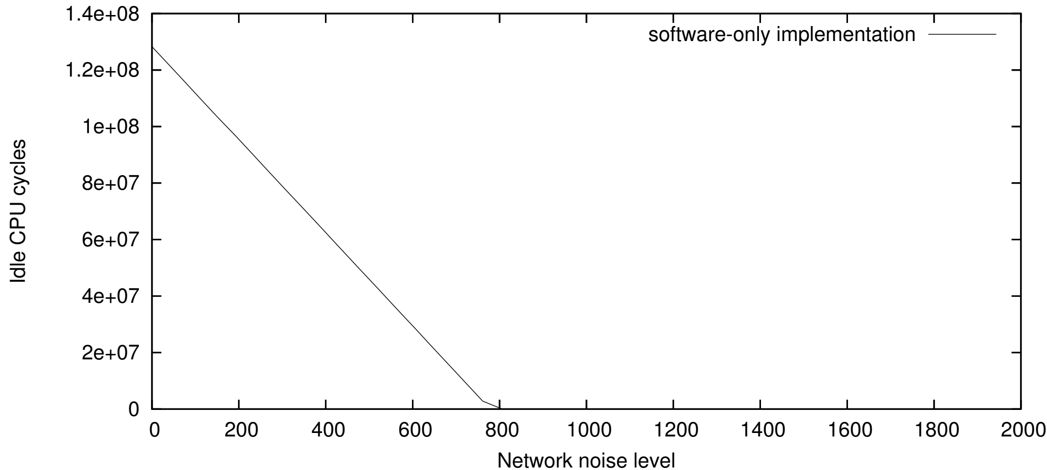

Figure 8: Changes in CPU idle time as network traffic increases, using software-only process implementations.

during the simulation run. In the scenario without any input during the call, the system can support up to 600 packets of network traffic before becoming overloaded.

5.5 Conventional ASIP Improvements

The ASIP method for improving the implementation involves first profiling it to find the most frequently executed parts (the hot spots), and then improving those places by replacing many instructions with a single custom instruction. Each process is isolated, and profiled separately to find its hot spots.

If this is done, then it will immediately become apparent that V takes up much more CPU time than N (Table 5). Using a conventional profiling approach, one might assume that the G.721 code would be the place to make improvements.

Profiling V in isolation reveals that most execution time is spent in the `quan()` function, and most of the calls to that function have a particular parameter set. The entire function can be moved into hardware as a custom instruction: even though it contains a loop and a condition, a high-level understanding of the purpose of the function (evaluation of log_2) allows it to be replaced with a simple hardware device (a priority encoder). We can expect this to take up just 11 LUTs (look up tables) on a Xilinx Virtex FPGA.

Much execution time is also spent in the `fmult()` function, which can also be replaced by hardware. The function is more complex - the cost of the extra hardware will be 352 LUTs. To reduce this cost, two different implementations were tried: "fmult a" is an all-hardware implementation of the function, and "fmult b" is a hybrid hardware/software implementation.

Figure 9 shows the results of the various enhancements to V. The "fmult a" improvement is the most successful, allowing up to 300 more sporadic network packets than the software-only solution. This improvement requires the definition of two custom instructions for `quan()` and `fmult()`.

5.6 Improving Response Times

The type of improvements discussed in the last section are the type which would be naïvely applied if the developer relied on a conventional profiling approach and improved the high-priority process that used the most CPU time: V. However, when we apply our method (see section 4), we are able to make a better-informed choice.

Our method begins by obtaining a whole-system profile, which we obtain by simulating the entire system using our modified version of Simplescalar. We obtain a whole-system profile under heavy network load conditions to gain information that is valid in the worst case. The most frequently executed code sections are listed in Table 6.

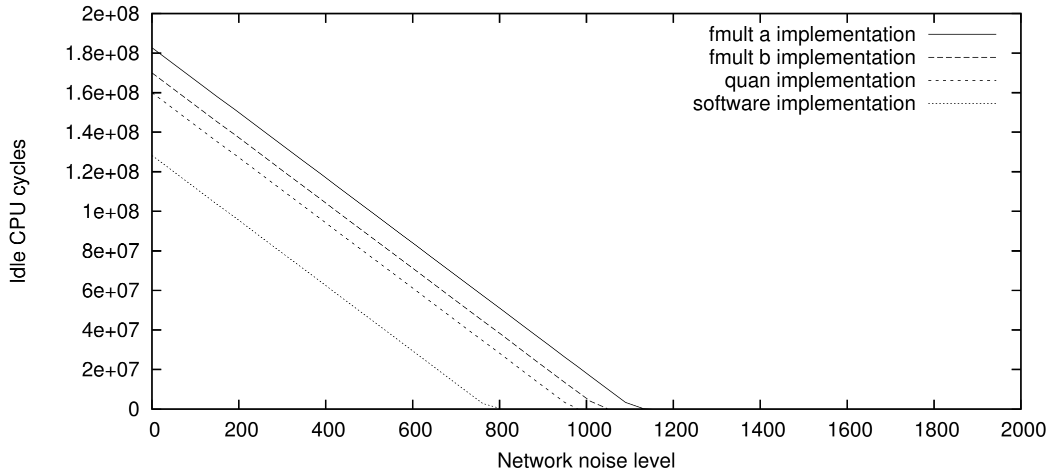

Figure 9: Effects of hardware improvements to V.

Exec. time	Symbol Name	Process
41.2%	`des_encrypt`	N
15.5%	`predictor_zero`	V
12.0%	`update`	V
5.7%	`jpeg_idct_ifast`	J
5.5%	`predictor_pole`	V
3.7%	`jinit_color_deconverter`	J
2.7%	`jinit_huff_decoder`	J

Table 6: Results of whole-system profiling for the case study.

This profiling turns up one clear candidate for optimisation. Most of the processing time is spent in process N, in the `des_encrypt()` procedure. V is also a candidate. Investigation reveals that an inlined version of `quan()` accounts for most of the time used by `predictor_zero()` and `update()`, as we found in the previous section. So in this case C = {`des_encrypt`,`quan`}. Our candidates are the DES encryption code from process N and the `quan()` code from process V.

In contrast to V, each execution of N does not use much CPU time. However, random network activity makes N a very significant user of CPU time, as Figure 10 illustrates. In Figure 10, handling network traffic accounts for over 50% of the processor's time.

In order to apply the analytical method to find the effects of each choice from C on the system, we first compute the effects of a hardware implementation on the WCET of each process, and find that V's WCET is reduced from 59,400μs to 23,500μs (choosing the best implementation option) and N's WCET is reduced from 2160μs to 60μs. Using this data, we compute the worst-case response time of each process, with each choice from C. Table 7 shows the results.

Table 7 clearly shows that the only candidate that can be chosen for optimisation is `des_encrypt()`. If no optimisations are carried out, then N is capable of starving both V and J of any processor time,

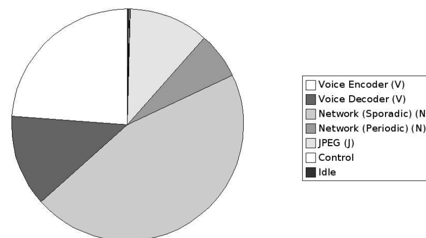

Figure 10: Distribution of CPU time between processes during high levels of network traffic.

Candidate	N WCRT/μs	V WCRT/μs	J WCRT/μs
none	2160	∞	∞
`des` (N)	60	64,980	436,240
`quan` (V)	2160	∞	∞

Table 7: Worst-case response times of all processes for each choice from C.

Implementation	WCET/μs	Utilisation/μs
Software	2,160	2,160
ASIP-style	60	60
Co-proc.	55	55
Co-proc. (par.)	60	58

Table 8: Values for the WCET and utilisation of process N, with four different implementation options.

as evidenced by their infinite WCRT values. This is because the minimum inter-arrival time of N is less than the WCET of a software implementation of N. In the worst case, network traffic can prevent any other processes from running, even if `quan()` is optimised. Therefore, `des_encrypt()` must be optimised first. Having done this, response time analysis predicts that N, V and J will meet their deadlines (700, 250,000 and 1,000,000μs respectively).

5.7 Possible Implementations of DES

Hardware optimisations for software are considered to take one of two forms in this work. As described in section 4.3, either a co-processor or ASIP-style custom instructions could be used. The method from section 4.3 will now be applied to find the best method to be used in this case.

The software-only WCET for N (T_s) is 2160μs. Using an ASIP-style approach, two types of implementation are possible: a single instruction that carries out the entire DES operation, or an implementation based on seven simple custom instructions, as described in the example in [24]. The first approach yields a WCET of 60μs, and the second yields a WCET of 1570μs. We choose the first approach here, so $T_i = 60\mu$s. However, the second approach may be valuable in cases where the amount of space available for additional logic is severely restricted.

150 DES operations are required to handle each entire network packet. Conventionally, DES processors encrypt one 64-bit word every 16 clock cycles, but the operation may be pipelined, so only 165 clock cycles are required to carry out 150 operations. This takes $T_p = 6.6\mu$s. The communication time is the time taken to transfer 150 64-bit words back and forth: $T_c = 48\mu$s. The context switch time $T_w = 5.21\mu$s.

We now have values for all of the timing model variables, except N, the number of invocations of DES during process N, which is 1. Applying the equations from Table 4 gives values for the WCET and utilisation of N, with each optimisation (Table 8).

Table 8 suggests that the best implementation is a non-parallelised co-processor. Parallelised co-processors bring extra context-switch time with them: there is a net increase in both the WCET and utilisation of N when a parallelised implementation is used.

Figure 11 shows the effect of each implementation on performance. This confirms the result obtained by analysis: a non-parallelised co-processor implementation is most effective.

5.8 Reducing Process Interaction Effects

Adding a user input process I (as described in section 5.2.3), affects the overall CPU time available to the system. Figure 12 shows three different configurations for process I (no input process, high priority, and low priority), with two types of DES implementation.

V and N make use of a shared network resource, and process I does not. Because execution of V generates a packet to be processed by the network, executions of V lead to additional executions of high-priority process N. When user input events are generated, process I uses CPU time, and reduces

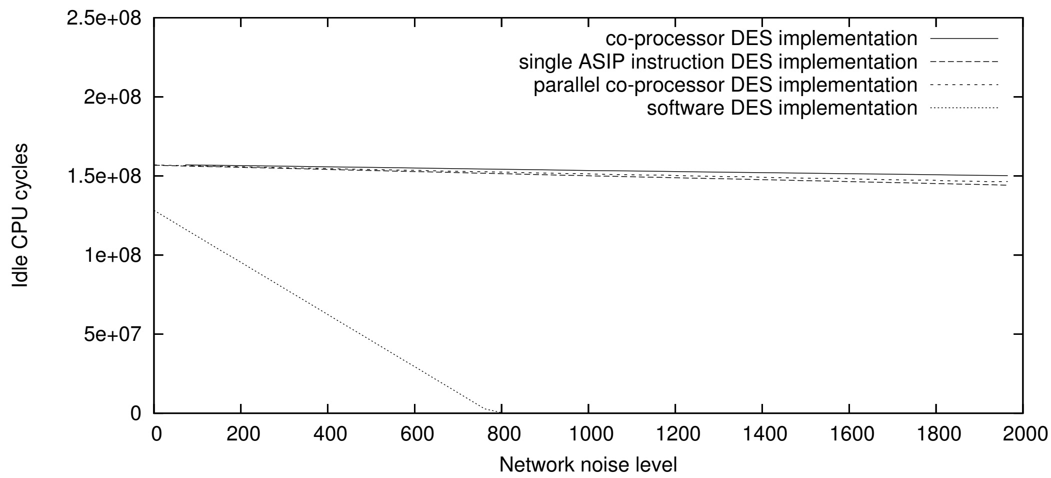

Figure 11: Effects of hardware improvements to DES.

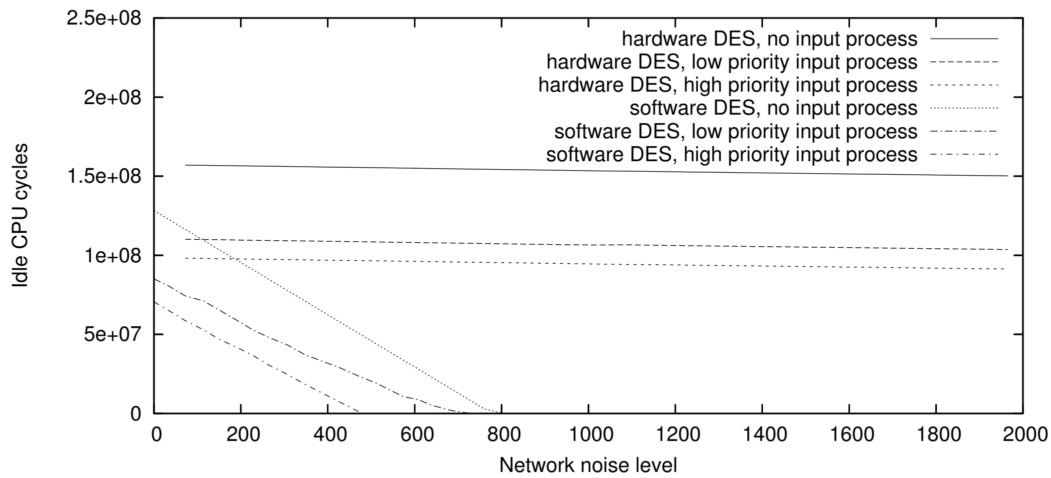

Figure 12: Effects of user input on the system.

Implementation	WCET/μs	% Improv. over sw
Software	188,200	0%
Parallel co-proc.	218,500	-16%
Non-parallel co-proc.	177,000	6%

Table 9: Values for the WCET of process J with three different implementation options.

the amount available for V when I > V. This reduces the number of executions of N, and causes the divergence between the high priority and low priority lines on Figure 12.

This is an example of how a lower priority process can affect the execution of a high priority process, and thus affect the execution of the entire system, similar to the example in section 3.4. Here, minimising N's WCET will minimise the effects of V's extra executions: when a hardware DES implementation is used, there is no visible difference between high priority and low priority input processes.

5.9 Freeing More Processing Time for Low Priority Processes

Process J has been largely disregarded in this discussion. It has the lowest priority: the assumption being that the user finds good quality of telephone service to be more important than JPEG images loading quickly. However, it is a significant user of processor time, using 19% of available time when the system load is low (Figure 7). One way to speed up J is to attempt to optimise it directly.

The most significant section of J is the IDCT (inverse discrete cosine transform) procedure, the core of the JPEG decoding algorithm [22]. IDCT is used for decoding several common still image and video formats, including MPEG-2 and MPEG-4. As a result, IDCT co-processors are readily available for use on FPGAs and ASICs for accelerating image and video decoding [8]. It is common practice to implement JPEG/MPEG decoders using both a program and an IDCT co-processor. The program handles the aspects of JPEG decoding that require a state machine, access to operating system functions, and access to memory, and the IDCT co-processor handles the decoding function itself. However, it is also common to make use of a complete JPEG decoder core, which is able to handle all decoding functions for data sent to it over the system bus [9]. These cores are much larger than IDCT co-processors. For example, the CAST JPEG core takes up 3923 slices [9] on a Xilinx Virtex FPGA, whereas the CAST IDCT core only takes up 1391 slices [8].

The libjpeg [14] implementation used in J is already very fast. Software modelling shows us that a parallelised co-processor for IDCT brings no benefits, without any requirement to work out execution times, because the processing time required is insignificant compared to the context switch time. However, modelling also shows that a non-parallelised co-processor may provide some improvement over software alone (Table 9).

Taking a whole-system view, much better results can be achieved by optimising process V. Improving V frees up more time for J. If the optimisations for V described in section 5.5 are applied in addition to those for N, then much more idle time is available for J, as can be seen in Figure 13. In fact, the time available to J (idle time, plus time used by J) increases by 29% - far better than the best improvement that could be made by optimising IDCT (6%). Extra hardware is more usefully applied to V, not J.

5.10 Energy Consumption

`sim-panalyzer`, a power modelling extension for Simplescalar, can be used to calculate the relative energy consumption of systems implemented using the various technologies under different loading conditions. For the purposes of this experiment, we assumed a 25MHz clock frequency, and 250mW power consumption during idle time. We also assumed that the custom instruction hardware used the same amount of power as an additional integer ALU. Figure 14 shows the variation in energy consumption for six implementations.

Figure 14 shares many features with Figure 12. The relationship between idle CPU times and energy usage is roughly linear up until the point where the system becomes overloaded. Reducing WCET and reducing power requirements are related problems - in this example, they are effectively equivalent. Future work could involve applying ASIP optimisations to explicitly maximise idle time,

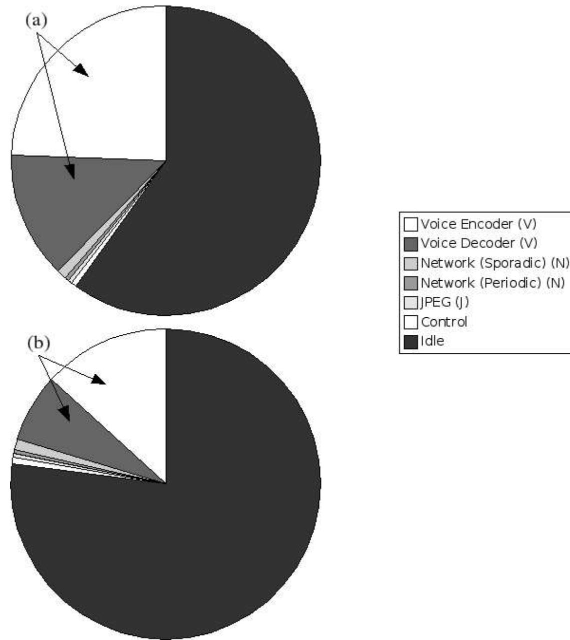

Figure 13: The effect of optimisations to V on the overall system. (a) shows the CPU usage of an unoptimised version of V, and (b) shows the CPU usage of an optimised version. In (b), much more idle time is available for process J.

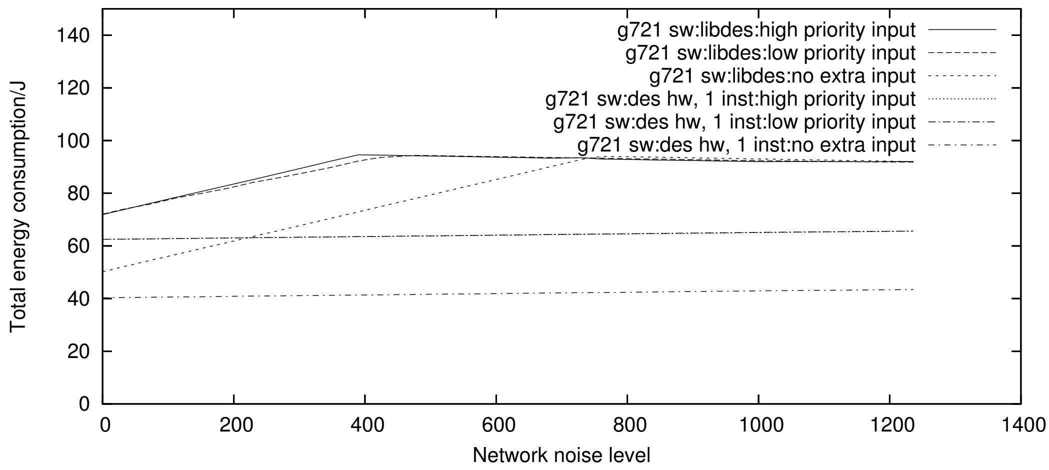

Figure 14: Total system energy consumption due to network traffic, with six different implementations.

which could be combined with the work on maximising procrastination intervals from [15] to reduce energy consumption even further.

5.11 Parallelised Co-processors

When analysing process N, and modelling process J, parallelised co-processors were considered, but eliminated from consideration because they provided no overall improvement. In both cases, this was because the context switch time proved to be significant. From this, we draw the conclusion that in order for a parallelised co-processor to be useful, it must offer a time saving that is substantially greater than the context switch time, and the communication time. There are benefits to using parallelised co-processors, but they are unlikely to be suitable for simple functions that require only a small amount of computation time.

In the case study, we found no examples of parts of processes that could be moved to a co-processor. It seems that the profiling technique is an inadequate way to find parts of processes that can be moved in this way, because profiling tends to find only short sections of code that occupy a great deal of processor time. It does not find long sections that are bottlenecks, such as complete processes, which would potentially work well in a parallel co-processor.

This finding is a general property of profiling, and not specifically a shortcoming in our method. It appears that the allocation of processes to parallel co-processors is a process that can only be guided by profiling data at present. For example, whole-system profiling can reveal which processes occupy significant proportions of processor time. This allows us to see that process V could be moved in its entirety to a co-processor, such as [1]. As V takes up 37% of the processor time with a software implementation, and 20% with an ASIP-style implementation, the benefits of implementing V as a parallelised co-processor could be significant: far more computing time would be available to other processes. The examination of these tradeoffs will be the subject of future work.

5.12 Case Study Findings

Our improvements to the case study have primarily targeted process N. These improvements allow the system to efficiently support far more network traffic than possible with a software-only implementation, by significantly reducing N's WCET. Additionally, we have determined that improvements to process V are a more effective way to speed up process J than direct improvements to J, and showed the effects of process interaction by looking at process I.

6 Conclusion

The conventional methods of applying ASIP and co-processor technology to a multi-tasking real-time system will not work in all cases, because process interaction must be considered. We have described two methods of determining how to take this into account when applying hardware optimisations: analysis, and measurement of software models running under whole-system profiling. Our case study has shown the application of these methods to an example system.

However, profiling the entire system as it executes on the target platform will only produce useful information if all input conditions can be tested. We would not have seen the value of targeting process N in the case study if we had not tried simulating the system under high network load conditions.

Simulation software provides an ideal environment for the types of profiling that are needed for analysis, because it allows both the operating system and applications to be profiled non-intrusively and as a whole. However, executing the whole system in any sort of prototyping environment is sufficient. Although the problem of improving a complex real-time system using ASIP and co-processor technology is not simple, it is not intractable because of the information that can be gained from whole-system profiling, which can then be applied to analysis.

References

[1] Altera Inc. Multi-standard ADPCM co-processor (accessed 20 may 05). http://www.altera.com/-products/ip/dsp/speech_audio_processing/m-amp-multi-std-adpcm.html.

[2] Anonymous. Coprocessor synthesis - increasing SoC platform ROI. White paper, CriticalBlue Corporation, 2002.

[3] ARC International. Integrated profiler (accessed 1 Apr 05). http://www.arc.com/software/-operatingsystems/prototyping/integratedprofiler.html.

[4] ASIP Meister. Home page (accessed 1 Apr 05). http://www.eda-meister.org/asip-meister/.

[5] N. C. Audsley and K. Bletsas. Fixed priority timing analysis of real-time systems with limited parallelism. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS 04)*, Catania, Italy, Jul 2004. IEEE Computer Society.

[6] D. Burger. SimpleScalar tools (acc. 1 Apr 05). http://www.cs.wisc.edu/~mscalar/simplescalar.html.

[7] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 2001.

[8] CAST Inc. IDCT decoder core (accessed 20 may 05). http://www.cast-inc.com/cores/idct/-index.shtml.

[9] CAST Inc. JPEG decoder core (accessed 20 may 05). http://www.cast-inc.com/cores/jpeg-d/-index.shtml.

[10] Eric Young. libdes (accessed 1 Apr 05). http://www.shmoo.com/crypto/.

[11] T. Glökler and H. Meyr. *Design of Energy-Efficient Application-Specific Instruction Set Processors*. Kluwer Academic Publishers, 2004.

[12] B. Grattan, G. Stitt, and F. Vahid. Codesign-extended applications. In *Proc. 10th Int. Symp. Hardware/Software Codesign*, pages 1–6, 2002.

[13] R. K. Gupta and G. D. Micheli. Hardware-software cosynthesis for digital systems. *IEEE Des. Test*, 10(3):29–41, 1993.

[14] Independent JPEG Group. libjpeg (accessed 19 May 05). http://www.ijg.org/.

[15] R. Jejurikar and R. Gupta. Procrastination scheduling in fixed priority real-time systems. In *Proc. LCTES*, pages 57–66, 2004.

[16] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Int. Symp. Microarchitecture*, pages 330–335, 1997.

[17] G. D. Micheli, W. Wolf, and R. Ernst. *Readings in Hardware/Software Co-Design*. Morgan Kaufmann Publishers Inc., 2001.

[18] Mike Bennett. wayv (accessed 1 Apr 05). http://www.stressbunny.com/wayv/.

[19] OAR Corporation. RTEMS (accessed 1 Apr 05). http://www.rtems.com/.

[20] Tensilica Corporation. Accelerating existing C code (accessed 1 Apr 05). http://www.tensilica.com/html/accelerating_existing_c_code.html.

[21] Tensilica Corporation. Xtensa cores excel in EEMBC benchmarks (accessed 1 Apr 05). http://www.tensilica.com/html/technology_eembc.html.

[22] G. K. Wallace. The JPEG still picture compression standard. *Commun. ACM*, 34(4):30–44, 1991.

[23] A. Wang, E. Killian, D. Maydan, and C. Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *DAC '01: Proc. 38th conf. on Design automation*, pages 184–188, 2001.

[24] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. 27th Int. Symp. Computer Architecture*, pages 225–235, 2000.