Software Acceleration Using Coprocessors: Is it Worth the Effort?

Martyn Edwards
UMIST
Department of Computation
P.O. Box 88, Manchester, M60 1QD, United Kingdom
mde@ap.co.umist.ac.uk

Abstract

A commonly accepted technique in hardware/software co-design is to implement as many system functions as possible in software and to move performance critical functions into special-purpose external hardware in order to either satisfy timing constraints or reduce the overall execution time of a program – this is known as “software acceleration”. This paper investigates the limits to the performance enhancements obtainable using software acceleration techniques. A practical target architecture, based on the use of programmable logic, is used to illustrate the problems associated with software acceleration. It is shown that normally little benefit can be obtained by applying software acceleration methods to general-purpose applications. Whereas software acceleration can profitably be used in a limited number of special-purpose applications, a designer would probably be better off developing ASIC components, based on heterogeneous multiprocessor architectures.

1. Introduction

The first premise of software acceleration is that computationally-intensive applications normally spend most of their execution time in a small section of executable code, typically in an inner loop. The performance of such an application can be enhanced by identifying these critical sections of code and transferring their implementations from software to hardware. The second premise is that the performance of a critical code section can be significantly enhanced by executing its functions in hardware, which can be customised to implement the required functionality in an efficient manner. In order to be applicable to a wide range of general purpose applications, the hardware is often assumed to be programmable. This is normally achieved using an FPGA so that new critical sections can be readily and quickly loaded into these devices. Using these types of components, software acceleration methods can also be applied to the implementation of real-time embedded systems, which are typically realised using a generic architecture comprising of conventional programmable processors, memories, and custom co-processors connected via a shared bus. In the design and development of such systems the partitioning of the functionality between hardware and software subsystems becomes a critical issue in order to meet both cost and performance constraints. In this paper, it is assumed that the critical sections of an application’s software have already been identified in some way and equivalent hardware structures synthesised for execution in a customised co-processor [2, 6, 8, 9]. The hardware generation process is either performed manually or automatically using high-level synthesis tools. A key issue in both approaches is the amount of hardware optimisation, in terms of the implementation of pipelining techniques and exploitation of parallelism, that can be applied to a design.

Section 2 reviews Amdahl’s law, which can be used to measure the increase in performance that may be achievable by implementing a critical section of software in hardware. Andahl’s law, therefore, allows the validity of the above two basic premises to be evaluated. Section 3 surveys the work of other researchers in the area of software acceleration and reviews their performance results. The related work undertaken at UMIST, based on a commonly used target architecture, is examined in Section 4 to assess the general applicability of software acceleration as a viable form of hardware/software co-design. This paper argues that, in general, the implementation of software acceleration techniques is not worth the effort in terms of the small advantages that are accrued.

2. Application of Amdahl’s law

Before committing a system function to hardware it is advisable to determine the performance enhancement that may be gained by implementing the function in this way. The degree of software acceleration obtained can be used to establish whether or not it is worthwhile implementing the original software-based function in hardware. We can employ Amdahl’s Law [1] to give an indication of this
enforcement. Amdahl's Law states that

\[
T = \frac{1}{(1 - \alpha) + \frac{\alpha}{s}}
\]  

(1)

where \(\alpha\) is the fraction of the execution time of the original program that could be enhanced by implementing the corresponding software-based function in hardware, \(s\) is the speedup obtained by executing that fraction in hardware (that is, the execution time of \(\alpha\) in software divided by its execution time in hardware), and \(T\) is the overall speedup of the original program obtained by implementing \(\alpha\) in hardware. Figure 1 gives a plot of \(T\) versus \(\alpha\) for different values of \(s\).

![Graph](image)

**Figure 1. Amdahl's law I: T versus \(\alpha\)**

Figure 1 indicates that the overall speedups obtained quickly decrease as the code fraction \(\alpha\), implemented in hardware, is reduced irrespective of the corresponding value of \(s\). In fact, in the limit the maximum overall speedup obtainable (max\(T\)) is \(1/(1 - \alpha)\). For example, if 90\% of the execution time of the original program can be transferred to hardware, the maximum overall speedup is 10 times the original software execution time.

This limit on the maximum values of \(T\) for given values of \(\alpha\) suggests that it is not worthwhile generating large values for \(s\) if the code fraction \(\alpha\) is not large – certainly if \(\alpha\) is much less than 0.9! If we take the working assumption that the objective of a viable general-purpose software acceleration method should be to achieve an overall speedup of 10 times the original software execution time, then it is possible to determine the associated values for \(s\) given values of \(\alpha\) greater than 0.9, as shown in Figure 2.

The graph indicates that if \(\alpha = 0.975, 0.95\) and 0.925 then \(s = 13, 19\) and 37, respectively. If \(\alpha < 92\%\) then the required speedup \(s\) may be too large to be easily achieved using programmable logic components; for example if \(\alpha = 0.915\) then \(s = 61\) and if \(\alpha = 0.905\) then \(s = 181\). In such cases, the corresponding decrease in \(\alpha\) is not significant given the large increases required in \(s\) to maintain the overall speedup equal to 10.

![Graph](image)

**Figure 2. Amdahl's law II: \(s\) versus \(\alpha\)**

Amdahl's law gives a very pessimistic view of the possibilities for software acceleration unless it is possible to find software-based functions which give large values for \(\alpha\).

3. Software Acceleration Studies

Athanas and Silverman [2] describe the PRISM approach where an application is specified as a C program, which is manually partitioned into hardware and software subsystems. The C functions which have the greatest impact on the overall performance of an application are suitable candidates for implementation in an FPGA co-processor. They use conventional automatic hardware synthesis techniques and start with the C functional description of the chosen candidate(s). The target architecture consists of a Motorola 68010 and an array of Xilinx 3090 FPGAs. Experimental results indicate that the speedup of the enhanced fraction of an application can vary between 2.9 and 54 with an average of about 23. It must be noted that the highest speedups were obtained for functions which had trivial hardware implementations, for example, a simple wiring assignment of input to output pins. For a discrete event simulator example, a code section was identified which accounted for 90\% of the software execution time. An overall speedup of 6.7 times was obtained, giving the code section a speedup of 18.

Wazlowski, et al [13] describe an updated architecture for PRISM based on an AMD 29050 and an array of Xilinx 4010 FPGAs. In this case, experimental results indicate that the speedup of an enhanced fraction can vary between 6.7 and 86.3 with an average of about 30.

Bertin, et al [3] constructed the Perle-0 system, which
is a general-purpose co-processor coupled to a Motorola 68020. Perle-0 consists of a 5 x 5 array of Xilinx 3020 FPGAs connected to static RAM – a later version Perle-1 contained Xilinx 3090 FPGAs. The authors accelerate a software application by executing its “inner loop” in an appropriate hardware circuit in Perle. The identification of the inner loop and its hardware realisation are performed manually. For a range of different applications [4] they consistently obtained overall speedups of about 10 times faster than the corresponding software versions. The best results were achieved with highly parallel and, where possible, pipelined hand-crafted hardware solutions.

Braige and Madsen [5] discuss the acceleration of a computer graphics application (cylinder rotation) written in C. An analysis of the program execution characteristics indicated that vector arithmetic operations accounted for 34% of the total software execution time. A special-purpose ASIC co-processor was proposed to implement a 3D vector arithmetic unit. An estimated overall speedup of 1.5 may be obtained with a SPARC and the co-processor which implies the speedup of the code section was approximately 14.

Ernst, et al [8] specify the behaviour of a system as a set of tasks in the language C*, which is a superset of ANSI C. They automatically partition a system into hardware and software sub-systems, where the computation time intensive parts are implemented in a hardware co-processor. They use automatic synthesis techniques to generate the co-processor which are based on conventional high-level synthesis methods. The authors cite a chromakey example where two code sections, which account for 90% of the software execution time on a SPARC processor, are executed in the co-processor (an ASIC). An overall speedup of 3 times was achieved, which implies that the speedup of the two code sections was about 3.9.

Jantsch, et al [10] have examined the characteristics of seven test C programs, which range from data flow oriented, computation intensive applications to ones which operate on data streams. They use a SPARC and an FPGA co-processor, connected via shared memory. Estimates of the overall speedups obtained by executing the test program code sections in the FPGA range from 1 to 14.4. As expected, the best acceleration figures are obtained with the data flow type applications. The authors discount all candidates for hardware implementation that account for less than 10% of the software execution time!

Theißinger, et al [12] describe the software acceleration of a data compression program, Gzip, using a SPARC processor and an application-specific co-processor. Analysis of the C source program indicated that one function accounted for 69% of the total software execution time. By synthesising this function in hardware its speedup was estimated to be 2.2, which gives an overall speedup of 1.6.

Knapp [11] argues that a number of computationally intensive functions commonly found in digital signal processing applications, for example, multiply-accumulate, can be completely implemented in Xilinx FPGAs to obtain significant performance enhancements. Knapp estimates that a Xilinx XC4013E-2 can implement an 8-bit, 16-tap FIR filter 22 times faster than a single 50 MHz DSP processor and almost 100 times faster than a 133 MHz Pentium processor. This performance improvement is obtained by hand-crafting a highly parallel hardware implementation. In a second application, a Viterbi decoder, the performance of a DSP processor with an FPGA coprocessor was nearly 3 times faster than a software implementation using two DSP processors. Again, the nature of the algorithm resulted in a highly parallel implementation in the FPGA coprocessor.

From an analysis of the published results, the use of software acceleration techniques is rather disappointing where general purpose applications have been examined. Some success has been obtained where it is necessary to speed up a code section in order to satisfy performance constraints, so long as the required performance increase is small. The only significant results are where complete applications are transferred to hardware (α = 1) or where the hardware implementations of code sections are trivial. It would appear that current automatic hardware synthesis techniques do not tend to generate highly parallel circuits, thereby not exploiting the full potential of any hardware-based co-processor.

4. UMIST Software Acceleration Studies

A number of test applications have been evaluated on the hardware/software co-design development environment at UMIST [6, 7]. In our approach, an application written in C/C++ is interactively partitioned into hardware and software subsystems. The most computation time intensive code section is realised in a hardware co-processor, which is implemented in a Xilinx 4010 FPGA. Automatic, high-level hardware synthesis techniques are used to generate the co-processor circuit. An outline of the system target architecture is given in Figure 3, where the co-processor is connected to a Motorola 68332 processor and communication takes place via a shared memory and memory-mapped registers in the FPGA.

Two simple test programs produced overall speedups of 1.4 and 3.1. The first example was “memory intensive” (processing a 7000 element array in shared memory) and the second was computation intensive. The results indicated that accessing shared memory has a significant negative effect on the speedups obtainable for an application – this phenomenon is architecture dependent and has been noted by other researchers [8, 10]. We experimented with further, more typical, test programs
based on well known data compression, image compression and data encryption algorithms. The results are presented below.

**Figure 3. Target architecture**

For the LZ77 data compression algorithm, a code section was identified which accounted for 71% of the software execution time. Unfortunately, the synthesised hardware was too large to fit on the XC4010 (150% of the available CLBs were needed), which indicates a major drawback of any fixed target architecture. The original code section was, however, partitioned into smaller units and one of these, which accounted for 38.7% of the software execution time, was transferred to hardware. An overall speedup of 1.5 times was obtained (maxT = 1.63), giving the code section a hardware speedup of 7.2.

For the Huffman algorithm, a code section was identified which accounted for 57.2% of the software execution time for data decompression. Unfortunately, an overall speedup of 0.95 times was obtained. This 'slowing down' effect was due to a very high memory access rate, including parameter passing, and a low computational complexity. This resulted in the total shared memory access time being far greater than the computation time for the function.

For the JPEG image compression algorithm, using fixed-point arithmetic, the forward 'discrete cosine transform' accounted for 28.4% of the software execution time. Unfortunately, once again the synthesised hardware was too large to fit onto the FPGA. There were no other candidate functions which accounted for a sufficiently large percentage of the execution time to warrant implementation in the co-processor. For example, the next most computationally expensive code section was the 'quantise' algorithm which accounted for only 9.1% of the execution time.

Data encryption algorithms produced similar results. In the case of the DES encryption algorithm, a code section accounted for 64.3% of the software execution time. This function was too large to fit onto the FPGA (171% of the available CLBs were needed). Partitioning the function into smaller ones resulted in speedups of less than one being obtained. Again the total memory access time for the synthesised functions was greater than the corresponding computation time. In the RSA algorithm, one code section accounted for 64.2% of the software execution time. This function proved to be unsynthesisable in the FPGA due to its computational complexity - multi-precision integer arithmetic was required.

We have found it very difficult to find applications where a large percentage of the software execution time is expended in a function(s) which can be successfully implemented in our co-processor. Even if we increased the size or number of FPGAs it is probable that we would still not achieve a significant speedup due to the inherent limitations in this type of target architecture. The main overheads are the time taken to pass parameters between the software and hardware components, and the time taken to access the shared memory by the coprocessor. The reduction in performance observed in some applications is due to these overheads. For example, for a hardware-based function, let

\[
\begin{align*}
T_p & = \text{time to transfer parameters from/to the FPGA} \\
& \quad \text{(10 clock cycles per parameter)} \\
T_m & = \text{additional time required for the FPGA to access} \\
& \quad \text{the shared memory (9 clock cycles per memory access)} \\
T_{hw} & = \text{time to execute the function in the FPGA,} \\
& \quad \text{excluding the shared memory accesses} \\
T_{sw} & = \text{time to execute the function in software,} \\
& \quad \text{including shared memory accesses}
\end{align*}
\]

It is apparent that it is only viable to move a function from software to hardware if

\[
T_{sw} \gg p.T_p + m.T_m + T_{hw} \tag{2}
\]

where \( p = \) number of parameters, \( m = \) number of shared memory accesses.

If \( T_{sw} \gg T_{hw} \), then if \( (p.T_p + m.T_m)/T_{hw} \) is greater than 1, then it is probably not worth the effort to implement the chosen function in hardware. This will normally be the case for computationally simple functions with a lot of shared memory accesses and parameters.
5. Conclusions

The main conclusion must be that general-purpose software acceleration techniques do not provide any significant performance advantages over 'pure' software solutions when using a single processor/co-processor model. In fact, designers would be better advised to use a processor that is better tuned to their application. For example, either employ a faster processor or design an application-specific instruction set processor (ASIP). Data dominated applications should give greater speedups than control dominated examples - perhaps DSPs should be used in signal processing applications where the multiply-accumulate operation has been internally enhanced.

Our work has also indicated that it is essential to have an extensible architecture that is essentially modular and expandable to cope with large scale applications. Our current research is targeted at heterogeneous multiple processor architectures, starting from a high-level implementation-independent object-based model. The intention is to implement functions in hardware only to meet performance constraints rather than simply speed up the application.

6. Acknowledgements

I would like to thank colleagues in the Department of Computation for their assistance with this work.

7. References