Rapid Configuration & Instruction Selection for an ASIP: A Case Study

Newton Cheung
School of Computer Science & Engineering
University of NSW, Sydney
ncheung@cse.unsw.edu.au

Jörg Henkel
NEC Laboratories America
4 Independence Way
Princeton, NJ 08540
henkel@nec-labs.com

Sri Parameswaran
School of Computer Science & Engineering
University of NSW, Sydney
sridevan@cse.unsw.edu.au

Abstract

We present a methodology that maximizes the performance of Tensilica based Application Specific Instruction-set Processor (ASIP) through instruction selection when an area constraint is given. Our approach rapidly selects from a set of pre-fabricated coprocessors/functional units from our library of pre-designed specific instructions (to evaluate our technology we use the Tensilica platform). As a result, we significantly increase application performance while area constraints are satisfied. Our methodology uses a combination of simulation, estimation and a pre-characterised library of instructions, to select the appropriate co-processors and instructions. We report that by selecting the appropriate coprocessors/functional units and specific TIE instructions, the total execution time of complex applications (we study a voice encoder/decoder), an application's performance can be reduced by up to 85% compared to the base implementation. Our estimator used in the system takes typically less than a second to estimate, with an average error rate of 4% (as compared to full simulation, which takes 45 minutes). The total selection process using our methodology takes 3-4 hours, while a full design space exploration using simulation would take several days.

1 Introduction

Embedded system designers face design challenges such as reducing chip area, increasing application performance, reducing power consumption and shortening time-to-market. Traditional approaches, such as employing general programmable processors or designing Application Specific Integrated Circuits (ASICs), do not necessarily meet all design challenges. While general programmable processors offer high programmability and lower design time, they may not satisfy area and performance challenges. On the other hand, ASICs are designed for a specific application, where the area and performance can easily be optimised. However, the design process of ASICs is lengthy, and is not an ideal approach when time-to-market is short. In order to overcome the shortcomings of both general programmable processors and ASICs, Application Specific Instruction set Processors (ASIPs) have become popular in the last few years.

ASIPs are designed specifically for a particular application or a set of applications. Designers of ASIPs can implement custom-designed specific instructions (custom-designed specific functional units) to improve the performance of an application. In addition, ASIP designers can attach pre-fabricated coprocessors (such as Digital Signal Processing Engines and Floating-Point units) and pre-designed functional units (such as Multiplier-Accumulate units, shifters, multipliers etc). They can also modify hardware parameters of the ASIPs (such as register file size, memory size, cache size etc).

As the number of coprocessors/functional units increase and more specific instructions are involved in an application, the design space exploration of ASIPs takes longer. Designers of ASIPs require an efficient methodology to select the correct combination of coprocessors/functional units and specific instructions. Hence, the design cycle and chip area is reduced and an application performance is maximized.

Research into design approaches for ASIPs has been carried out for about ten years. Design approaches for extensible processors can be divided into three main categories: architecture description languages [3] [4] [13] [16] [20]; compilers [5] [8] [18] [21] and methodologies for designing different aspects of extensible processors [7] [9].

The first category of architecture description languages for ASIPs is further classified into three sub-categories based on their primary focus: the structure of the processor such as the MIMOLA system [17]; the instruction set of the processor as given in nML [6] and ISDL [11]; and a combination of both structure and instruction set of the processor as in HMDES [10], EXPRESSION [12], LISA [13], PEAS-III (ASIP-Meister) [16], and FlexWare [19]. This category of approach generates a retargetable environment, including retargetable compilers, instruction set simulators (ISS) of the target architecture, and synthesizable HDL models. The generated tools allow valid assembly code generation and performance estimation for each architecture described (i.e. "retargetable")

In the second category, the compiler is the main focus of the design process using compiling exploration information such as data flow graph, control flow graph etc. It takes an application written in a high-level description language such as ANSI C or C++, and produces application characteristic and architecture parameter for extensible processors. Based on these application characteristics, an application specific processor for that particular application can be constructed. In [21], Zhao used static resource models to explore possible functional units that can be added to the data path to enhance performance. Onion in [18] proposed a feed-back methodology for an optimising compiler in the design of an extensible processors, so more information is provided at the compile stage of the design cycle producing a better hardware extensible processors model.

In the third category, estimation and simulation methodologies are used to design extensible processors with specific register file sizes, functional units and coprocessors. Gupta et al. in [9] proposed a processor evaluation methodology to quickly estimate the performance improvement when architectural modifications are made. However, their methodol-

1530-1591/03 $17.00 © 2003 IEEE

ogy does not consider an area constraint. Jain [15] proposed a methodology for evaluating register file size in an extensible processors design. By selecting an optimum register file size, they are able to reduce the area and energy consumption significantly.

Our design flow fits between second and third categories of the design approaches given above. In this paper, we propose a methodology for extracting specific instruction from an application to build an instruction library and a heuristic algorithm for selecting pre-fabricated coprocessors/functional units and pre-designed (from our library) specific instructions to construct an extensible processors. Hence, the extensible processor achieves maximum performance within the given area constraint. There are four inputs of our design flow: an application written in C/C++, a set of pre-fabricated coprocessors/functional units, a specific instructions library and an area constraint. In addition, another feature of the design flow is a performance estimator of the configured extensible processor.

Our methodology for extracting specific instruction consists of a probability model for selecting program section, implements the program section as hardware, and characterizes the instruction. Hence, an instruction library is build. To our best knowledge, there is no formal methodology described on how to extract program section from an application program which constructs an instruction library.

Our heuristic algorithm is closely related to Gupta et al. in [8] and IMSP-2P-MIFU in [14]. Gupta et al proposed a methodology, which through profiling an application written in C/C++, using a performance estimator and an architecture exploration engine, to obtain optimal architectural parameters. Then, based on the optimal architectural parameters, they select a combination of four pre-fabricated components, being a MAC unit, a floating-point unit, a multi-ported memory, and a pipelined memory unit for an extensible processor when an area constraint is given. Alternatively, the authors in IMSP-2P-MIFU proposed a methodology to select specific instructions using the branch and bound algorithm when an area constraint is given.

There are three major differences between the work at Gupta et al. in [8] & IMSP-2P-MIFU in [14] and our work. Firstly, Gupta et al. only proposed to select pre-fabricated components for extensible processors. On the other hand, the IMSP-2P-MIFU system is only able to select specific instructions for extensible processors. Our methodology is able to select both pre-fabricated coprocessors/functional units and pre-designed specific instructions for extensible processors. The second difference is the IMSP-2P-MIFU system uses the branch and bound algorithm to select specific instructions, which is not suitable for large applications due to the complexity of the problem. Our methodology uses performance estimation to determine the best combinations of coprocessors/functional units and specific instructions in an extensible processor for an application. Although Gupta et al. also use a performance estimator, they require exhaustive simulations between the four pre-fabricated components in order to select the components accurately. For our estimation, the information required is the hardware parameters and the application characteristics from initial simulation of an application. Therefore, our methodology is able to estimate the performance of an application in a short period of time. *The final difference is that our methodology includes the latency of additional specific instructions into the performance estimation, where IMSP-2P-MIFU in [14] does not take this factor into account. This is very important since it eventually decides of the usefulness of the instruction selection when implemented into a real-world hardware.*

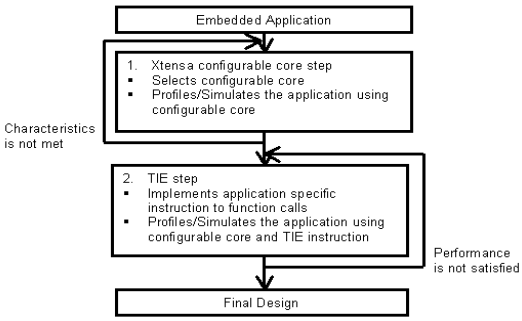

Figure 1. Xtensa application's design flow.

The rest of this paper is organized as follows: section 2 presents an overview of the Xtensa and its tools; section 3 describes the design methodology on configurable core and functional units; section 4 describes the DSP application used in this paper; section 5 presents the verification method and results. Finally, section 6 concludes with a summary.

2 Xtensa Overview & Tools

Xtensa is a configurable and extendable processor developed by Tensilica Inc. It allows designers to configure their embedded applications by constructing configurable core and designing application specific instructions using Xtensa software development tools. The project described in this paper used the Xtensa environment. Figure 1 shows the design flow of the Xtensa processor. In this section, we describe constructing configurable cores, designing specific instructions and the Xtensa tools in detail.

The work carried out and the methodology developed, however, is general and could have been conducted with any other similar reconfigurable processor platform.

Xtensa's configurable core can be constructed from the base instruction set architecture (ISA) by selecting the Xtensa processor configuration options such as the Vectra DSP Engine, floating-point unit, 16-bit Multiplier etc. The quality of the configuration is dependent on the design experience of the designer who analyses an application. Our methodology tries to reduce this dependence based on quick performance estimation of the application. As this paper uses a speech recognition application as a case study, it is necessary to look at the Vectra Digital Signal Processing (DSP) Engine and Floating-Point (FP) Unit closely. As DSP applications involve computational intensive algorithms such as in filter design, convolution and the FFT algorithm, most of the DSP processors have multiple functional units such as the ALU, shifter and multiplier that operate in parallel in order to achieve a fast computation time. The Vectra DSP Engine is a fully configurable, single instruction multiple data (SIMD) coprocessor with additional instructions, which targets parallel computational intensive algorithm. In the Vectra DSP Engine, there are five configurations with different register widths, different number of registers, and different functional units, which are able to handle a wide range of DSP applications. The floating-point unit provides single precision (32-bit) operation and extended single precision (48-bit) operation with 52 additional instructions such as multiply and accumulate (MAC), floating-point addition, floating-point subtraction, floating-point multiplication etc. However, it does not provide parallel computation as mentioned for the Vectra DSP engine. Moreover, in the T10xx.x Xtensa configuration, six instructions are not implemented, being division, reciprocal, reciprocal square root, square root

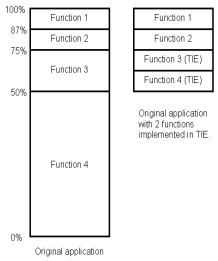

Figure 2. An example of application with 2 TIE.

and read/write floating-point status registers. This is because there has not been a significant demand for these features, especially for Xtensa customers in the embedded market, particularly, since these instructions can be fairly area intensive. Therefore, if an embedded application uses these unimplemented instructions heavily, then it is ideal to implement these instructions using the Tensilica Instruction Extensions.

The second part of designing the Xtensa processor is by using Tensilica Instruction Extensions (TIE). "Tensilica Instruction Extension (TIE) is a language that lets designers incorporate application specific functionality in the processor by adding new instructions." [7]. The main idea of TIE language is to design a specific functional unit to handle a specific functionality that is heavily used in the application, and hence this functional unit can lead to higher performance of the application. Figure 2 shows an example, which shows the percentage of execution time consumed by four functions and how TIE instructions can lead to higher performance of an application. In this example, function 4 consumes 50% of the execution time of an application and the second one (function 3) consumes 25% of the time. When both TIE instructions for function 3 and function 4 are implemented for this application, the execution time is able to reduce to half of the original execution time.

TIE language is a language that generates Verilog or VHDL code when compiled. The Verilog/VHDL code can then be put through the Synopsys tool Design Compiler to obtain timing and area.

However, adding TIE instructions may incur an increase in the latency of the processor. If this is the case, clock frequency must be slowed in order to compensate for the addition of TIE instructions. Since the simulator only considers the cycle-count, it would mislead the real performance of the application when the latency is increased. The real performance of an application should be the number of cycle-count multiplied by the latency caused by TIE instructions. Therefore, our methodology reinforces this critical point in our selection process of TIE instructions. For more information on TIE language, a detailed description can be found in [1] [2].

During the construction of configurable cores and the design of specific instructions, profiling (to get the application characteristics) and simulation (to find out the performance for each configuration) are required. In order to efficiently obtain the characteristics and performance information, Xtensa provides four software tools: a GNU C/C++ compiler, Xtensa's Instruction Set Simulator, Xtensa's profiler and TIE compiler. A detailed description about embedded applications using Xtensa processor can be found in [7].

3 Methodology

Our methodology consists of minimal simulation by utilizing a greedy algorithm to rapidly select both pre-fabricated coprocessors/functional units and pre-designed specific TIE instructions for an ASIP. The goal of our methodology is to

Notation	Descriptions
$Area\_Core_i$	area in gate for processor i
$Speedup\_TIE_{ij}$	speedup ratio of TIE instruction j in processor i
$Area\_TIE_j$	area in gate of TIE instruction j
P_{ij}	percentage of cycle-count for function j in processor i
CC_i	total cycle-count spent in processor i
$Clock\_Period_i$	clock period of processor i
$Latency_j$	latency of TIE instruction j
$Selected_i()$	array stores selected TIE instructions in processor i

Table 1. Notations for algorithm

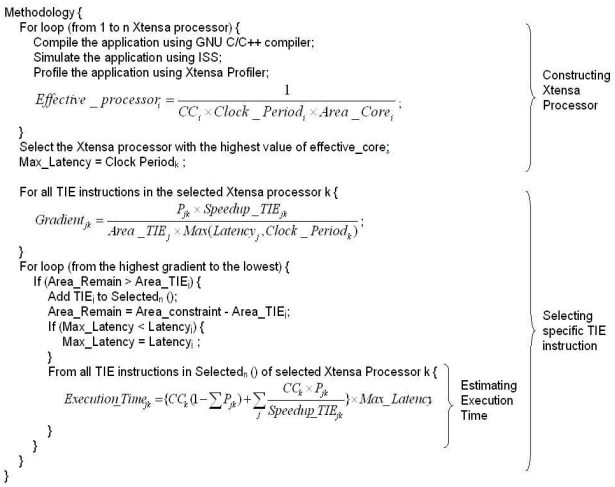

Figure 3. The algorithm

select coprocessors/functional units (these are coprocessors which are supplied by Tensilica) and specific TIE instructions (designed by us as a library), and to maximize performance while trying to satisfy a given area constraint. Our methodology divides into three phases: a) selecting a suitable configurable core; b) selecting specific TIE instructions; and c) estimating the performance after each TIE instruction is implemented, in order to select the instruction. The specific TIE instructions are selected from a library of TIE instructions. This library is pre-created and pre-characterised.

3.1 Assumptions

All TIE instructions are mutually exclusive with respect to other TIE instructions (i.e., they are different). This is because each TIE instruction is specifically designed for a software function call with minimum area or maximum performance gain.

Speedup/area ratio of a configurable options is higher than the speedup/area ratio of a specific TIE instruction when the same instruction is executed by both (i.e., designers achieve better performance by selecting suitable configurable options, than by selecting specific TIE instructions). This superior performance is achieved because configurable options are optimally designed by the manufacturer for those particular instruction sets, whereas the effective selection of TIE instructions is based on designers' experience.

3.2 Algorithm

There are three sections in our methodology: selecting Xtensa processor with different configurable co-processor core options, selecting specific TIE instructions, and estimating the performance of an application after each TIE instruction is implemented. Firstly, minimal simulation is used to select an efficient Xtensa processor that is within the area constraint. Then with the remaining area constraint, our

methodology selects TIE instructions using a greedy algorithm in an iterative manner until all remaining area is efficiently used. Figure 3 shows the algorithm and three sections of the methodology and the notation is shown in table 1.

Since the configurable coprocessor core is pre-fabricated by Tensilica and is associated with an extended instruction set, it is quite difficult to estimate an application performance when a configurable coprocessor core is implemented within an Xtensa processor. Therefore, we simulate the application using an Instruction Set Simulator (ISS) on each Xtensa processor configuration without any TIE instructions. This simulation does not require much time as we have only a few co-processor options available for use. We simulate a processor with each of the coprocessor options.

Through simulation, we obtain total cycle-count, simulation time, and a call graph of the application for each Xtensa processor. Then we calculate the effectiveness of the processor for this application by considering total cycle-count, clock period, and the area of each configurable processor. The effectiveness of the Xtensa processor *i*, $Effective\_processor_i$, is defined as:

$$Eff\_proc_i = \frac{1}{CC_i \times Clock\_Period \times Area\_Processor_i} \quad (1)$$

This factor indicates a processor is most effective when it has the smallest chip size with the smallest execution time (cycle-count CC multiplied by the clock period) of an application. This factor calculates the ratio of performance per area for each processor, and our methodology selects the processor with the highest performance per area ratio that falls within the area constraint. This factor may not select the processor with the best performance. Note, that "Area_Core" is not only an optimization goal but may also be a (hard) constraints.

Next, the methodology focuses on selecting specific TIE instructions. The selection of a specific TIE instruction is based on the area, speedup ratio, latency (maximum clock period in all TIE instructions and configured processor), and percentage of total cycle-count for the TIE instruction. We define the gradient of TIE instruction *j* in Xtensa processor *k* as:

$$Gradient_{jk} = \frac{P_{jk} \times Speedup\_TIE_{jk}}{Area\_TIE_j \times Max(Latency_j, Clock\_Period_k)} \quad (2)$$

This factor indicates that the performance gain per area of an application when the TIE instruction is implemented in the extensible processor. For example, if an application spent 30% of the time calling function X and 20% of the time calling function Y, two TIE instructions, say TIE_X and TIE_Y, are implemented to replace software function call X and software function call Y. The speedup of TIE instructions is 4 times and 6 times for TIE_X and TIE_Y respectively, and the area for the implemented TIE instructions are 3000 gates and 4000 gates respectively. The latency value (the clock period) for both instructions is the same. The gradient of functional units is 0.0004 and 0.0003 for TIE_X and TIE_Y. This means that TIE_X will be considered for implementation in the processor before TIE_Y. This example shows that the overall performance gain of an application will be the same whether TIE_X or TIE_Y is implemented. Since TIE_X has a smaller area, TIE_X is selected before TIE_Y. Equation 2 uses the remaining area effectively by selecting the specific TIE instruction with largest performance improvement per area. This is a greedy approach to obtain the largest performance improvement per area. This greedy approach proved to be effective and efficient in our application sets.

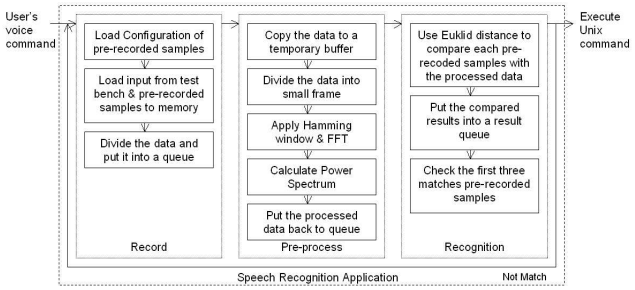

Figure 4. Speech Recognition Application.

In the last section, after TIE instructions are selected, an estimation of execution time is calculated. The execution time estimation (ETE) for an extensible processor *k* with a set of selected TIE instruction (from 1 to *j*) is defined as:

$$ETE_{jk} = (CC_k(1 - \sum_j P_{jk}) + \sum_j \frac{CC_k \times P_{jk}}{Speedup\_TIE_{jk}}) \times Latency_{max} \quad (3)$$

where CC_k is the original total cycle-count of an application running on an Xtensa processor k. The first part of the equation calculates the cycle-count of an application, then it multiples with the maximum latency between TIE instructions. Maximum latency is the maximum clock period value of all the TIE instructions and configured processor.

4 Speech Recognition Software & Analysis

We use a speech recognition application as a case study to demonstrate the effectiveness of our approach. This application provides user voice control over Unix commands in a Linux' shell environment. It employs a template matching based recognition approach, which requires the user to record at least four samples for each Unix command that he/she wants to use. These samples are recorded and are stored in a file. Moreover, since this software involves a user's voice, for the sake of consistency, a test bench is recorded and is stored in another file.

The application consists of three main sections: record, pre-process, and recognition. In the "record" section, it first loads the configuration of the speaker, then it loads the pre-recorded test bench as inputs, and the pre-recorded samples into memory. After that, it puts the test bench into a queue passing through to the next section. In the "pre-process" section, it copies the data from the queue to a buffer in order to divide the data into frame size segments. Then, it performs a filtering process using the Hamming window and applies single-precision floating-point 256-point FFT algorithm to the input data to minimize the work done in the following recognition section. Afterwards, it calculates the power spectrum of each frame and puts these frames back into the queue. Finally, in the "recognition" section, it implements the template matching approach utilizing Euclid's distance measure in which it compares the input data with the pre-recorded samples that are loaded in memory during the "record" section. In the next step, it stores the compared results in another queue. If the three closest matches of pre-recorded samples are the same command, then the program executes the matched Unix command. However, if the input data does not match the pre-recorded samples, the program goes back to the "record" section to load the input again, and so on. Figure 4 shows the block diagram of speech recognition application.

As this application involves a human interface and operates in sequence, it is necessary to set certain real-time con-

Function	Description	P1	P2	P3
Sqrtf	Single precision square root	1.3%	2.3%	13.4%
Mod3	Modular 3	0.2%	0.6%	3.0%
Logf	Single precision natural logarithm	2.7%	2.6%	2.4%
Divf	Single precision division	0.8%	1.8%	7.3%
Addf	Single precision addition	13.6%	25.1%	10.0%
Multf	Single precision multiplication	58.3%	31.1%	16.3%

Table 2. Percentage of time spent for functions

Xtensa Processor	P1	P2	P3
Area(mm2)	1.08	4.23	2.28
Area(gates)	35,000	160,000	87,000
Power(mW)	54	161	108
Clock Rate(MHz)	188	158	155
Simulation Time (sec)	2770.93	1797.34	641.69
Cycle-Count	422,933,748	390,019,604	131,798,414

Table 3. Hardware Cost and initial simulation

straints. For example, it is assumed that each voice command should be processed within 1 second after the user finished his/her command. So, "record" section should take less than 0.25s to put all the data into a queue. While "preprocess" and "recognition" should consume less than 0.5s and 0.25s respectively. Through a profiler we analysed the speech recognition software in a first approximation. Table 2 shows the percentage of selected software functions that are involved in this application in different Xtensa processors configurations that we denote as P1, P2 and P3. P1 is an Xtensa processor with the minimal configurable core options. P2 is a processor with Vectra DSP Engine and its associated configurable core options. P3 is a processor with a floating-point unit and its associated configurable core options (more configurations are possible, but for the sake of efficacy, only these three configurations are shown here). Moreover, in Table 4, the application spent 13.4% of time calling the single precision square root function in Xtensa processor P3.

This application simulates with a cycle-count of 422,933,748 and an instruction-count of 338,079,172 in an Xtensa processor with minimal configurable core options and with no additional instructions. The simulation time of this application is approximately 46 minutes and thus is a quite time consuming. The application consists 20 source files written in ANSI C, which include 50 subroutines and about 3500 lines of code. The compiled source code size is 620 Kbytes.

5 Verification methodology & Results

We have pre-configured three Xtensa processors, namely P1, P2 and P3, for this case study as explained above. As mentioned before, although Xtensa can be configured with other hardware parameters such as register file size, cache size etc, in our case, all processors are configured with the same register file size, cache size etc. We considered constructing an Xtensa processor P4 with DSP Engine and FP unit together, but this proved to be inefficient. The hardware cost and initial simulation result for the speech recognition application for each of these processors are shown in Table 3.

Function	TIE instruction/s	Area (gates)	Speedup factor P1	Speedup factor P2	Speedup factor P3	Latency (ns)
FP Addition	FA32	32000	↑ 8.14×	↑ 8.31×	↓	8.50
FP Division (1)	FFDIV	53800	↑ 17.4×	↓	↑ 15.9×	14.6
FP Division (2)	MANT,LP24,COMB	6800	↑ 6.48×	↑ 3.52×	↑ 5.28×	6.80
FP Multiplication	FM32	32000	↑ 11.6×	↑ 8.98×	↓	7.10
Natural Logarithm	FREXPLN	3300	↓	↓	↑ 1.10×	6.90
Modular 3	MOD3	5500	↑ 10.9×	↓	↑ 17.0×	6.40
Square root (1)	LDEXP,FREXP	3300	↓	↓	↑ 3.30×	7.00
Square root (2)	LDEXP	1100	↓	↓	↑ 2.50×	6.50
Square root (3)	FREXP	3200	↓	↓	↑ 1.90×	6.90

Table 4. Function unit's information

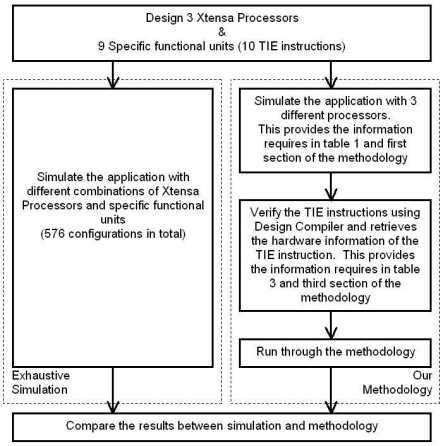

Figure 5. Verification methodology

Nine specific functional units (ten TIE instructions) are implemented and the corresponding information such as area in gates, and the speedup factor under each Xtensa processor are shown in Table 4. The down arrow in Table 4 represents a negative speedup, whereas the up arrow with a number, N, represents that the corresponding specific functional unit is N times faster than the software function call. Different sets of specific functional units are combined into a total of 576 different configurations representing the entire design space of the application. Figure 5 shows the verification methodology.

In order to verify our methodology, we pre-configured the 3 Xtensa processors and the 9 specific functional units (10 TIE instructions) at the beginning of the verification. Then we simulated the 576 different configurations using the ISS. We applied our methodology (to compare with the exhaustive simulation) to obtain the result under an area constraint. During the first phase of the methodology, just the Xtensa processor is selected without the TIE instructions. Table 3 provides the area, cycle-count, and clock rate for an application with each Xtensa processor. This information is used in equation 1 of the methodology. The information from Table 2 and Table 4 is retrieved from running initial simulation and verifying TIE instructions. The information provided in Table 3 is then used in the selection of TIE instructions. For equation 3, all the parameters are obtained from Table 2, Table 3, and Table 4. At the end, we compare results from simulation to our methodology.

5.1 Results

The simulation results of 576 configurations are plotted in Figure 6 with execution time verse area. The dark squares are the points obtained using our methodology for differing area constraints. Those points selected by our methodology corresponded to the Pareto points of the design space graph. As mentioned above, the 576 configurations represent the entire design space and these Pareto points indicate the fastest execution time under a particular area constraint. Therefore, there are no extra Pareto points in this design space. Moreover, Table 4 shows the information of all Pareto points such as area in gates, simulated execution time in seconds, estimated execution time in seconds, latency, selected Xtensa processor and the replaced software function calls. Table 5 also indicates the estimation of application performance for each configuration is on average within 4% of the simulation result. For the fastest configuration (configuration 9 in Table 5), the application execution time is reduced to 85% of

Configuration	Area (gates)	Simulated execution time (sec.)	Estimated execution time (sec.)	Percentage difference	Latency of the processor	Xtensa Processor selected	Software Function replaced
1	35,000	1.8018	–	–	5.32	P1	–
2	67,000	1.0164	1.1233	10.5	7.1	P1	Floating-point multiplication
3	72,500	1.0147	1.1188	10.2	7.1	P1	Modular 3, Floating-point multiplication
4	87,000	0.2975	–	–	6.45	P3	–
5	88,100	0.2738	0.2718	0.7	6.5	P3	Square root (2)
6	93,600	0.2670	0.2694	2.1	6.5	P3	Square root (1), Natural log
7	100,400	0.2632	0.2651	1.9	6.8	P3	Square root (1), Natural log, Floating-point division (2)
8	103,700	0.2614	0.2642	1.0	6.9	P3	Square root (2), Modular 3, Natural log, Floating-point division (2)
9	105,900	0.2586	0.2638	2.0	7.0	P3	Square root (1), Modular 3, Natural log, Floating-point division (2)
Average	–	–	–	4%	–	–	–

Table 5. Pareto points

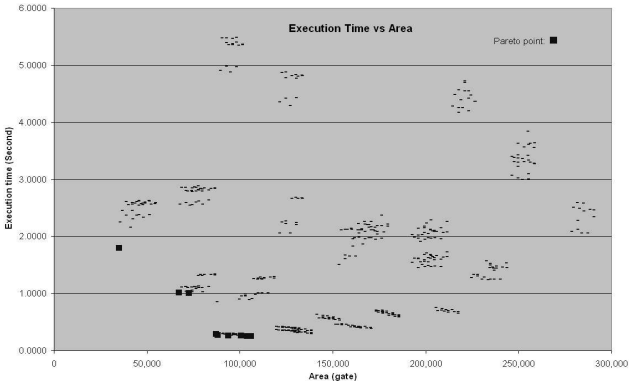

Figure 6. The results and the Pareto points.

the original execution time (configuration 1 in Table 5). The fastest configuration is with Xtensa processor P3 (that is with a floating-point unit and its associated configurable core options). Seven TIE instructions (LDEXP, FREXP, MYMOD3, FREXPLN, MANT, LP24, COMB) are also implemented in the fastest configuration to replace four software functions. There are square root, modular 3, natural logarithm and floating-point division. The time for selection of core options and instructions are in the order of a few hours (about 3-4hrs), while the exhaustive simulation method would take several weeks (about 300 hrs) to complete.

6 Conclusions

This paper describes a methodology to maximize application performance in an ASIP, through the selection of coprocessors / functional units and specific TIE instructions when an area constraint is given. The methodology described uses a combination of simulation and estimation to greedily construct an ASIP.

Our methodology has demonstrated how a speech recognition application can be designed within a reconfigurable processor environment (we used Xtensa). The application's execution time is reduced by 85% when a floating-point coprocessor is selected and seven of our proprietary TIE instructions (LDEXP, FREXP, MYMOD3, FREXPLN, MANT, LP24, COMB) are implemented to replace four software function calls: square root, modular 3, natural logarithm and floating-point division. In addition, our methodology was able to locate all nine Pareto points from the design space of 576 configurations. Finally, the performance estimation for the proposed implementation is on average within 4% of the simulation results.

References

[1] *Tensilica Instruction Extension (TIE) Language Reference Manual (For Xtensa T1040 Processor Cores).* Tensilica, Inc, 2001.

[2] *Tensilica Instruction Extension (TIE) Language User's Guide (For Xtensa T1040 Processor Cores).* Tensilica, Inc, 2001.

[3] ALOMARY, A., NAKATA, T., HONMA, Y., IMAI, M., AND HIKICHI, N. An asip instruction set optimization algorithm with functional module sharing constraint. In *ICCAD* (1993), pp. 526–532.

[4] BINH, N., IMAI, M., AND TAKEUCHI, Y. A performance maximization algorithm to design asips under the constraint of chip area including ram and rom size. In *ASP-DAC* (1998).

[5] CHOI, H., AND PARK, I. Coware pipelining for exploiting intellectual properties and software codes in processor-based design. In *13th Annual IEEE International ASIC/SOC* (2000), pp. 153–157.

[6] FAUTH, A. Beyond tool-specific macine description. *Code Generation for Embedded Processors* (1995), 138–152.

[7] GONZALEZ, R. Xtensa: A configurable and extensible processor. *IEEE Micro* (2000).

[8] GUPTA, T. V. K., KO, R. E., AND BARUA, R. Compiler-directed customization of asip cores. In *10th International Symposium on Hardware/Software Co-Design* (2002).

[9] GUPTA, T. V. K., SHARMA, P., BALAKRISHNAN, M., AND MALIK, S. Processor evaluation in an embedded systems design environment. In *13th International Conf. on VLSI Design* (2000), pp. 98–103.

[10] GYLLENHAAL, J. C., HWU, W. M., AND RAU, B. R. *HMDES Version 2.0 specification.* University of Illinois, 1996.

[11] HADJIYIANNIS, G., RUSSO, P., AND DEVADAS, S. A methodology for accurate performance evaluation in architecture exploration. In *DAC* (1999), ACM Press, pp. 927–932.

[12] HALAMBI, A., GRUN, P., GANESH, V., KAHARE, A., DUTT, N., AND NICOLAU, A. Expression: A language for architecture exploration through compiler/simulator retargetability. In *DATE 99* (1999), pp. 485–490.

[13] HOFFMANN, A., KOGEL, T., NOHL, A., BRAUN, G., SCHLIEBUSCH, O., AND WAHLEN, O. A novel methodology for the design of application-specific instruction-set processors (asips) using a machine description language. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 20*, 11 (2001), 1338–1354.

[14] IMAI, M., BINH, N., AND SHIOMI, A. A new hw/sw partitioning algorithm for synthesizing the highest performance pipelined asips with multiple identical fus. In *EURO-VHDL'96* (1996), pp. 126–131.

[15] JAIN, M. K., WEHMEYER, L., STEINKE, S., MARWEDEL, P., AND BALAKRISHNAN, M. Evaluating register file size in asip design. In *9th international symposium on Hardware/software codesign* (2001), pp. 109–114.

[16] KOBAYASHI, S., MITA, H., TAKEUCHI, Y., AND IMAI, M. Design space exploration for dsp applications using the asip development system peas-iii. In *IEEE International Conf. on Acoustics, Speech, and Signal Processing* (2002), pp. 3168 – 3171.

[17] LEUPERS, R., AND MARWEDEL, P. Retargetable code generation based on structural processor descriptions. In *Design Automation for Embedded Systems* (1998), pp. 75–108.

[18] ONION, F., NICOLAU, A., AND DUTT, N. Incorporating compiler feedback into the design of asips. In *DATE* (95), pp. 508–513.

[19] PAULIN, P. G., LIEM, C., MAY, T. C., AND SUTAWALA, S. Flexware: A flexible firmware development environment for embedded systems. In *Code Generation for Embedded Processors* (1995), pp. 65–84.

[20] YANG, J.-H., KIM, B.-W., ET AL. Metacore: an application specific dsp development system. In *DAC* (1998), pp. 800–803.

[21] ZHAO, Q., MESMAN, B., AND BASTEN, T. Practical instruction set design and compiler retargetability using static resource models. In *DATE* (02), pp. 1021–1026.