INSIDE: INstruction Selection/Identification & Design Exploration for Extensible Processors

Newton Cheung
School of Computer Science & Engineering
University of New South Wales
Australia
ncheung@cse.unsw.edu.au

Sri Parameswaran
School of Computer Science & Engineering
University of New South Wales
Australia
sridevan@cse.unsw.edu.au

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

ABSTRACT

This paper presents the INSIDE system that rapidly searches the design space for extensible processors, given area and performance constraints of an embedded application, while minimizing the design turn-around-time. Our system consists of a) a methodology to determine which code segments are most suited for implementation as a set of extensible instructions, b) a heuristic algorithm to select pre-configured extensible processors as well as extensible instructions (library), and c) an estimation tool which rapidly estimates the performance of an application on a generated extensible processor. By selecting the right combination of a processor core plus extensible instructions, we achieve a performance increase on average of 2.03x (up to 7x) compared to the base processor core at a minimum hardware overhead of 25% on average.

1. INTRODUCTION

Extensible processors are becoming increasingly popular with embedded system designers who use such processors to meet competing design challenges such as high performance, chip area constraints, high flexibility and short design turn-around-time. An extensible processor combines the programmability of a general-purpose processor (GPP) and the customizability of an application specific integrated circuit (ASIC). Extensible processors allow the designer to implement custom-designed specific instructions on top of a base instruction set. In addition, pre-fabricated coprocessors and functional units (such as DSP) can be attached to extensible processors in order to further enhance performance. In brief, extensible processors allow the simultaneous tuning of hardware configurations and enriching of the instruction set architecture.

Tuning the hardware configurations consists of selecting suitable components such as DSP coprocessors, floating-point units, multipliers, the memory/cache architecture, register file size, etc. Enriching the instruction set architecture refers to designing specific customized (extensible) instructions for an application program. After configurable hardware and extensible customized instructions are specified, the extensible processor can be generated.

Although there are a wide range of commercial design suites [1, 2, 4, 5, 6, 7] for extensible processors, a number of problems remain unsolved. These are: i) the identification of a set of suitable extensible instructions given design constraints and an embedded application; ii) the automatic generation of extensible instructions; iii) the automated selection of pre-configured processor cores that can afterwards be extended by extensible instructions; and iv) rapidly (beyond the speed of ISS) estimating the performance of the application on the newly created processor (an instruction set simulator takes a long time). To date, the identification of code segments and the creation of extensible instructions are solely based on the designers experience and are a time consuming process. Similarly, selecting coprocessors and specific instructions and integrating them into an extensible processor are known to be NP-hard problems [12]. In order to further shorten the design turn-around-time of an extensible processors, heuristics are needed.

We present the INSIDE system which is a design system consisting of a methodology for identifying a set of extensible instructions, a heuristic algorithm (this algorithm contains two parts, Part I & II, one to select the processor and the other to select the instructions) for generating extensible processors, and a performance estimator to estimate the application performance on a newly configured extensible processor.

Without limitation of generality, the work presented within this paper used the Xtensa extensible processor from Tensilica [7] as the target platform for development and evaluation of the INSIDE system. The work carried out and the design flow developed could have been conducted with any other extensible processor platform with the same capabilities (configurability and instruction set extension).

The rest of this paper is organized as follows: Section 2 describes motivations behind our work while Section 3 presents the basic idea and contribution. Section 4 describes the related work. Our INSIDE system for generating extensible processor is introduced in Section 5. Experiments and results are discussed in Section 6 while Section 7 concludes the paper.

2. MOTIVATION

The arrival of extensible processors has allowed the embedded system designers to customize their systems to a greater extent. There is sufficient evidence to show that for a given application, it is possible to achieve better performance with a smaller footprint using an extensible processor (as opposed to an off-the-shelf embedded processor) [8, 13, 14, 16, 20]. The design space of an extensible processor may comprise many thousand configurations. Therefore, design turn-around-time of an extensible processor can be in the order of months and even then all possible configurations are often not searched through. Thus, it is important to reduce the design turn-around-time of an extensible processor in an environment where time to market of embedded products is crucial. The motivation of this work is to shorten the design turn-around-time of extensible processors.

To reduce the time design turn-around-time we have identified a number of open problems. The first problem is to

```
1 static int fmult(int an, int srn) {
2      short anmag, anexp, anmant, wanexp, wanmant, retval;
3      anmag = (an > 0) ? an : ((-an) & 0x1FFFF);
4      anexp = quan(anmag, power2, 15) - 6;
5      anmant = (anmag == 0) ? 32 :
                      (anexp >= 0) ? anmag >> anexp :
                      anmag << -anexp;
6      wanexp = anexp + ((srn >> 6) & 0xF) - 13;
7      wanmant = (anmant × (srn & 077) + 0x30) >> 4;
8      retval = (wanexp >= 0) ? ((wanman << wanexp) & 0x7FFF) :
                 (wanmant >> -wanexp);
9      return (((an ∧ srn) < 0) ? -retval : retval);
10 }
```

Figure 1: Motivational example

identify the right code segment to be implemented as an extensible instruction. For example, in Fig. 1, each line within the function (from lines 3-9), can be implemented as one or more instructions. Alternatively, line 3 can be an instruction, line 4 can be a separate instruction and lines 5-9 can be a third instruction. Even for this simple example, there are hundreds of combinations. The second problem is that the creation of the extensible instruction is a time consuming process (in the order of days for a single instruction), and thus created the instruction must be in a reusable form for other applications which might use them later. The third problem is selecting the correct combination of coprocessor and extensible instructions (which has been identified as an NP-hard problem [12]). For example, let us assume that there are five sections of code which can be implemented as instructions. Instruction 1-5, take up 2000, 2500, 3000, 5000, and 10,000 gates respectively. If an area constraint of 8000 were to be imposed (excluding the base processor), then only a subset of the instructions can be chosen. This choice is made such that the performance is maximized. The problem is further complicated with the availability of coprocessors. The final and the fourth problem is that simulating the performance of a configurable extensible processor with an application using Instruction Set Simulator (ISS) to evaluate several configurations is a time consuming process.

Contributions

The contributions of our work include:

1. A fitting function is introduced which identifies sections of code segments (within an application), which are suitable for translation to instructions within the processor.
2. We have implemented a two-stage approach to first select a pre-defined processor, and then to select the right instruction set for this extensible processor. We have furthermore developed cost functions and heuristic algorithms to guide the selection process.
3. We introduce a set of pre-configured processors (from which one is chosen for final implementation) and a pre-designed library of extensible instruction to prune the design space of the extensible processor. The instruction library also allows re-usability of instructions across application domains.
4. A performance estimator has been implemented to estimate an application's performance instead of running each configuration repeatedly through an instruction set simulator and thus minimizing the evaluation time.

This is the first system of its kind that achieves high speedups at reasonable costs without extensive (time consuming) design space explorations.

3. RELATED WORK

Extensible processors have become commercially available within the last 3-5 years. Many shortcomings of the commercial approaches are addressed by researchers in order to enhance the usability and acceptance of these tool suites.

In [9, 20, 24] tool suites have been proposed that take a specification written in an architectural description language and generates retargetable compilers, instruction set simulators (ISS) of the target architecture, and synthesizable HDL models of the target processor. The generated tools allow valid assembly code generation and performance estimation for each specified architecture.

Since designing an application-specific processor for an embedded system this way can still be very time intensive, several methodologies were proposed to shorten the design time by predicting the parameters of the processor close to the specification phase. In 2000, [15] proposed a processor evaluation methodology to quickly estimate the performance changes when architectural modifications are made. Their methodology considers small functions in an application and their estimations are accurate within 30% compared to a whole simulation. Jain [17] proposed a methodology for evaluating the register file sizes in extensible processor designs, thus optimizing the area and energy consumption significantly.

Instruction generation for extensible processors is often referred to as template generation in a compiler environment. A template is an instance of a computational pattern (e.g. $a = b + 10$). In [10], the authors searched for regularity in sequential, parallel and combined sequential/parallel templates in a dataflow graph. New sets of instructions were generated by combining basic instructions. Kastner et al. in [18] searched for an optimal cover of a set of regular templates, and then constructed an optimal set of sequential instructions. Zhao et al. in [25] used static resource models to explore possible new instructions that can be added to the data path to enhance performance.

The overall design flow for extensible processors involves a combination of instruction generation and design space exploration tools. In [19], Kathail et al. proposed a design flow for a VLIW processor, which consists of a selection of non-programmable hardware accelerators (NPA), design space exploration of implementing different combinations of NPAs, and evaluation of the designs. An NPA is a coprocessor for functions expressed as compute-intensive nested loops in C. Similarly, Lee et al. in [22] proposed a design flow with instruction encoding, complex instruction generation, and a heuristic design space exploration approach. Instruction encoding enables trade-offs between the size of opcodes and operand(s). A complex instruction is an instruction with more than one basic, frequently occurring, operation. Their design flow first takes an application program and basic Instruction-Set (IS) as inputs. Then they convert the code into assembly code. Based on the resource and the timing information of the basic IS and application program, complex instruction patterns are generated and are placed in a library. Then they select a set of complete instructions for an extensible processor. Finally, in [23] the design flow consists of generating instructions automatically, inserting instructions, and performing a heuristic design space exploration. Automatic instruction generation locates the regular templates derived from program dependence graphs, and implements the most suitable ones as extensible instructions, enhancing performance of the application program. Their design flow takes an application, which is profiled, and from the output a program dependence graph is created. Blocks within the graph are ranked, and the highest ranking blocks are implemented as instructions.

4. THE INSIDE SYSTEM

The INSIDE design flow for generating extensible processors involves an all over of 11 individual steps (see Fig. 2). The input to the INSIDE system consists of: an application written in C/C++, a set of pre-configured processors, a library of extensible instructions, and area/performance constraints. First, the application is compiled and is profiled on each of the pre-configured processors[1]. During the profiling phase, we obtain for each of the pre-configured processors, characteristics and information for all functions/subroutines[2] within the application. The design flow consists of four major phases (our contributions are highlighted in grey): I) a heuristic algorithm (Part I) for selecting a pre-configured processor from the set of pre-configured processors (steps 2-3 in Fig. 2), which is described in section 4.1; II) a methodology for identifying code segments which are implementable as instructions, and designing them as extensible instructions (steps 4-8), which is described in section 4.2; III) a heuristic algorithm (Part II) for selecting a set of extensible instructions to be implemented on the selected pre-configured processor (steps 9-10), which is described in section 4.3; and IV) performance estimation of an application on the designed extensible processor (step 11), which is described in section 4.4. This section describes these four phases of the INSIDE system in detail and concludes with a summary of the overall design flow.

One of the main ideas behind INSIDE is the hierarchical approach of designing an extensible processor by a) first limiting the design space through selection of a pre-configured processor core and b) by selecting an appropriate set of extensible instructions on that specific processor. This approach allows to efficiently search the design space while allowing for designer's input. The pseudo code is shown in Fig. 3. Details will be explained in the following sections.

4.1 Selecting a pre-configured processor

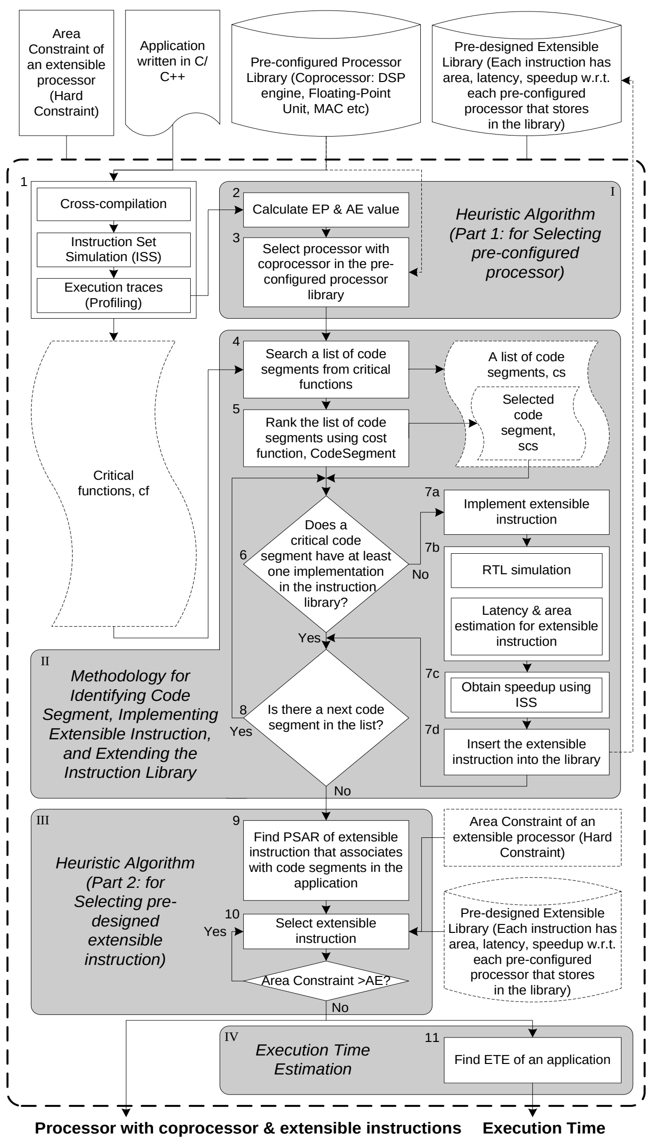
Our library currently consists of three *P1, P2, and P3* pre-configured processors that are proven to cover a wide range of application characteristics. The characteristics are shown in Table 1. *P1* is the base processor with a DSP engine. *P2* has a floating point unit and the base processor, and *P3* is the base processor alone. (more processors might be pre-configured, but these three have been chosen to exemplify our methodology). Through simulation and profile information (step 1 of Fig. 2), we obtain the total cycle-count as well as information of critical functions within the application. Then, the "effectiveness" EP_i of the processor i for a certain application is reviewed by considering the total cycle-count (CC), clock period (Clk_PD), and the area of each extensible processor as follows:

$$EP_i = \frac{1}{CC_i \times Clk\_PD_i \times Area\_Proc_i} \qquad (1)$$

This function is inversely proportional to the area delay product and we rank the processor with the highest performance-per-area ratio in order to apply the heuristic algorithm (Part I).

[1]Note, that this step implies that the set of pre-configured processors will have at least one that addresses the constraints/characteristics of a specific application reasonably well. This first step is a major designer's input that allows the designer to provide the system with domain-specific architectural features without fixing the processor core.

[2]Granularity may vary.



Area Constraint of an extensible processor (Hard Constraint)
Application written in C/ C++
Pre-configured Processor Library (Coprocessor: DSP engine, Floating-Point Unit, MAC etc)
Pre-designed Extensible Library (Each instruction has area, latency, speedup w.r.t. each pre-configured processor that stores in the library)
1
Cross-compilation
Instruction Set Simulation (ISS)
Execution traces (Profiling)
2
Calculate EP & AE value
3
Select processor with coprocessor in the pre-configured processor library
I
Heuristic Algorithm (Part 1: for Selecting pre-configured processor)
4
Search a list of code segments from critical functions
A list of code segments, cs
Selected code segment, scs
5
Rank the list of code segments using cost function, CodeSegment
Critical functions, cf
7a
Implement extensible instruction
6
Does a critical code segment have at least one implementation in the instruction library?
No
7b
RTL simulation
Latency & area estimation for extensible instruction
Yes
7c
Obtain speedup using ISS
II
Methodology for Identifying Code Segment, Implementing Extensible Instruction, and Extending the Instruction Library
8
Is there a next code segment in the list?
Yes
7d
Insert the extensible instruction into the library
No
III
Heuristic Algorithm (Part 2: for Selecting pre-designed extensible instruction)
9
Find PSAR of extensible instruction that associates with code segments in the application
Area Constraint of an extensible processor (Hard Constraint)
10
Yes
Select extensible instruction
Pre-designed Extensible Library (Each instruction has area, latency, speedup w.r.t. each pre-configured processor that stores in the library)
Area Constraint >AE?
No
IV
Execution Time Estimation
11
Find ETE of an application
Processor with coprocessor & extensible instructions
Execution Time

Figure 2: INSIDE's design flow for automatically generating extensible processor (Double square box: commercial tools; Gray box: our contributions)

```
Heuristic Algorithm {
    For (1 to n extensible processor) { (Part I)
        $EP_i = \frac{1}{CC_i \times Clk\_PD_i \times Area\_Proc_i}$;
    }
    Select the processor with the highest value of $EP$;
    For all extensible instructions in the instruction library { (Part II)
        $PSAR_{jk} = \frac{\%_{jk} \times Sp\_Inst_{jk}}{Area\_Inst_j \times Max(Latency_j, Clk\_PD_k)}$;
    }
    For (the highest PSAR to the lowest PSAR) {
        If ($Area\_Remain > Area\_Inst_j$) {
            Select $Inst_j$;
            $Area\_Remain = Area\_Constraint - Area\_Inst_j$;
        }
    }
}
```

Figure 3: Pseudo code of the heuristic algorithm (Parts I & II) for selecting pre-configured processor and extensible instructions

4.2 Identifying SW code segments to convert to instructions

Identifying code segments and implementing extensible instructions from an application program are traditionally based on designers' experience. The purpose of our methodology is to identify code segments within a program function that are amenable to implement into an extensible instruction. The input of our methodology is a list of the critical functions that are obtained from profiling . A critical function is a function that contributes more than $\theta\%$ of the total execution time (in our case, θ is 5). The identification consists of five points:
i) Exhaustively searching a list of critical code segments as part of critical functions (step 4 in Fig. 2);
ii) Identifying and ranking the list of critical code segments using a fitting function (step 5);
iii) Checking whether an equivalent implementation is part of our extensible instruction library (step 6);
iv) If there is not an equivalent instruction to the code segment in the library, then implement the code segment as an extensible instruction and characterize the instruction using the Xtensa development tools from Tensilica Inc. [7] and Design Compiler from Synopsys Inc. [3] with associated scripts (steps 7a - 7d);
v) If there is an equivalent instruction that matches the code segment, then move down to the next item in the list of code segments (step 8). Note that the code segment matching against the instructions in the library is presently performed manually. The methodology outputs a set of extensible instruction with its area, latency and speedup. The created extensible instructions are added to the extensible instruction library for reuse and can be selected in the later phase of the design flow.

Fitting Function

The fitting function evaluates suitable code segments that can be implemented as extensible instructions. The fitting function broadly indicates speedup / area ratio of an instruction. The input is a list of critical functions (by profiling). Within the critical functions, all possible code segments (only consecutive statements are allowed) are exhaustively searched and ranked using this fitting function. This fitting function could lead to more than one suitable extensible instruction for a single code segment. Multiple instructions for the same code segment gives additional freedom for selection in latter stage of the design flow.

This fitting function is derived from studying manually performed extensible processor designs. The fitting function of $CodeSegment_x$ is defined as:

$$CodeSegment_x = FU_x \times \frac{1}{\lceil \frac{NO_x}{\alpha} \rceil} \times TO_x \times BO_x$$

where
FU_x = the frequency of use of a code segment x;
NO_x = the number of operands in a code segment x;
TO_x = the percentage of integer (short) type operands in all the operands (char would be considered an integer);
BO_x = the percentage of bit operations in all the operations;
α is the ideal number of operands in the code segment (in our case, the number of operands is less than or equal to 2 inputs and 1 output).

The frequency of use, FU_x, indicates how often a code segment is executed in the application. FU_x is obtained from the execution traces of the application program. Moving these segments to extensible instructions are likely to have great impact upon the speed of the program.

```
static int fmult(int an, int srn) {
    short anmag, anexp, anmant, wanexp, wanmant, retval;
    anmag = (an > 0) ? an : ((-an) & 0x1FFFF);
    anexp = quan(anmag, power2, 15) - 6;
    anmant = (anmag == 0) ? 32 :
                (anexp >= 0) ? anmag >> anexp :
                anmag << -anexp;
    wanexp = anexp + ((srn >> 6) & 0xF) - 13;
    wanmant = (anmant × (srn & 077) + 0x30) >> 4;
    retval = (wanexp >= 0) ? ((wanman << wanexp) & 0x7FFF) :
            (wanmant >> -wanexp);
    return (((an ∧ srn) < 0) ? -retval : retval);
}

static int quan(int val) {
    static short table[15] = {1, 2, 4, 8, 0x10, 0x20, 0x40, 0x80,
            0x100, 0x200, 0x400, 0x800, 0x1000, 0x2000, 0x4000}
    for (int i = 0; i < 15; i++)
        if (val < table[i])
            break;
    return (i);
}
```

Figure 4: Example Code for Fitting Function

If the number of operands to be implemented as an instruction has just three or less operands (less than or equal to two source and one destination operand), then it is an instruction which can be implemented fairly easily, as processors typically have 2 source busses and one destination bus going to the ALU. When the number of operands are more than the above, then multiple cycles are needed to ferry the operands to the newly created functional unit increasing the latency of the operation. This is reflected as NO_x in the cost function.

The amount of bit operations in a segment, favors its implementation in hardware, since such an instruction requires a small cycle count and has high performance gain on the application program. The amount of bit operations in a segment is reflected in the cost function as BO_x.

Finally, the type of operands, TO_x, in an instruction relates to the type of register file and the manipulation of the operands. If the type of operands is different, then the processor needs extra registers or even custom designed registers. These registers increase the area of an extensible instruction. If the manipulation of the operands is needed, then the latency of the instruction increases as well. The increase in latency and area is reflected as TO_x in the cost function. By including these four factors (FU_x, NO_x, BO_x and TO_x) in the fitting function, the fitting function relates to the speedup / area ratio of an extensible instruction for a code segment. In the result section, Fig. 7 shows how well the fitting function indicates the speedup/area ratio of the instructions, for a variety of code segments.

Fig. 4 gives an example of how the fitting function is used. The example consists of two segments: *fmult* and *quan* where *fmult* is use up 22% of the execution time. The number of operands of this function is 3 (namely *an, srn.* The operation types in the function are mostly bit operations (i.e. and, left shift, right shift etc.), so $BO_{fmult} = 0.8$. The types of operands are integers and therefore $TO_{fmult} = 1$. Thus the value of the fitting function 0.176 whereas *quan* will yield 0.28 indicating that there is a higher benefit in *quan* rather than *fmult.* As shown later in Table 2, the extensible instruction, QUAN, has better area, speedup, and latency than the extensible instruction, FMULT.

4.3 Selecting a set of extensible instructions

The selection of an extensible instruction is based on the area ($Area\_Inst$) to speed-up ratio which is clock-cycle based (Sp_Inst), latency (maximum clock period of all extensible instructions and the pre-configured coprocessor), and percentage of total cycle-count ($\%_{jk}$) for the function in the application under consideration for the pre-configured processor k. We define the potential speedup/area ratio, $PSAR_{jk}$, of extensible instruction j in extensible processor k as:

$$PSAR_{jk} = \frac{\%_{jk} \times Sp\_Inst_{jk}}{Area\_Inst_j \times Max(Latency_j, Clk\_Pd_k)} \quad (2)$$

$PSAR_{jk}$ indicates the performance gain per area of an application when the extensible instruction is implemented. The ratio is used to rank the instructions in order from the most performance gain per area to the least. In addition, our equation takes into account the latency of the instruction which is critical to the real performance of the application.

4.4 Performance and Area Estimation

After a pre-configured processor with coprocessor and a set of extensible instructions are selected, we perform an estimation of *execution time* (step 11 of Fig. 2). The first part of the equation 3 calculates the cycle-count of an application program running on the proposed extensible processor, then the cycle count is multiplied by the maximum latency. Maximum latency is the maximum clock period value of all the extensible instructions and the pre-configured processor that have been selected. The execution time estimation, ETE_{jk}, for an extensible processor k with a set of selected extensible instruction (from 1 to j) is defined as:

$$ETE_{jk} = \{CC_k \times (1 - \sum_j \%_{jk}) + \sum_j \frac{CC_k \times \%_{jk}}{Sp\_Inst_{jk}}\} \times Latency_{max} \quad (3)$$

where CC_k is the original total cycle-count of an application running on an extensible processor k.

The *area* estimation used in the selection of pre-configured processor and extensible instructions, is defined as:

$$AE_{jk} = Area_{base} + \sum Area_{copr} + \sum Area_{inst} \quad (4)$$

where, $Area_{base}$ is the number of gates used by the base processor, $Area_{copr}$ is the additional gates incurred by the selected coprocessor and $Area_{inst}$ is the number of gates that the selected instruction occupy. The first two terms of the equation 4 are estimated using the Xtensa generator [7] and the final term is estimated using the Design Compiler from Synopsys, Inc. [3]. Since the busses are not changed significantly by the addition of coprocessors, the gate count still gives a good indication of the area. The custom register files and any extra tristates inserted for increased bus lengths are reflected by the last term.

4.5 Overall Design Flow Algorithm

The overall algorithm brings together the various steps in the INSIDE system. We first compile, simulate and profile the application program to obtain a list of critical functions, cf. Then we calculate the cost functions for area and speed area ratio, AE and EP, of each pre-configured processor. We then select a processor with the highest value of EP, with AE being less than the area constraint. From the list of critical functions, we exhaustively search all possible combinations of code segments which are consecutive lines of code. We rank the "code segments" according to our fitting function, $CodeSegment$. We select code segments with a $CodeSegment$ value greater than 0.001. Then, if any of the "selected code segment", scs, do not exist in the instruction library, we create the instruction manually and then add it to the extensible instruction library. We continue to implement instructions until all the selected code segments are available in the library. After this we run the Instruction selection algorithm to select a set of extensible instructions using cost functions, $PSAR$ (potential speed area ratio) and AE. Finally we perform an estimation, ETE, to check on the performance of the created extensible processor. Fig. 5 gives the overall design flow algorithm of the INSIDE system.

```
Overall INSIDE Algorithm() {
    \* Compile, simulate and profile the application *\
    Compile the application program;
    Simulate the application program using ISS;
    Profile the application program (a list of critical functions, $cf_i$);

    \* Selecting a pre-configured processor *\
    For (i = 1 to w pre-configured processor) { (Step 2)
        $EP_i = \frac{1}{CC_i \times Clk\_PD_i \times Area\_Proc_i}$;
        $AE_i = Area_{base_i} + \sum Area_{copr_i}$;
    }
    For (i = 1 to w pre-configured processor) { (Step 3)
        If ($AE_i < Area\_Constaint$)
            Select processor with the highest value of $EP$;
    }

    \* Identifying code segment *\
    For (i = 1 to x critical functions, $cf_i$) {
        Search exhaustively for all code segments, $cs_{ij}$; (step 4)
        For (j = 1 to y code segment, $cs_{ij}$, in function i) { (step 5)
            $CodeSegment_j = FU_j \times \frac{1}{\lceil \frac{NO_j}{\alpha} \rceil} \times TO_j \times BO_j$;
            If ($CodeSegment_j$ of $cs_{ij} > 0.001$) {
                Insert into a list of selected code segment, $scs_j$;
            }
        }
    }
    \* Manually checking whether a selected
    code segment matches an instruction *\
    For (j = 1 to z selected code segment, $scs_j$) { (Step 6)
        If (selected code segment, $scs_j$, is not in library) {
            Manually create code segment into instruction; (step 7a)
            Characterize the instruction; (steps 7b-7c)
            Insert the extensible into the library; (step 7d)
        } Else {
            Continue search; (step 8)
        }
    }

    \* Selecting a set of extensible instruction *\
    For all extensible instructions in instruction library { (step 9)
        $PSAR_{jk} = \frac{\%_{jk} \times Sp\_Inst_{jk}}{Area\_Inst_j \times Max(Latency_j, Clk\_PD_k)}$;
    }
    For (j = the highest PSAR to the lowest PSAR) { (step 10)
        If ($Area\_Remain > Area\_Inst_j$) {
            Select $Inst_j$;
            $AE_{jk} = Area_{base_k} + \sum Area_{copr_k} + \sum Area_{inst_j}$;
            $Area\_Remain = Area\_Constraint - AE_{jk}$;
        }
    }

    \* Execution time estimation *\ (step 11)
    $ETE_{jk} = \{CC_k \times (1 - \sum \%_{jk}) + \sum \frac{CC_k \times \%_{jk}}{Sp\_Inst_{jk}}\} \times Latency_{max}$;
}
```

Figure 5: Overall algorithm of the INSIDE system

5. EXPERIMENTS & RESULTS

We have set up our design flow described in the previous section using tools and scripts to design extensible processors for a set of applications. The target extensible processor used in our experiments is the Xtensa processor from Tensilica, Inc. Two libraries have also been created: a pre-configured processor library and a library of pre-designed extensible instructions, which stores a set of pre-configured processors and all the extensible instructions generated through our methodology respectively. These experiments were conducted on a Sun UltraSPARC10 running at 440MHz with 1Gb of RAM.

We have pre-configured three extensible processors in the first library, namely P1, P2, and P3. These processors are configured from the T1040.0 version of the Xtensa processor in 0.18μ technology. Pre-configured processor P1 has a Vectra

Pre-configured Processor	P1	P2	P3
Area [mm2]	4.23	2.28	1.7
Area [gates]	159800	86720	61620
Power [mW]	161	108	96
Clock Rate [MHz]	158	155	183

Table 1: Hardware Cost for Each Processor

Extensible Inst	Application Used	Area [gates]	Speedup under P1	P2	P3	Latency [ns]	Cost function
GSMS	$\mathbf{gsm_{dec}}$, gsm_{enc}	2740	3.50×	1.12×	1.20×	6.00	0.03
CAL_1	$\mathbf{gsm_{dec}}$, gsm_{enc}	16000	4.50×	4.50×	3.50×	6.00	0.009
GSMMR	$\mathbf{gsm_{dec}}$, gsm_{enc}	23400	3.45×	N.A.	N.A.	7.46	0.004
GSMLM	$\mathbf{gsm_{dec}}$, gsm_{enc}	13200	N.A.	1.25×	1.30×	6.25	0.003
DC3	$\mathbf{adpcm_{enc}}$	2630	N.A.	1.20×	1.30×	6.00	0.02
DC4	$\mathbf{adpcm_{enc}}$	5810	3.65×	3.20×	3.00×	6.00	0.03
DC1,DC2	$\mathbf{adpcm_{enc}}$	10154	1.30×	1.20×	1.50×	6.00	0.004
MOD3	**voice**	5500	N.A.	17.0×	10.9×	6.40	0.11
LDE	**voice**, $mpeg2_{dec}$	1100	N.A.	2.50×	N.A.	6.50	0.06
MN,LP,CB	**voice**, $mpeg2_{dec}$	6800	3.52×	5.28×	6.48×	6.80	0.03
FM32	**voice**, $mpeg2_{dec}$	32000	8.98×	N.A.	11.6×	7.10	0.02
FREXP	**voice**, $mpeg2_{dec}$	3200	N.A.	1.90×	N.A.	6.90	0.02
LDE,FRE	**voice**, $mpeg2_{dec}$	3300	N.A.	3.30×	N.A.	7.00	0.02
FREXPLN	**voice**, $mpeg2_{dec}$	3300	N.A.	1.10×	N.A.	6.90	0.01
FD32	**voice**, $mpeg2_{dec}$	53800	N.A.	15.9×	17.4×	14.6	0.01
FA32	**voice**, $mpeg2_{dec}$	32000	8.31×	N.A.	8.14×	8.50	0.009
CC	$\mathbf{mpeg2_{dec}}$	50	3.30×	4.12×	4.20×	4.33	0.34
MYSAT	$\mathbf{mpeg2_{dec}}$	180	4.51×	5.55×	5.55×	5.57	0.12
ADD14	$\mathbf{mpeg2_{dec}}$	1065	3.30×	N.A.	N.A.	6.00	0.11
ADD8	$\mathbf{mpeg2_{dec}}$	950	1.20×	1.30×	1.30×	5.96	0.05
QUAN	$\mathbf{g721_{dec}}$, $g721_{enc}$	1200	8.30×	10.0×	10.0×	6.00	0.28
FMULT	$\mathbf{g721_{dec}}$, $g721_{enc}$	10000	10.0×	24.0×	24.0×	8.15	0.22
RECONS.	$\mathbf{g721_{dec}}$, $g721_{enc}$	7000	2.60×	2.50×	2.50×	7.20	0.007
SSIZE	$\mathbf{g721_{dec}}$, $g721_{enc}$	21000	3.00×	3.00×	3.00×	7.50	0.005

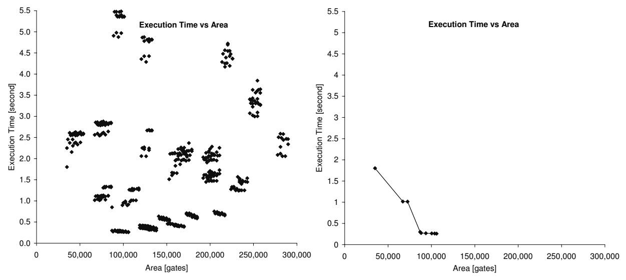
Table 2: Extensible Instructions Library

DSP engine, P2 has a floating point unit, and P3 is the base processor with no additional coprocessor. All processors are set up with direct mapped 1KB instruction and data caches, 128-bit wide system bus and a generic register file with 64 32-bit registers. Table 1 shows the parameters of these pre-configured processors. The instruction library contains 26 specific instructions in total. Table 2 shows the information available to the designer from the extensible instruction library. The first column is the extensible instruction name. The second column lists the application name that uses the extensible instruction. The applications in bold shows from which of the applications the instruction was derived. The next five columns indicates the area, the speedup of the specified instruction when associated with processor *P1*, *P2*, *P3*, and latency of the instruction respectively. The last column is the fitting function's value of the corresponding code segment. Please note that the fitting function is only comparable within the application from which the instruction was derived. While the cost functions of GSMS and CAL_1 are directly comparable, GSMS and DC3 are not.

We performed experiments on seven multimedia benchmarks that were obtained from [11, 21]. These are adpcm encoder, g721 encoder, g721 decoder, gsm encoder, gsm decoder, mpeg2 decoder, voice recognition encoder. For verification purposes, we simulated all the possible combinations of extensible processors with configurable coprocessors and instructions on each benchmark, so that the entire design space (including the Pareto points) of each benchmark is obtained. In addition, we started with a tight area constraint and relaxed the area constraint during our experiments in order to obtain all the possible Pareto points in the design space.

5.1 Results

Our experiments evaluate the design flow of the INSIDE system. We looked at: i) the efficacy of the methodology for identifying code segments for implementation as instructions; ii) the efficiency of the heuristic algorithm (Part I) for selecting a pre-configured processor with coprocessor; iii) the efficiency of the heuristic algorithm for selecting a set of extensible instructions (Part II); and iv) the accuracy of the execution time estimation. Table 3 summarizes these results. The first column of Table 3 indicates the application name, the second column shows the number of Pareto points obtained using the INSIDE system. The total number of Pareto points in the design space is shown in brackets. Column three shows the error between the value obtained by the performance estimator in equation 3 and the performance value obtained using ISS. The next four columns are the optimal solution and the original solution of an application with respect to area and execution time. Finally, the last two columns are the exploration time of the full design space using simulation and our INSIDE system respectively.



(a) Full design space (b) Pareto points

Figure 6: A benchmark's design space.

The design space exploration time for our design flow is on average 2% (the comparison between the last two columns in Table 3) of the design space exploration time using full simulation methodology. Fig. 6 shows the design space of the voice recognition benchmark with 576 configurations and the Pareto points walk through the design space using the INSIDE design flow. For this benchmark, the design process takes approximately 24000 minutes (400 hours) to explore the entire design space, whereas our design flow took only 150 minutes (2.5 hours) to obtain these Pareto points in the design space. Column two of Table 2 shows that an extensible instruction can be reused in more than one application within the same domain. This methodology eliminates repeated effort in the creation of extensible instruction in our INSIDE system. The result clearly indicates that the INSIDE system shortens the design turn-around-time of extensible processor.

To evaluate the effectiveness of the fitting function of the code segment, we included the cost function value of the code segment in the last column of Table 2. Fig. 7 shows a relationship between the speed/area ratio of the instruction and the fitting function in four different benchmarks. Note that the cost function's value for different code segments should only be compared within a benchmark. The benchmarks under consideration improved in performance by up to 7x (on average 2.03x) (the comparison of execution time between our best solution (column 5 in Table 3) and the original solution (column 7) due to the inclusion of coprocessor and extensible instructions. The hardware overhead was 25% on average(compare the area from column 4 and column 6 respectively). The result shows that the methodology for identifying instructions indicates useful instructions to extract from an application program.

The second column from Table 3, shows that the number of Pareto points obtained using the INSIDE system. The total number of Pareto points in the design space is shown in brackets. The heuristic algorithm (Part I & II - for selecting

Application.	Pareto Points obtained (Total Pareto Points)	Error rate of perf. estimation on Pareto Points [%]	Our Best Solution wrt time		Original Solution		Exploration Time	
			Area [gates]	Execution Time [sec.]	Area [gates]	Execution Time [sec.]	Full [minutes]	INSIDE [minutes]
adpcm encoder	3(6)	3%	77,964	1.77	61,620	2.06	3100	100
gsm encoder	4(4)	7%	79,540	13.36	61,620	13.68	9600	160
gsm decoder	5(5)	7%	78,093	6.58	61,620	7.21	3840	110
g721 encoder	4(6)	4%	73,200	1.96	61,620	2.69	1800	90
g721 decoder	2(3)	5%	63,200	2.06	61,620	2.81	1800	90
mpeg2 decoder	5(5)	7%	63,255	0.6321	61,620	0.8021	3600	45
voice recognition	9(9)	4%	105,900	0.2638	61,620	1.8018	24000	150

Table 3: Experimental results

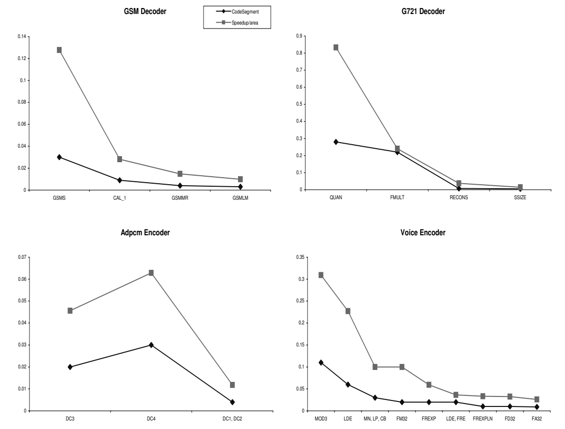


Figure 7: The relationship between the fitting function and speedup/area ratio of the instruction.

pre-configured processor, and extensible instructions when a tight area constraint is given and then progressively relaxed) is able to obtain on average 83% of the Pareto points for all the benchmarks. Although our algorithm does not obtain all the Pareto points, application performance is on average within 3% of the Pareto points'(which were not obtained) application performance. The reason that our heuristic algorithm fails to deliver all Pareto points, is due to the heuristic which searches for ratios rather than absolute values.

Finally, in order to show the efficiency and the accuracy of the execution time estimation of the system, we estimated the execution time for all the obtained Pareto points for extensible processors on each benchmark and compared with the execution time obtained using ISS. The estimation of the execution time is on average within 5% (column 5 in Table 3) of the real execution time of an application program.

6. CONCLUSIONS

This paper described the INSIDE system, for reducing the design turn-around-time of an extensible processor. This reduction is achieved through the identification of efficient code segments to implement as extensible instructions, and a two-level hierarchical selection approach: first, the design space is limited through selection of a pre-configured processor (including a coprocessor), then a set of pre-designed (from a library) extensible instructions is selected for that extensible processor. In addition, execution time estimation for an application program running on an readily configured extensible processor is performed. By use of the INSIDE system it has been demonstrated how seven different, real life, multimedia benchmarks can be designed within an extensible processor environment. The design space exploration time of the INSIDE system is on average 2% of the design space exploration time using full simulation for a given set of benchmarks. The fitting function for identifying the right code segment relates to the speed / area ratio of the instruction. In addition, our heuristic algorithm was able to locate on average 83% of all Pareto points from the entire design space in all benchmarks. The execution time estimation for the proposed extensible processor is on average within 5% of results obtained with an ISS, and is generated in typically less than a second. Finally, the application program execution time is reduced by up to 7x (on average 2.03x), with an average area overhead of 25% on the benchmarks.

7. REFERENCES

[1] Arctangent processor. ARC International. (http://www.arc.com).
[2] Asip-meister. (http://www.eda-meister.org/asip-meister/).
[3] Design compiler. Synopsys Inc. (http://www.synopsys.com).
[4] Jazz dsp. Improv Systems Inc. (http://www.improvsys.com).
[5] Lisatek. CoWare Inc. (http://www.coware.com).
[6] Sp-5flex dsp core. 3DSP Corp. (http://www.3dsp.com).
[7] Xtensa processor. Tensilica Inc. (http://www.tensilica.com).
[8] A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi. An asip instruction set optimization algorithm with functional module sharing constraint. In *DAC*, pages 526–532. IEEE Computer Society Press, 1993.
[9] N. Binh, M. Imai, and Y. Takeuchi. A performance maximization algorithm to design asips under the constraint of chip area including ram and rom size. In *ASP-DAC*, 1998.
[10] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *CASES*, 2002.
[11] N. Cheung, J. Henkel, and S. Parameswaran. Rapid configuration & instruction selection for an asip: A case study. In *DATE*, 2003.
[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co, 1979.
[13] R. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 2000.
[14] M. Gschwind. Instruction set selection for asip design. In *7th international workshop on Hardware/software codesign*, pages 7–11. ACM Press, 1999.
[15] T. V. K. Gupta, P. Sharma, M. Balakrishnan, and S. Malik. Processor evaluation in an embedded systems design environment. In *VLSI Design*, pages 98–103, 2000.
[16] M. Imai, N. Binh, and A. Shiomi. A new hw/sw partitioning algorithm for synthesizing the highest performance pipelined asips with multiple identical fus. In *EURO-VHDL*, pages 126–131. IEEE Computer Society Press, 1996.
[17] M. K. Jain, L. Wehmeyer, S. Steinke, P. Marwedel, and M. Balakrishnan. Evaluating register file size in asip design. In *CODES*, pages 109–114, 2001.
[18] R. Kastner, S. Ogrenci-Memik, E. Bozorgzadeh, and M. Sarrafzadeh. Instruction generation for hybrid reconfigurable systems. In *ICCAD*, 2001.
[19] V. Kathail, shail Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman. Pico: Automatically designing custom computers. *Computer*, 35(9):39–47, Sept 2002.
[20] S. Kobayashi, H. Mita, Y. Takeuchi, and M. Imai. Design space exploration for dsp applications using the asip development system peas-iii. In *ASSP*, pages 3168 – 3171, 2002.
[21] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.
[22] J. Lee, K. Choi, and N. Dutt. Efficient instruction encoding for automatic instruction set desifn of configurable asips. In *ICCAD*, 2002.
[23] F. Sun, S. Ravi, A. Raghunathan, and N. Jha. Synthesis of custom processors based on extensible platforms. In *ICCAD*, 2002.
[24] J.-H. Yang, B.-W. Kim, et al. Metacore: an application specific dsp development system. In *DAC*, pages 800–803, 1998.
[25] Q. Zhao, B. Mesman, and T. Basten. Practical instruction set design and compiler retargetability using static resource models. In *DATE*, pages 1021–1026, 2002.