MINCE: Matching INstructions using Combinational Equivalence for Extensible Processor

Newton Cheung†, Sri Parameswaran†, Jörg Henkel‡, Jeremy Chan†

†School of Computer Science & Engineering, University of New South Wales, Australia
‡NEC Laboratories America, 4 Independence Way, Princeton, NJ 08540, USA
ncheung@cse.unsw.edu.au, sridevan@cse.unsw.edu.au, henkel@nec-labs.com, jeremyc@cse.unsw.edu.au

Abstract

Designing custom-extensible instructions for Extensible Processors[1] is a computationally complex task because of the large design space. The task of automatically matching candidate instructions in an application (e.g. written in a high-level language) to a pre-designed library of extensible instructions is especially challenging. Previous approaches have focused on identifying extensible instructions (e.g. through profiling), synthesizing extensible instructions, estimating expected performance gains etc. In this paper we introduce our approach of automatically matching extensible instructions as this key step is missing in automating the entire design flow of an ASIP with extensible instruction capabilities. Since matching using simulation is practically infeasible (simulation time), and traditional pattern matching approaches would not yield reliable results (ambiguity related to a functionally equivalent code that can be represented in many different ways), we adopt combinational equivalence checking. Our MINCE tool as part of our ASIP design flow consists of a translator, a filtering algorithm and a combinational equivalence checking tool. We report matching times of extensible instructions that are 7.3x faster on average (using Mediabench applications) compared to the best known approaches to the problem (partial simulations). In all our experiments MINCE matched correctly and the outcome of the matching step yielded an average speedup of the application of 2.47x. As a summary, our work represents a key step towards automating the whole design flow of an ASIP with extensible instruction capabilities.

1 Introduction

Application Specific Instruction-set Processors (ASIPs) are designed for specific applications or application domains in embedded systems. ASIPs typically consist of a configurable base processor core and a base instruction set plus the capability of extending this instruction set through new extensible instructions that further enable to address more specifically performance and power constraints. Using commercial and research ASIP platforms [1, 2, 3, 4, 5, 6], it has been shown that performance and power benefits can be orders of magnitudes more efficient compared to general purpose processors when deployed in the same embedded systems [8, 16, 17]. Within the commercial platforms the steps of identifying, matching, synthesizing and estimating (performance/power) of extensible instructions are mostly supported by tools that come with the ASIP tool suite. However, the process of efficiently exploring the design space is up to the designer. Therefore recent research (see related work) has focused on automating this pro-

cess. One of the most challenging and so far unsolved steps in this process is that of automatically *matching* extensible instructions: given a library of pre-designed candidates for extensible instructions that may or may not be included (depending on the application and its constraints) within the final design of the ASIP. The goal of *matching* is to automatically *match* instructions in the library with code segments of the application in order to automatically judge whether a specific code segment (software) of the application might be replaced by an extensible instruction or not. This is a complex task.

The traditional approach to instruction matching consists of instruction simulation [28], and data control graph matching techniques [14, 18, 21, 27]. In the simulation approach, a code segment and the equivalent hand-designed instruction are simulated with the same set of input vectors, while comparing output vectors. The drawback of this approach is the necessity to simulate a complete set of data vectors in order to ensure that the extensible instruction and the software code segment are functionally equivalent. This makes the process not only time computation intensive but also potentially error prone unless a 100% data set coverage is guaranteed.

Another technique, data control graph matching, enables the matching of extensible instructions with a structurally equivalent representation of the according code segment. Since the same segment can be represented graphically in many different ways, such a method will often result in a false negative. The differences in the graphical representation can arise from the level of granularity and the method of decomposition in a function.

To overcome the shortcomings of the simulation and the pattern matching techniques, we propose the MINCE tool. MINCE consists of a translator, a filtering algorithm and a combinational model equivalence checking tool. The translator converts a code segment described in a high-level language (typically C/C++) to a combinational Verilog representation. The filtering algorithm rapidly prunes candidate instructions that cannot match any pre-designed extensible instructions. Finally, the combinational model equivalence checking tool is used to ensure that the functionality of the code segment and the extensible instruction are equivalent. The advantages of the MINCE tool are:

- it automates the step of instruction matching and is superior to computation-intensive and error-prone simulation approaches.
- the usage of functional equivalence checking ensures that the results (i.e. found candidates for extensible instructions) are largely independent of the programming style of the application that is to be accelerated.

MINCE is the automated tool for matching extensible instruction to the functional equivalence of code segments in an ASIP environment.
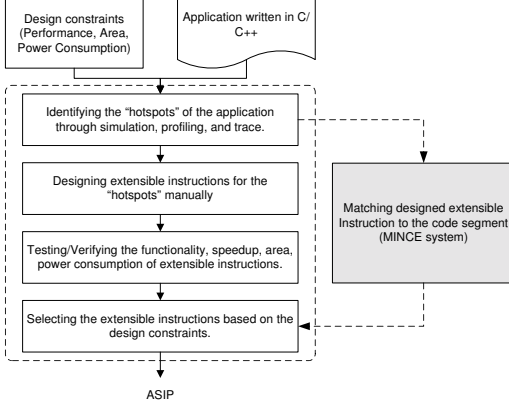
Figure 1. A generic design flow for designing an ASIP and how MINCE system fits in the design flow

(a) Code Segment & Extensible Instruction

(b) BDD representations

Figure 2. Code Segment & Extensible Instruction and the BDD representations

The rest of the paper is organized as follows: Section 2 introduces the background as well as the goal of this work whereas Section 3 describes related work. Then Section 4 presents the steps and flow of our MINCE tool. Experimental setups are described and results are presented for diverse real-world applications (Mediabench) in Section 5. Finally, a conclusion is given in Section 6.

2   Background

A generic design flow (see Fig. 1) for designing an ASIP (given an application written in C/C++ and design constraints such as performance, area etc) typically involves the following four major steps:

1. identifying the "hot spots" (frequently executed code segments) of the application through simulations, profiling and traces;

2. analyzing/designing functional equivalent extensible instructions manually for the identified code segments of the embedded application;

3. estimating/verifying the latency, speedup, power consumption and area of the extensible instructions;

4. selecting the extensible instructions based on the design constraints.

Some approaches use libraries of pre-designed extensible instructions in order to limit the design space and then efficiently search within that space. Thus if an instruction is already designed, then that instruction can be reused in another application, as long as that instruction matches a code segment of the application (see the grey section of Fig. 1). The MINCE approach focuses on this key problem of the ASIP design flow, namely the *matching* – i.e. automatically matching candidate instructions of an embedded application (given in a high-level language like C/C++) to pre-designed library of extensible instructions that will enhance the core instruction set of the embedded processor. This key step compliments our existing ASIP framework.

The following describes some basics of binary decision diagrams (BDD) and gives a rough motivational idea on the techniques used in the MINCE tool. A Reduced Ordered BDD (ROBDD, but often simply referred to as "BDD") is a canonical data structure that uniquely represents a boolean function with the maximal sharing of substructure [10]. Dynamic variable ordering is often applied to change the order of the variable continuously (without changing the original function being represented) while the BDD application is running in order to minimize memory requirements [25]. There are many derivatives of BDDs such as the multi-valued BDD (MDD) -
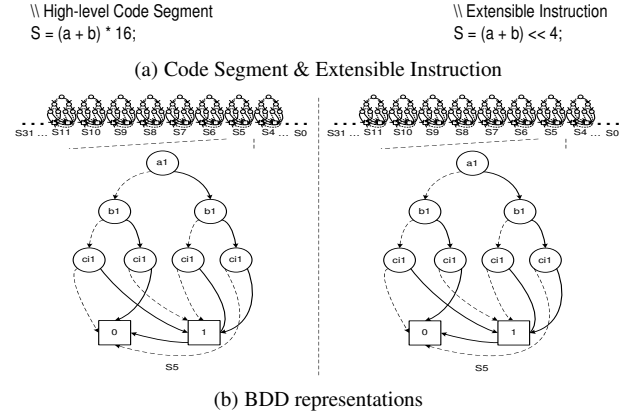
which has more than two branches and potentially has a better ordering etc. Using BDDs to solve combinational equivalence checking problem was proposed by Madre et al. [22, 23]. In brief, if two functions have the same functionality (but they have different circuit representations), then their BDDs will still be identical.

For example, Fig. 2a shows the high level language representation for a code segment ($S = (a + b) * 16$) and an extensible instruction ($S = (a + b) << 4$) (both are functionally equivalent). Fig. 2b shows the BDD representation of the code segment and the extensible instruction. Since there are 32 bits in each variable (a, b, S), the BDDs of variable S (bit 11 to bit 4) are shown. One of these (bit 5 of variable S) is expanded out for clarity. Note that the c_i in the BDDs in Fig. 2b is the carry in of each bit. The BDD representation of extensible instruction is identical to the BDD representation of the code segment, which indicates, that both the code segment and the extensible instruction are both functionally equivalent.

3   Related Work

Related work is twofold: first, there is work for automating the one or the step of an ASIP design flow with extensible instruction capabilities. Secondly there is work related to automatically matching/identifying software language constructs to equivalent hardware descriptions. We give a non-exhaustive overview of both.

Starting with the first group, Lee et al. in [20] proposed a design flow with instruction encoding, complex instruction generation, and a heuristic design space exploration in order to reduce the design-turn-around time for ASIPs. Their speedup of complex instruction is mainly achieved through reducing the size of the op-codes and operands and shortening the instruction fetch/decode time. Secondly, in [12], the design flow includes a methodology for rapidly selecting extensible instructions in a pre-designed instruction library. There, the extensible instructions are optimally but manually designed. In [29] the design flow comprises the generating of 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. Automatic instruction generation is based on matching regular templates in the graph only and then selecting the combination of the regular templates using a graph representation and algorithm.

On the other side, matching hardware to a software code segment has been attempted in various forms during the last
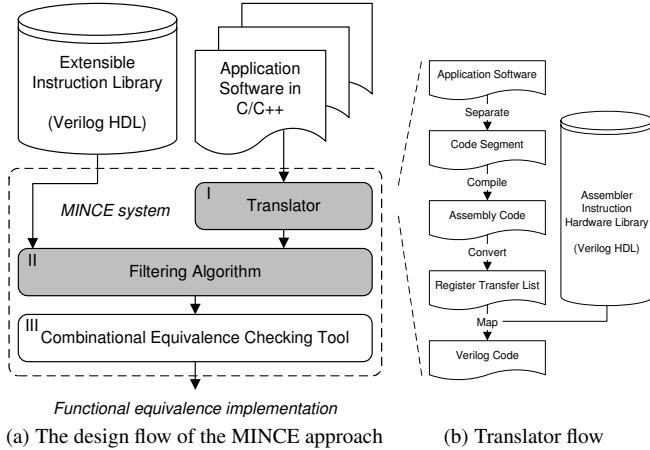
(a) The design flow of the MINCE approach   (b) Translator flow

Figure 3. MINCE: an automated tool for matching extensible instructions

decade and can be categorized into three research disciplines: graph matching approaches [14, 18, 21, 27], extensive simulation [28], and equivalence verification [13, 24, 26].

Graph matching approaches can be further divided into the template/pattern matching [14, 18] and instruction-set matching [21, 27]. These approaches are based on the concept of graph representation such as control/data flow graphs (CDFG), and then heuristic algorithms are applied to search the equivalent pre-defined template instructions in the graph representation of the software application. The limitation of this approach is that only instructions with structurally equivalent templates/patterns can be matched. Since extensible instructions often contain special modules to meet design constraints, it is practically infeasible to find a structural match.

Extensive simulation using Instruction-Set Simulator (ISS) enables the matching of functional equivalent instruction with corresponding software code segments [1, 6]. However, this approach requires the designer to locate the corresponding software code segment manually. Furthermore, a large data set is required to be simulated in order to ensure the functional equivalence of an instruction. Hence, the simulation approach is a very time-consuming process.

Several tools for verifying the combinational equivalence between C/C++ code and an HDL description have recently appeared [13, 24, 26]. In 1998, Pnueli, Siegel and Shtrichman introduced the idea of verifying the equivalence (safety-critical) of a software implementation in C with a small BDD transition model [24]. However, the C program is restricted to a subset of C. Semeria et al. developed a tool for verifying the combinational equivalence of RTL-C and an HDL in [26]. Once again, the C code is only limited to a subset of C, which is very close to the hardware description (RTL code). In other words, the C code needs to be written in a very similar way to the RTL code. Recently, Clarke et al. presented a tool for verifying the behavioral consistency of C and Verilog HDL programs [13]. This tool translates both C and Verilog HDL to bit vector equations, then the two bit vector equations are translated to SAT instances which are used to verify the equivalence using a bounded model checker. In fact, our MINCE tool extends their approach to verify an extensible instruction and a C software code segment which does not require the insertion of extra functions in the C program.

4 Overview of MINCE

The MINCE tool (shown in Fig. 3a) for matching extensible instructions consists of a translator, a filtering algorithm and a combinational equivalence model checker. The instruction library (containing pre-designed extensible instructions in Verilog HDL) and the application (in C/C++) are the input to our tool. The initial step of the flow is to separate the application into code segments which are suitable for matching with the instruction in the library. The choice of these segments is left to the designer, though approaches to choose these segments have been presented [7], for example. The next step of the flow is to convert a code segment to a Verilog HDL using our translator. There are two reasons for converting a code segment to a Verilog HDL:

1. the extensible instruction is designed in Verilog HDL and hence no manipulation is required if the verification tool uses Verilog HDL files as input as well;

2. the granularity even of small code segments in C/C++ is high, and hence would slow down the verification time significantly.

After that, we apply the filtering algorithm to eliminate instructions which will not match with any code segments. The instructions which pass through the filter are then compared one by one with the code segment using a combinational equivalence checking tool. The tool is called Verification Interfacing with Synthesis (VIS), which was jointly developed by the University of California, in Berkeley and the University of Colorado, Boulder [9].

4.1 The Translator

The translator flow is illustrated in Fig. 3b. The input code segments are obtained from a complete application written in C/C++ which is profiled and then segmented, according to a ranking criteria which is described in [12]. The assembler instruction hardware library contains individual assembler instructions implemented in Verilog HDL. We refer to these instructions in hardware as "base hardware modules". These hardware modules are used for technology mapping in our translator.

The C/C++ code segment is first translated into assembly achieving the following objectives:

- it uses all of the optimization methods which are available to the compiler to reduce the size of the compiled code;

- it converts the translated code into the same data types as the instructions in the library;

- it also unrolls loops with deterministic loop counts in order to convert the code segment to a combinational implementation.

An example of this step (code segment to assembler) is shown in Fig. 4 step II. The software code segment in the example contains addition, multiplication and shift right operations (*mult* - multiplication, *move* - move register, *sar* - shift right, and *add* - addition). The reason the assembly code contains a *move* instruction is that the *mult* produces a 64-bit product, and hence the *move* instruction is used to reduce the size of the product to 32-bit data.

The assembler code is then transformed into a list of register transfer operations. The translator converts each assembly instruction into a series of register transfers. The main goal of this conversion step is to convert any non-register transfer type operations, such as *pop* and *push* instructions, into explicit register transfer operations. In this step MINCE renames the variables in order to remove duplicate name assignments automatically. Duplicate names are avoided as Verilog HDL is a static single assignment form language [15]. In the example given in Fig. 4, this is shown as step III. In this example, the
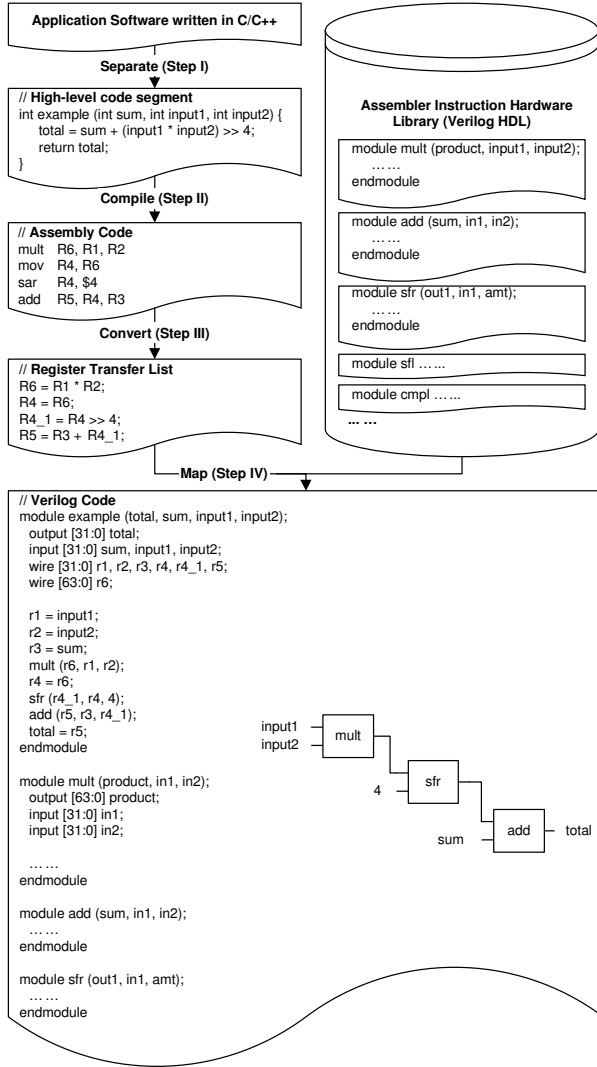
Figure 4. An example for translating to Verilog in a form that allows matching through the equivalence checker

Complex Module	Implementation - Hardware Module
Multiplier (32-bit)	Add, Shift
Multiplier (32-bit)	Multiplier (16-bit), Adder, Multiplexor
Division (32-bit)	Multiplier (32-bit), Reciprocal
Division (32-bit)	Subtract, Shift
Square Root (32-bit)	Multiplier (32-bit), Add, Subtract
Sine (32-bit)	Multiplier (32-bit), Add, Subtract
Cosine (32-bit)	Multiplier (32-bit), Add, Subtract

Table 1. A subset of complex module with limited implementations

```
Algorithm Filtering (v₁, v₂) {
    if (Σ input(v₁) != Σ input(v₂)) return filtered;
    if (Σ output(v₁) != Σ output(v₂)) return filtered;
    if (Σ |input(v₁)| != Σ |input(v₂)|) return filtered;
    if (Σ |output(v₁)| != Σ |output(v₂)|) return filtered;
    for all modules v₂ do {
        if (modules(v₂) == complex_module)
            cm_list = implement(modules(v₂));
    }
    for all element i in cm_list do {
        if (cm_listᵢ ⊆ Σ modules(v₁)) return potentially_equal;
    }
    return filtered;
}
```

Figure 5. Algorithm Filtering for eliminating the number of extensible instructions into the equivalence checking model

instructions might not directly map into a single base hardware module. Those instructions map to more complex hardware modules.

4.2 Filtering Algorithm in MINCE

Some code segments can be pruned as a non-match due to

- differing number of ports (the code segment might have two inputs, while the extensible instruction only one);
- differing port sizes;
- insufficient number of base hardware modules (for example, if the code segment just contained an XOR gate and an AND gate, while the extensible instruction contained a multiplier, then a match would be impossible).

The pruning filter greatly decreases the evaluation time of MINCE.

Fig. 5 presents the pseudo code of the filtering algorithm. It takes two Verilog HDL files as inputs: the converted software code segment and the extensible instruction in the library. The steps: First, it checks whether the number of the input/output variables and the size of the input/output variables are equal in both Verilog HDL representations. If not, it is pruned. In the other case, the filtering algorithm then determines whether the instruction contains any modules which are complex (see Table 1, column 1, for a subset of such modules). If the code segment does not contain the corresponding hardware modules given in column 2 of Table 1, then the instruction is eliminated. Note also that the matching may need to be recursive.

The complex modules in Table 1 were chosen, since these require extremely large BDDs (i.e. uses 1Gb RAM) to represent them and thus represent a good test of MINCE. The complexity of this filtering algorithm is $O(mno)$, where m is number of ways to implement a complex module, n is the number of complex instructions and o is the number of base hardware modules in an extensible instruction.

4.3 Instruction Matching Through Combinational Equivalence Checking

After filtering out unrelated instructions to the given code segment in the library, MINCE checks whether the Verilog

translator converts each assembly instruction into a single register transfer. The register transfer operations show the single assignment statement of each register, R4, R4_1, R5 and R6, where R4_1 is the variable renamed by our tool.

After the assembly code is converted to register transfer operations, the next step is the technology mapping (step IV of Fig. 4). In this step the register transfer operations are mapped to the base hardware modules given in the pre-designed assembler instruction hardware library. Once each register transfer has been mapped to a base hardware module, the translator creates a top-level Verilog HDL description interconnecting all the base hardware modules together. The Verilog code, shown in Fig. 4, is based upon the code segment and the register transfer operations. There are three input variables (sum, input1 and input2), one output variable (total), seven temporary connection variables (r1, r2, etc.) and three hardware modules (addition, multiplication and shift right) in this example. The top-level Verilog HDL declares the corresponding number of variables and contains the mapped code of the register transfer operations. The technology mapping step provides a system-level approach to converting register transfer operations to a combinational hardware module. One of the drawbacks to this approach is that control flow operations such as branch and jump

HDL converted from the software code segment is functionally equivalent to an instruction written in Verilog HDL. The checking is performed using VIS (Verification Interfacing with Synthesis) [9]. This part of the work could have been carried out with any similar verification tool.

We first convert both Verilog HDL files into an intermediate format, called BLIF-MV which VIS operates on, by a stand-alone compiler VL2MV [11]. The BLIF-MV hierarchy modules are then flattened to a gate level description. Note that VIS uses both BDDs and its extension the MDDs (multi-valued decision diagrams) to represent boolean and discrete functions. VIS is also able to apply dynamic variable ordering [25] to improve the possibility of convergence.

The two flattened combinational gate level descriptions are declared to be combinationally equivalent if they produce the same outputs for all combinations of inputs and MINCE declares the code segment and the extensible instruction to be functionally equivalent.

5 Experimental Setup & Results

The target ASIP compiler and profiler used in our experiments is the Xtensa processor's compiler and profiler from Tensilica, Inc. [6]. Our extensible instruction library is written in Verilog HDL as well as the assembler instruction library (See Fig. 3).

To evaluate the MINCE tool we conducted two separate sets of experiments. In the first, we created arbitrary diverse instructions and matched them against artificially generated C code segments. These segments either: a) matched exactly (i.e. they were structurally identical); b) were only functionally equivalent; c) the I/O ports match (i.e code segment passes through the filter algorithm but is not functionally equivalent); d) did not match at all. This set of experiment was conducted to show the efficiency of functional matching as opposed to finding a match through the simulation-based approach. In the simulation-based approach, the C code segment is compiled and is simulated to obtain results with a data set. The results are compared with the pre-computed result of the extensible instruction. The simulation was conducted with 100 million data sets each (approximately 5e-10% of the full data set with two 32-bit variables as inputs of the code segment). The reason for choosing 100 millions as the size of the data set is that the physical limits of the hard-drive. Each data set and each pre-simulated result of the instruction require approximately 1Gb of memory space. If more than n (n = 1 million (1% of the data set) for our experiments) differences occur in the simulation results, computation is terminated, we state that a match is non-existent.

The second set of experiments used real-life C/C++ applications (Mediabench) and automatically matched code segments to our pre-designed library of extensible instructions. We examined the effectiveness of the filtering algorithm by comparing the complete matching time including and excluding the filtering step. We selected the following applications, *adpcm encoder*, *g721 encoder*, *g721 decoder*, *gsm encoder*, *gsm decoder*, *mpeg2 decoder* from Mediabench site [19] and complete *voice recognition* system. All experiments were conducted on a Sun UltraSPARC III running at 900MHz (dual) with 4Gb of RAM.

5.1 Obtained Results and Discussion

Table 2 summaries the results of our first experiment. The first column indicates the type of instruction and the hardware modules it contains. The second column displays the type of software code segment (as compared to the instruction being matched) while the third column shows the number of corresponding code segments used in the experiment. The following column represents the average matching time of the

Instruction (Hardware module)	Software Code Segment	No of Code Segt.	Simulation Time [min.]	MINCE Time [min.]
Instruction 1 (Add, logical AND)	Exact Match	1	79	2
	Functional Equ.	3	82	3
	I/O Match only	3	< 1	< 1
	Do Not Match	3	< 1	< 1
Instruction 2 (Shift right, logical XOR)	Exact Match	1	46	2
	Functional Equ.	3	46	2
	I/O Match only	3	< 1	< 1
	Do Not Match	3	< 1	< 1
Instruction 3 (Add, Rotate shift right)	Exact Match	1	65	2
	Functional Equ.	3	65	3
	I/O Match only	3	< 1	< 1
	Do Not Match	3	< 1	< 1
Instruction 4 (Add, Shift left)	Exact Match	1	86	2
	Functional Equ.	3	87	3
	I/O Match only	3	< 1	2
	Do Not Match	3	< 1	< 1
Instruction 5 (Add, Shift right, logical AND)	Exact Match	1	41	2
	Functional Equ.	3	42	3
	I/O Match only	3	< 1	2
	Do Not Match	3	< 1	< 1
Instruction 6 (Add, shift, extra register)	Exact Match	1	49	10
	Functional Equ.	3	55	20
	I/O Match only	3	< 1	12
	Do Not Match	3	< 1	< 1
Instruction 7 (Shift right, multiplier)	Exact Match	1	85	60
	Functional Equ.	3	90	85
	I/O Match only	3	< 1	15
	Do Not Match	3	< 1	< 1
Instruction 8 (add, multiplier)	Exact Match	1	102	70
	Functional Equ.	3	105	75
	I/O Match only	3	< 1	20
	Do Not Match	3	< 1	< 1
Instruction 9 (Comparator, Shift left)	Exact Match	1	64	2
	Functional Equ.	3	65	10
	I/O Match only	3	< 1	7
	Do Not Match	3	< 1	< 1
Instruction 10 (Combine, logical XOR, logical OR)	Exact Match	1	35	5
	Functional Equ.	3	45	9
	I/O Match only	3	< 1	6
	Do Not Match	3	< 1	< 1

Table 2. Experimental results on hardware instructions on different kinds of software code segments

simulation-based approach for finding whether or not the extensible instruction is functionally equivalent with the corresponding software code segment. Finally, the last column displays the average matching time of MINCE. In our first experiment, we show that MINCE matches various quite diverse (since generated) software code segments successfully (in both experiments the correct result for all the software code segments were obtained). Our tool performed on average 8.8x (up to 39.5x) faster than the simulation-based approach. On the examples with complex instructions, our MINCE slowed down due to memory resource explosion during the creation of BDDs. Despite this observation, MINCE by far outperformed the simulation approach. Fig. 6 summaries the matching time of simulation vs. MINCE. Note that the simulation does not guarantee a match and is only a necessary condition, whereas the MINCE tool guarantees a match.

Table 3 summaries the results for matching instructions from the library to code segments in six different, real-life multimedia applications. We compare the number of instruction matched and time of matching extensible instructions with a reasonably experienced human ASIP designer and simulation-based approach. The ASIP designer selects the code segments manually and simulates code segments using 100 million data sets. The first column of table 3 indicates the application. The second column shows the speedup achieved by the ASIP designer and our MINCE tool. The third and forth columns represent the number of instructions matched and the matching time used by the ASIP designer & simulation-based approach
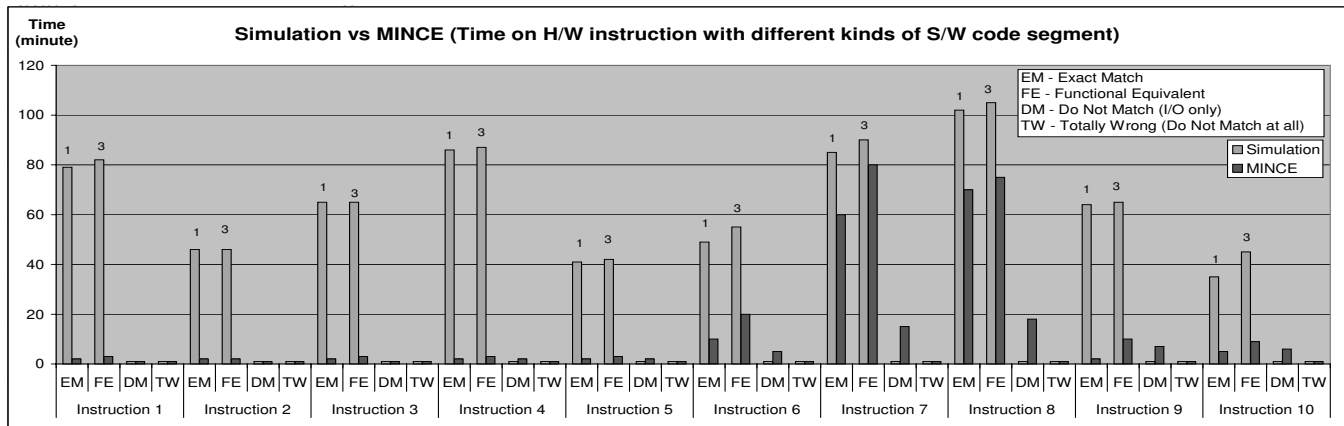
Figure 6. Results in terms of computation time for the instruction matching step: *Simulation* **vs.** *MINCE*

Application Software	Speedup [x]	ASIP Designer & Simulation		MINCE (w/o filtering alg.)		MINCE	
		No of Inst. matched	Time [hour]	No of Inst. matched	Time [hour]	No of Inst. matched	Time [hour]
Adpcm encoder	2.2	3	80	3	25	3	10
g721 encoder	2.5	4	75	4	20	4	8
g721 decoder	2.3	4	74	4	20	4	9
gsm encoder	1.1	4	105	4	40	4	25
gsm decoder	1.1	4	95	4	35	4	15
mpeg2 encoder	1.3	4	115	4	21	4	18
voice recognition	6.8	9	205	9	40	9	25

Table 3. Number of instructions matched, matching time used and speedup gained by different systems

respectively. The next two columns show the number of instructions matched and time used by MINCE (without the filtering algorithm). Finally, the last two columns displays the same characteristics by the MINCE tool. Our automated tool is on average 7.3x (up to 9.375x) faster than manually matching extensible instructions. We show that the effectiveness of the filtering algorithm, which reduces the equivalence checking time by more than half (compare column six and eight). In addition, we show the speed-up of the embedded application that could be achieved through the automatic matching, which is 2.47x on average (up to 6.8x). Note also that the identical matches were made by both the human designer and our MINCE system.

6 Conclusions

We have presented the MINCE tool as part of an ASIP design framework. MINCE translates selected code segments of an embedded application to a hardware description, filters out those code segments that would not match and eventually matches code segments to a pre-defined library of extensible instructions using functional equivalence checking. We have shown in experiments using the applications of the Mediabench suite that our approach is feasible as the tool was able to automatically match application code segments to extensible instructions in the library. Thereby, the time for matching was on average 7.3x faster than a simulation based approach that has been the state-of-the-art in ASIP design so far. We have also evaluated the speedup of the embedded application that could be achieved through the automatic matching which is 2.47x on average and therefore identical to a hand-optimized design (optimum solution).

It is therefore the first computationally feasible approach to fully automate an ASIP design flow by filling the missing gap of instruction matching.

What is currently not yet solved in our system is the matching of complex code segments that include not only data operations but also control statements. This will be part of our future work.

7 References

[1] Arctangent processor. ARC International. (http://www.arc.com).
[2] Asip-meister. (http://www.eda-meister.org/asip-meister/).
[3] Jazz dsp. Improv Systems Inc. (http://www.improvsys.com).
[4] Lexra processor. Lexra Inc. (http://www.lexra.com).
[5] Lisatek. CoWare Inc. (http://www.coware.com).
[6] Xtensa processor. Tensilica Inc. (http://www.tensilica.com).
[7] K. Atasu, L. Pozzi, and P. Lenne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *DAC*, 2003.
[8] 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.
[9] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, et al. Vis: a system for verification and synthesis. In *CAV*, 1996.
[10] R. B. Bryant. Graph-based algorithms for boolean function maipulation. In *IEEE Transactions on Computers*, August, 1986.
[11] S. Cheng, R. Brayton, G. York, et al. Compiling verilog into timed finite state machines. In *Verilog HDL Conference*, 1995.
[12] N. Cheung, J. Henkel, and S. Parameswaran. Rapid configuration & instruction selection for an asip: A case study. In *DATE*, 2003.
[13] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *DAC*, 2003.
[14] M. Corazao, M. Khalaf, et al. Instruction set mapping for performance optimization. In *ICCAD*, 1993.
[15] R. Cytron, J. Ferrante, B. Rosen, et al. An efficient method of computing static single assignment form. In *TOPLAS*, 1989.
[16] T. V. K. Gupta, P. Sharma, M. Balakrishnan, and S. Malik. Processor evaluation in an embedded systems design environmen. In *VLSI Design*, 2000.
[17] M. K. Jain, L. Wehmeyer, S. Steinke, P. Marwedel, and M. Bal-akrishnan. Evaluating register file size in asip design. In *CODES*, 2001.
[18] K. Kang and K. Choe. On the automatic generation of instruction selector using bottom-up tree pattern matching, 1995.
[19] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communica-tons systems. In *International Symposium on Microarchitecture*, 1997.
[20] J. Lee, K. Choi, and N. Dutt. Efficient instruction encoding for automatic instruction set desifn of configurable asips. In *ICCAD*, 2002.
[21] C. Liem, T. May, and P. Paulin. Instruction-set matching and selection for dsp and asip code generation. In *EDAC*, 94.
[22] J. C. Madre and J. P. Billion. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *DAC*, 1989.
[23] J. C. Madre, O. Coudert, and J. P. Billion. Automating the diagnosis and the rectification of design errors with priam. In *ICCAD*, 1989.
[24] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (cvt) - automatic verification of a compilation process. In *Int. Journal of Software Tools for Technology Transfer (STTT)*, 1998.
[25] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD*, 1993.
[26] L. Semeria, A. Seqwright, et al. Rtl c-based methodology for designing and verifying a multi-threaded processor. In *DAC*, 2002.
[27] J. Shu, T. Wilson, and D. Banerji. Instruction-set matching and ga-based selection for embedded-processor code generation. In *VLSI-Design*, 1996.
[28] M. Stadler, T. Rower, H. Kaeslin, et al. Functional verification of intellectual properties (ip): a simulation-based solution for an application-specific instruction-set processor. In *ISSS*, 1999.
[29] F. Sun, S. Ravi, A. Raghunathan, and N. Jha. Synthesis of custom processors based on extensible platforms. In *ICCAD*, 2002.