General Overview of an Adaptive Dynamic Extensible Processor

Hamid Noori, Kazuaki Murakami and Koji Inoue

Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University, 6-1 Kasuga-koen, Kasuga, Fukuoka, 816-8580, Japan
noori@c.csce.kyushu-u.ac.jp
{murakami, inoue}@i.kyushu-u.ac.jp

Abstract

This paper describes an approach for adaptive dynamic instruction set extension, tuning embedded processors to specific applications which are going to be executed in their life-time. These new instructions are generated after production. The processor has two modes: training mode and normal mode. Training mode can be done offline or online. The application-specific instructions are extracted from the critical portions of the code detected by a binary-level profiler at training mode. At normal mode custom instructions are executed on a reconfigurable coarse grain accelerator. The sequencer (an augmented hardware to the base processor) decides when, which custom instruction should be executed and switches between functional unit and accelerator outputs. It also takes the responsibility for switching between different accelerator configurations. In this methodology there is no need to a new compiler, extra opcodes or recompiling the source code. This paper provides a general overview of the system.

1. Introduction

Although availability of tools, programmability, and ability to rapidly deploy general purpose processors (GPPs) in embedded systems are good reasons for the common use of GPPs in embedded systems, they do not offer the necessary performance.

The application-specific nature of embedded system creates new opportunities to customize processor architecture for a particular application. Application specific instruction set processors, or ASIPs, have the potential to meet the challenging high-performance demands of embedded applications. The synthesis ASIPs traditionally involved the generation of a complete instruction set architecture (ISA) for the targeted application. However, this full-custom solution is too expensive and has long design turnaround times.

Another method for providing enhanced performance in processors is application-specific instruction set extension. By creating application-specific extensions to an instruction set, the critical portions of an application's dataflow graph (DFG) can be accelerated by mapping them to custom functional units. Though not as effective as ASICs, instruction set extensions improve performance and reduce energy consumption of processors. Instruction set extensions also maintain a degree of system programmability, which enables them to be utilized with more flexibility. The main problem with this method is that there are significant non-recurring engineering costs associated with implementing them. The addition of instruction set extensions to a baseline processor brings along with it many of the issues associated with designing a brand new processor in the first place.

The recent emergence of configurable and extensible processors has met with favorable tradeoff between efficiency and flexibility, while keeping design turnaround times short. The important motivation toward specialization of existing processors versus the design of complete ASIPs is to avoid the complexity of a complete processor and toolset development. Xtensa from Tensilica [23], Stretch S5 engine from Stretch [24], ARCtangent from ARC [25], and LISATek products from CoWare [26] are some commercial examples for these kinds of processors.

In the design of embedded systems-on-chip, the success of a product generation depends on the efficiency and flexibility to accommodate future design changes. Flexibility allows system designs to be easily modified or enhanced in response to bugs, market shifts, evolution of standards, or user requirements, during the design cycle and even after production which also means increase in productivity. Efficient implementations are required to meet the tight cost, timing, and power constraints present in embedded systems. Efficiency and flexibility are critical, but usually conflicting, design goals in embedded system design. While efficiency is obtained through custom hardwired implementations, flexibility is best provided through programmable implementations.

This paper presents a work-in-progress of adaptive dynamic extensible processor architecture. In this methodology the processor looks for the custom instructions (CIs) in the frequently executed portions of the code detected during binary-level profiling. The custom instruction selection is done automatically and transparently in the training mode, before the processor starts the normal operation. Then in the normal mode, the CIs are executed on a coarse grain reconfigurable accelerator. The sequencer mainly determines the microcode execution sequence. The CIs are detected and added after fabrication. Due to the features of the processor, it does not need to a new compiler, new opcode for CIs or source code recompiling. The processor has been developed by augmenting extra hardware to a single or four-issue in-order RISC processor.

This paper is organized as follows: In Section 2 we highlight related work. The general overview of processor architecture will be described in Section 3. In Section 4 our method for critical regions detection is discussed. Section 5 explains how custom instructions are generated and gives some results. And finally, paper is closed by conclusion and future work.

2. Related work

Identifying optimal set of custom instruction to improve the computational efficiency of applications has received a lot of attention recently. In contrast to domain instruction set extension (e.g. multimedia instructions of Intel and AMD processors) in which, adhoc techniques are typically exploited, recent research in extensible processors has mainly been revolving around automatic instruction generation/selection.

In [1], [6] the authors describe methods to generate custom instructions from operation patterns. Methods to identify custom instructions from applications are described in [14]. In [13], the authors search for regular templates in a dataflow graph and generate complex instructions by grouping the primitive ones. An automatic system to generate extensible instructions is described in [8]. In [17], a semi-automated method for detection and exploitation of custom instructions for VLIW processors is provided. The work in [18] describes a method to generate complex instructions from primitive ones.

While studying these techniques is necessary to found a good base, we are looking for some light weight tools that can be run on the embedded processor. Also we are not going to develop any new compiler. These techniques are computationally intensive and usually need a new compiler for supporting extended instructions. Additionally, these techniques are typically used for ASIPs where the underlying computational hardware for executing extended instructions is not already available. But in our architecture there is a reconfigurable accelerator for handling CIs.

A large body of research has gone into the design of reconfigurable functional units and accelerators. Examples include Chimaera [3], XiRisc [5], OneChip [10] and Garp [11] with tightly coupled fine grain accelerators and MorphoSys [4] and ADRES [12] with coarse grain accelerators. Although fine granularity brings more flexibility, it suffers from larger configuration memory, slower hardware and longer configuration time.

Some adaptive dynamic optimization systems such as, Turboscalar [7], rePlay [19], PARROT [20], and Warp Processors [21] were studied. These systems select frequently executed regions of the code through dynamic profiling, optimize the selected regions, cache/rewrite the optimized version for future occurrences. The execution of the optimized version is carried on by extra tasks sharing the main processor and/or by extra hardware. We also do binary-level profiling. But we use other methods than trace cache for profiling and besides we utilize smaller parts of the code for acceleration.

[2] has almost the same goal as ours but with different methodology. It uses rePlay framework, which utilizes trace cache for profiling, for CI generation. We believe our profiler is simpler comparing trace cache. Unlike their method, a sequencer is used for subgraph replacement.

3. General Overview of the Processor Architecture & Functionality

By *Adaptive* we mean that the processor can tune its extended instructions to the running applications. And we claim it is *Dynamic*, because instruction set extension is done after production and even at run-time in the gap between two consecutive executions of the application (e.g. printers, cell phone and etc). Instructions set extensions are going to be done fully automatically and transparently.

3.1 Architecture

Our **A**daptive dyna**M**ic extensi**B**l**E** processo**R** (AMBER) has been designed and developed by adding some units to a general RISC processor. Figure 1 depicts the added sections to the baseline processor, which has been proposed for AMBER. The figure shows a general view of the idea and concept.

The baseline processor can be a single or four-issue in-order RISC processor. There are three main units that have been augmented to the baseline processor: a profiler, a coarse grain reconfigurable accelerator and a sequencer.

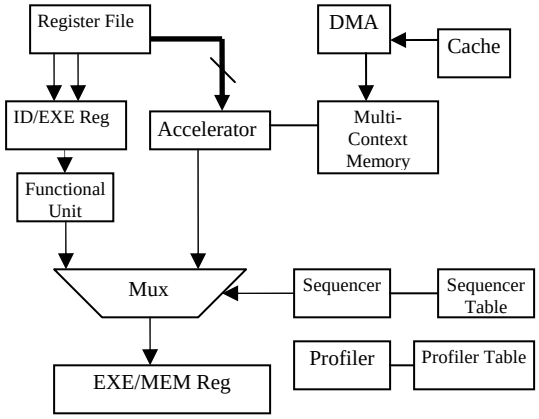

Fig. 1. General overview of the architecture

This processor has two modes: training mode and normal mode. In training mode the processor learns about application-specific instructions and then generates the proper and necessary configuration data. In normal mode, although it can keep on profiling, it does not generate new configuration. It just uses the configured architecture for running the application including the execution of CIs on the accelerator. Training mode can be done offline or online. If it is done offline a simulator (e.g Simplescalar [16]) will be needed to run the applications that are going to be executed on the processor and gather the profiling information. In this case AMBER does not need hardware for profiling. But if it is going to be done online, then some hardware will be required. The profiler hardware contains two registers to store the previous program counter (PC) and current PC, a comparator to compare the values of the two registers and a table to store the start address of hot basic blocks (HBBs) and their corresponding execution frequency. In online training mode, tools for generating CI and configuration data are run on the base processor. More information about profiling is given in the following section.

As it has been also mentioned in [2], [9], according to the size of data in the processors, a matrix of functional units (FUs) seems efficient and reasonable enough for accelerating dataflow subgraphs as CIs. Using coarse grain reconfigurable accelerators demand for less configuration memory. Also mapping instructions on them will be easier. Although fine grain accelerators are more flexible, they are slower comparing coarse grain ones. We assume that each FU of the accelerator supports all fixed-point instructions of the baseline processor except multiplication, division and load. Using some selectors, the inputs of accelerator are directly connected to the outputs of the registers of the register file. Therefore, accelerator does not need to read the data from the register file. The outputs of FUs in each row are fully connected just to the inputs of the FUs in the subsequent row. The accelerator has a variable delay. The delay of the accelerator depends on the number of rows that is required by a mapped CI. The execution delay of each CI is known at training mode after mapping CIs on the accelerator. The accelerator has a two-level configuration memory: a multi-context memory and a cache. The configurations of custom instructions that are most probable to be executed in near future are stored in the multi-context memory and the others in the cache. Using two-level configuration memory can hide the overhead of loading new configuration from cache to multi-context memory. The size of these memories limits the number of CIs.

The sequencer mainly determines the microcode execution sequence by selecting between the accelerator and the processor functional unit outputs. It has a table in which the start addresses of dataflow subgraphs, which are going to be executed as custom instructions on the accelerator, are specified. The table of the sequencer is initialized according to the locations of the custom instructions in the object code at the training mode, when they are generated.

3.2 Functionality

We believe that it is fundamental to handle the instruction set extensions in a fully automated manner. On the other hand we are looking for a transparent post-production instruction set extension approach without developing a new compiler. Automated instruction-set extension tools are time consuming so that they can not be applied dynamically while the processor is doing its duties.

Therefore two modes have been defined for the processor: training mode and normal mode. In the training mode, the user runs desired applications with favorite inputs. The applications are executed on the processor using only the processor functional unit as usual (online training mode). Meanwhile, the profiler starts profiling. According to the information gathered by the profiler, critical regions are detected. Utilizing the critical regions the CIs are generated. More information about the profiler, critical regions detection and CI generation will be given in following section.

For some embedded applications, there are some gaps between consecutive executions of the applications (e.g. printer and cell phones). In these systems the processor can keep on profiling even at normal mode. When the application execution finishes, it switches to training mode and applies the tools again to update or optimize more the custom instructions. For those embedded applications without this feature, the processor enters the training mode (or training can be done offline) at first once, and then it switches to the normal mode. In the training mode, also data for accelerator configuration memories are generated and loaded into the cache and multi-context memory. The sequencer table is also initialized in this mode.

After finishing CI and configuration data generation in training mode the processor switches to the normal mode. In the normal mode, using the accelerator, its configuration data, the sequencer and its table, the CIs are executed on the accelerator. The sequencer monitors the PC and compares to its table entries. When it detects that a CI is going to be executed in near future, it checks whether the corresponding configuration is available in the multi-context memory or not. If it exists, the multi-context memory selects the proper configuration for the accelerator; the sequencer switches from processor functional unit to the accelerator, waits for specified clock cycles (CIs are mapped on the accelerator at training mode therefore it is known how many cycles is necessary to execute each CI) and let the accelerator finish the execution of the custom instruction and again switches to the processor functional unit. If the configuration is not available in the multi-context memory and there is enough time, the configuration data will be loaded to the multi-context memory from cache. Otherwise the original code will be executed on the processor functional unit as usual. Therefore in the case of absence of configuration of a custom instruction we will only miss the expected speedup for that instruction and there will be no penalty. Each CI generates proper PC after its execution finishes, considering original sequence execution, so that processor can continue from correct address.

By comparing the PC and the contents of the table (the sequencer table contains the start address of CIs), the sequencer can distinguish which custom instruction will be executed at what time. Therefore the sequencer knows when it should switch between the outputs of the accelerator and the functional unit. By that time the corresponding CI configuration is loaded to the accelerator. Because the outputs of registers in register file are already available at the inputs of the accelerator, it does not need to read the registers. The sequencer will send the result of accelerator to the next stage after waiting for determined number of clock cycles (the custom instruction are assumed to be multi-cycle) and again switches to the output of functional unit.

One of the advantages of using the sequencer is that it obviates the need for adding new opcodes for the custom instructions. The disadvantage is that we have to generate different configurations for custom instructions that have the same operations but different operands. Using the proposed architecture and hardware obviates the need for developing a new compiler and recompiling the source code.

4. Detecting Critical Regions of Code

CIs are extracted from hot basic blocks (HBBs). CI generation has been limited to one HBB. A basic block is a sequence of instructions that ends in a control instruction. Therefore the basic block has one branch or jump instruction which is the last instruction. HBBs are basic blocks that are executed more than a specified threshold. Start address of basic blocks are determined by the profiler hardware monitoring the PC (if the training mode is supposed to be done online. For offline training mode, profiler routine monitors the simulator). Figure 2 shows the flow for generating CIs.

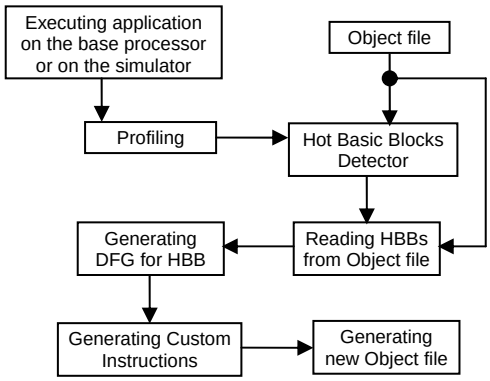

Fig. 2. Custom instructions generation flow

Usually the critical regions are in loops or functions. Therefore our profiler looks for jumps and branches by monitoring PC. In every clock cycle, the profiler hardware compares the current value and the previous value of PC. If the difference of these two values is not equal to the instruction length, a taken branch or jump has occurred. The profiler has a table with a counter for each entry, which keeps the execution frequency. In the case of taken branch/jump detection by the profiler, the profiler table is checked. If the target address (current PC) is in the table, the corresponding counter is incremented, otherwise current PC is added as a new entry and the corresponding counter is initialized to one.

After application execution (and profiling) then HBB detector software reviews the table and selects the entries (target addresses) with counters more than a specified threshold. These addresses determine the start address of HBBs. Using these addresses, the HBBs are read from the object code. Reading an HBB is terminated as a control instruction is seen. In this method, if the branch instruction of a hot basic block is mostly no taken, the following basic block of this HBB is also hot, but by just looking to the profiler table it can not be detected.

To solve this issue, after detecting the start addresses of the HBBs, the HBB detector reads the HBBs from the object code and checks their last instructions. If the last instruction is branch (not jump), the branch target address (BTAs), the counter of the current HBB and the start address of not taken part are saved in a new list. The counter shows that how many times the branch has been executed. Jump instructions are always taken, so their target can be detected by looking into the table. Therefore, we have to check just the branches.

After saving these values for detected HBBs in the new list, all branch target addresses (BTAs) of the new list, are checked if they are in the HBB list or not. If a BTA is in the current HBB list, then it is ignored. Otherwise the branch target address of the new list is searched in the profiler table. If the BTA of the new list can be found in the profiler table, then the counter of the profiler table is subtracted from the corresponding counter of the BTA of the new list. The counter of the profiler table shows how many times the branch is taken and the counter of the new list shows the execution frequency of branch instruction of the HBB. By comparing the result of subtraction to the threshold value it can be distinguished if the not taken direction is hot or not. If it is hot, the not taken start address is added to the HBB list as a new entry otherwise it is ignored. If the BTA of the new list can not be found in the profiler table, it means that this branch is always not taken which means that the not taken part is hot. In this case, the not taken start address is added to the HBB list as a new entry. This algorithm is run again for every new HBB entry and continues until no new HBB is found.

Simplescalar tool set (PISA configuration) [16] and Mibench [15] (a free, commercially representative embedded benchmark suite) have been used, to determine the required number of entries for the profiler table. The sim-safe tool was modified to write PC of committed instructions in a file. Then different applications of Mibench (basicmath, cjpeg, djpeg, lame, dijkstra, patricia, blowfish, sha, gsm, rijndael, crc, fft and adpcm) were run on the sim-safe. After that our profiler used the file generated by sim-safe, which contains sequences of PCs of retired instructions. Reviewing the results of running our profiler for different applications show that that the maximum number of basic blocks is 2149. The number of detected basic blocks by the profiler also shows the required entries for our profiler table because these two values have the same concept. By using replacement policies (such as methods that have been used in [22]) which replace low frequent basic block's start addresses with higher frequent ones in the profiler table smaller profiler table will be needed. The first row of Table 1 shows the floor threshold for execution frequency of basic blocks and the second row depicts the maximum number of basic blocks that have an execution frequency more than the numbers in the first row. According to Table 1, a profiler table with 512 entries and a floor threshold value of 512 or 1024 for low frequent basic blocks replacement (which seem small enough comparing the floor threshold for HBB detection) can be suitable choices for our HBB detector. The threshold that is used for detecting HBBs varies for each application and ranges from 8K to 2000K.

Floor threshold for exec freq of basic blocks	**128**	**256**	**512**	**1024**	**2048**
Max number of detected basic blocks	970	656	459	390	296

Table 1. Max number of detected basic blocks according to the floor threshold execution frequency of basic blocks

5. Generating Custom Instruction

Custom instructions are generated utilizing HBBs. Custom instructions can include all fixed-point instructions. Each CI can have at most one control (branch or jump) and one store. The custom instruction can not contain multiply, divide, floating-point or load instructions.

When the HBB detector detects the start address of HBBs then each HBB is read from the object code using the HBB start address. Then the dataflow graph (DFG) is generated for the HBB. The output of the DFG generator is given to the custom instruction generator routine. This routine first looks for the biggest sequence of instructions that can be executed on the accelerator (supported instructions). Then it tries to add more instructions to the head and tail of founded instruction sequence by moving instructions. Moving the instructions should not change the logic of the application. Figure 3 shows an example. The five blocks make an HBB. The algorithm looks for the

biggest supported instructions sequence and find block 3. If the instructions in block 1 and block 2 do not have flow-dependence and anti-dependence then block 1 and 2 are replaced. Otherwise the flow and anti dependence are checked for block 2 and 3. If there is no dependence then block 3 is moved up. The same is done for the lower blocks. When these kind of moving are done, the corresponding parts of object code should also be rewritten.

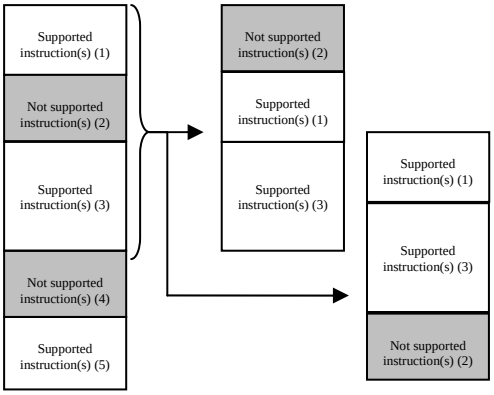

Fig. 3. Moving instructions to make a bigger CI

In figure 4 the percentage of CIs for different length is shown. The numbers which appear after the applications' name show the threshold value used for selecting HBBs. Due to the results; in most cases more than 70% of CIs have less than 6 instructions. Table 2 illustrates more information about the CIs and performance enhancement. The second column shows the number of executed and profiled instructions. Third column contains threshold values for selecting hot basic blocks. In the next column, number of detected HBBs has been written. The numbers in the fifth column specifies how many CIs could be extracted from the HBBs.

Sometimes, HBBs are very long so that several custom instructions can be extracted from one hot basic block (e.g. in JPEG) and sometimes hot basic blocks are too small or contain unsupported instructions so that no custom instruction can be extracted (e.g. *basicmath*). The sixth column shows the speedup versus the baseline processor (single issue in-order RISC processor). As it was expected, applications such as *sha*, *gsm*, and *rijndael* that have longer custom instructions could reach better performance. The last two columns show the percentage of the code size and execution time that were covered by the custom instructions, respectively. As it can be seen, a very small part of the object code is executed for many clock cycles.

As it has been mentioned before, the accelerator has been assumed to have a variable delay according to the height of mapped custom instruction. It has been presumed that the first row of the accelerator takes one clock cycle and the other rows, which do not have register read and write take 0.5 clock cycle for execution. For example, suppose that there is a CI containing nine instructions. After mapping this CI on the accelerator, it takes three rows of the accelerator. The first row takes one clock cycle and the second and third rows take 0.5 clock cycle. Therefore it takes two clock cycles for the CI to be executed on the accelerator. For N times execution of this CI; (9-2)*N clock cycles will be saved. At this stage, we did not consider any limitations for number of inputs, number of outputs, width and depth of the accelerator. Taking these assumptions, speedup was calculated.

6. Conclusions and Future Work

We have presented a general overview of an architecture for an adaptive dynamic extensible processor. This processor is capable to add application-specific instructions to its instruction set after production.

The architecture is based on an in-order single or four-issue RISC processor. A profiler, a reconfigurable accelerator and a sequencer has been augmented to the baseline processor and some modifications have to be applied to the register file and pipeline intermediate registers. This processor has two modes: training mode and normal mode. In training mode, using the profiler, it learns about CIs and generates the configuration data for its accelerator and initializes its sequencer table. In normal mode using the information generated in training mode, tries to run the CI on the accelerator. It does not need any new compiler and new opcodes for the extended instructions. Critical regions of code (hot basic blocks) are detected by using a simple hardware which monitors program counter of the processor. CIs are extracted from these regions. The preliminary performance evaluation shows a performance enhancement between 7.8% and 52% for some of Mibench programs.

As future work we will focus on more micro-architecture details and developing tool for mapping custom instruction on the accelerator. Also we try not to limit CIs to one HBB. Executing primitive instructions and custom instructions on processor functional unit and accelerator in parallel can also be considered as another task. Determining the threshold value should be done dynamically. Finally a complete simulator framework is highly required.

App.	Exe Instr (M)	Threshold (K)	No. HBB	No. CI	% Speedup	% code size	% exe time
basicmath	170	64	37	18	19.6	1.4	31.6
cjpeg	101	32	42	52	27	1.5	44
djpeg	25	8	22	32	31.5	0.8	48
lame	260	32	142	104	8.6	1.1	16
dijkstra	254	64	34	20	21.4	0.7	38.6
patricia	217	128	51	17	7.8	0.6	14.6
blowfish	260	128	18	28	33	2.7	59
rijndael (enc)	260	128	63	92	36	6.1	51.7
rijndael (dec)	259	128	63	78	36	4.5	51.7
sha	154	64	9	13	52	1.1	73
adpcm (enc)	260	2000	14	8	21	0.32	42
adpcm (dec)	265	2000	12	5	24	0.24	41
crc	265	512	4	2	20	0.1	44.9
fft	189	128	43	19	18.6	0.93	30
fft (inv)	190	128	43	19	18.6	0.93	30
gsm (cod)	265	128	34	41	25.1	1.53	47.2

Table 2. More details on custom instructions (Mibench programs)

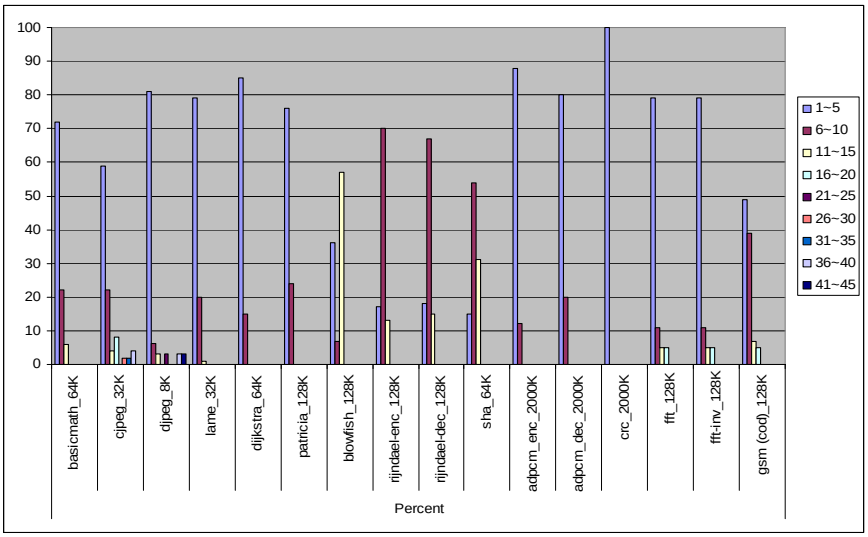

Fig. 4. Percentage of custom instruction according to their length

7. References

[1] Sun, F., Srivaths, R., Raghunathan, A., Jha, N.K.: Custom-Instruction Synthesis for Extensible-Processor Platforms. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 23, No. 2, 2004

[2] Clark, N., et al.: Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization. MICRO-37, 2004.

[3] Ye, Z.A., et al.: Chimaera: a high-performance architecture with tightly-coupled reconfigurable functional unit. Proceeding of 27th ISCA, pages 225-235, 2000.

[4] Parizi, H., et al..: MorphoSys: a Coarse Grain Reconfigurable Architecture for Multimedia Applications. Euro-Par 2002 Parallel Processing. Springer-Verlag. pp.844-8, 2002.

[5] Lodi, A., et al., R.: A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications. IEEE Journal of solid-state circuits, Vol. 38, No. 11, 2003.

[6] Atasu, K., Pozzi, L., Lenne, P.: Automatic application-specific instruction-set extensions under microarchitectural constraints. 40th Design Automation Conference, 2003.
[7] Black, B., Shen, J.P.: Turboscalar: A High Frequency High IPC Microarchitecture. ISCA- 27, 2000.
[8] Goodwin,D., et al.:Automatic generation of application specific processors. CASES, 2003.
[9] Clark, N., et al..: An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors. ISAC-32, 2005.
[10] Carrillo, J.E., Chow, P.: The effect of reconfigurable units in superscalar processors. Proceedings of the 2001 ACM/SIGDA FPGA, pages 141-150. ACM Press, 2001.
[11] Hauser, J.R., Wawrzynek., J.: GARP: A MIPS processor with a reconfigurable coprocessor. IEEE Symposium on FPGAs for Custom Computing Machines, 1997.
[12] Mei, B., et al.: ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix, FPL, 2003.
[13] Brisk, P., Kaplan, A., Kastner, R., Sarrafzadeh, M.: Instruction generation and regularity extraction for reconfigurable processor. CASES, 2002.
[14] Clark,N., et al.:Automatically generating Custom instruction set extensions. Workshop of Application-Specific Processors, 2002.
[15] http://www.eecs.umich.edu/mibench/
[16] http://www.simplescalar.com/
[17] Arnold, M., Corporaal, H.: Designing domain-specific processors. Proc. Int. Symp. HW/SW Codesign, pp. 61-66, 2001.
[18] Choi, H., et al.: Synthesis of application specific instructions for embedded DSP software. IEEE Trans. Computers, vol. 48, no. 6, pp. 603-614, 1999.
[19] Patel S., Lumetta, S.: rePlay: A Hardware Framework for Dynamic Optimization. IEEE Trans. on Computers, 50(6), pp 590-608, 2001.
[20] Rosner, R., Almog, Y., Moffie, M., Schwartz N., Mendelson, A.: Power Awareness through Selective Dynamically Optimized Traces. ISCA'04 (2004)
[21] http://www.cs.ucr.edu/~vahid/warp/
[22] M. C. Merten, A. R. Trick, C. N. George, J. Gyllenhaal, W. W. Hwu, "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization", ISCA 1999.
[23] Tensilica Inc., http://www.tensilica.com
[24] Stretch Inc., http://www.stretchinc.com
[25] ARC International, http://www.arc.com
[26] CoWare Inc., http://www.coware.com