Processor Optimization using Instruction Set Reduction: A brief Survey

Shilpa Srinivasan
Department of Computer Engineering
San Jose State University

Kunal Goswami
Department of Computer Engineering
San Jose State University

Aakansha Talati
Department of Computer Engineering
San Jose State University

Veena Manasa Kanakamalla
Department of Computer Engineering
San Jose State University

Nitesh Jain
Department of Computer Engineering
San Jose State University

***Abstract*— Various factors affect the performance of a given Processor. Instruction Set Architecture and Code are one of the major contributors to this. The introduction to novel extensions to ISA improves the design complexity and demands more hardware verification and debugging. To resolve these complexities and ensure the efficient performance of processor, compiler and architectural approaches are proposed. This paper aims at reducing the processor power consumption, code density, compiler optimization performance by achieving code optimization using various techniques like Instruction bit width, operand factorization, Instruction recycling, Instruction register file, packing Instruction into Register Window, Reducing Instruction cycle and Dynamically optimizing code of micro operations. The introduction to novel extensions to ISA improves the design complexity and demands more hardware verification and debugging. The experimental results show significant improvements in code size, power and additionally improvement in execution time.**

***Keywords*—Instruction Set Architecture; Processor Optimization; Code Compression; Microarchitecture Modification;**

I. INTRODUCTION

Instruction Set Architecture is the interface between hardware and software for any computing system. A program contains instructions written in a higher level language, however, the underlying hardware needs binary executable to be able to perform the same operations as the programmer intended. This function can be achieved by translating the given program into a lower level assembly language code, which on the other hand is translated into machine code. Essentially, in binary for the processor to understand. Every processor is bound by its primitives, whether it is the type and number of instructions which it supports or the memory capacity on the chip. These primitives make is necessary for a programmer to optimize the code in such a way that it makes the most effective use of the underlying processor. One of the major constraints on the program is the on-chip memory, the program has to be of a size which can be easily placed into the same memory. Making code compression a very important factor in optimization, to be able to perform the same number and nature of operations by using minimal amount of instructions is what we mean by code compression. Every instruction set architecture provides the programmer with different tools to modify the code and effectively use memory, but there're ways beyond the programming part which can optimize the performance such as the compression of the instruction opcode, compression of the repetitive patterns encountered in a translated program and compression with the help of packing instructions into a specific register file. They're not limited to what we mention here, however we underline some of the essential techniques in code compression which are wrapped around innovative ideas to general solutions to general problems applied specifically towards a better processor design and performance. We also explore the techniques such as instruction recycling and reduction of instruction cycles to optimize the performance of the processor.

References [1,2,3,4] contain various techniques to code compression which more or less address the limitations of the instruction set architecture. These limitations which solved together provide us with a better design of the processor and not only that, but also an improved perspective of looking at the same problem of code optimization. Furthermore, instruction recycling[9] and packing of instructions into a specific register file[5,7] take a different turn in providing optimization, earlier methods focus on the written code and not so much on the instructions while these two techniques extensively focus on the instructions for a given Instruction set architecture and deliver a solution which achieves the goal. Transparent Instruction set framework [10] also gives a way to increase the processor performance. We also explored that instruction fetch energy[8] leads to processor optimization by reduction.

The rest of this document is organized in the following manner. Section II contains the in-depth analysis of the various approaches to processor optimization which we explored. Section III contains the various advantages and disadvantages of these approaches by placing them under evaluation. Section IV contains the possible improvements to each and every approach which we covered to improve them even further. Followed by the conclusion over such an extensive study.

II. APPROACHES

Let us now go through various methods to achieve Instruction Set Architecture Code Optimization along with the details of the performance of each approach.

A. Code Optimization by code compression techniques

Lefurgy et al.[2] identify a crucial factor in any program, which is code repetition in the object code. For this code to be stored on an embedded processor, it would mean that the processor has to have enough memory to store all varied size of programs on top of it. Becoming the inspiration for an effective code compression technique which could facilitate the processor without increasing the performance cost on top of it.

A program is a set of instructions, which basically can be treated as text streams. And text streams can be compressed with the help of indexing common phrases with the help of a dictionary. The identification of every such phrase is an NP-Complete problem, however a good enough solution can be obtained with the help of "greedy" approach. Lefurgy et. al. propose microarchitecture modifications and a post compilation analyzer along with the text compression scheme to achieve code optimization. The methodology is just as simple as it sounds, to encode the repeating set of instructions into a set of code-words, the same code-words become the index of a dictionary which later on executes the original set of instructions used. This approach has an intrinsic advantage over sub-routine calls used for repetitive instructions, a hardware implementation which reduces the overhead of a sub-routine call. This hardware implementation, referred as "call-dictionary" is provided with the *location and length*, allowing the processor to go to a certain location and execute "length" number of instructions depending on that code-word. Such a technique has it's disadvantages as well, this "call-dictionary" method in particular limits the size of the dictionary which can be used with this approach.

Some of the issues faced by them include the branch instructions and the branch targets, this implementation leaves the branch instructions which require calculation of the address using the offset untouched. To avoid complications in the decoding process, the branches in the executable are patched to use the new addresses generated with the help of the underlying cache. The evaluation of this approach is carried out on PowerPC, ARM and i386 architectures with the help of fixed sized code-words and variable sized code-words. The following table illustrates the performance of this algorithm when the dictionary entry size is constrained to 4 instructions.

Benchmark	Maximum number of code words used
compress	72
gcc	7577
go	2674
ijpeg	1616
li	454
m88ksim	1289
perl	2132
vortex	2878

TABLE 1: Maximum number of codewords used in baseline compression.

They were able to obtain 39%, 34% and 26% code reduction in PowerPC, ARM and i386 architectures by keeping the dictionaries to contain a maximum of 16 bytes per entry. The experimentation with variable sized code-words only helped them conclude that a 2 byte code-words is better at providing a higher compression ratio than a 4 byte code-word. Also, for achieving a good compression ratio the number of code-words in the dictionary have to be increased rather than the size of a particular dictionary entry. Thus, determining the two crucial factors which can be adjusted to achieve a higher compression ratio. Lefurgy et. al. concluded that the size of the dictionary is most crucial in determining the compression ratio, followed by the fact that compression ratio can be significantly improved by reducing the code word size below the size of a single instruction.

B. Dynamic Optimization of Micro-Operations in x86 architecture

Complex instruction sets such as x86 contain inefficiencies in the post-compilation code. This code contains a set of micro-operations which are generated from the complex instructions present in the program. These operations might have redundancy amongst them, also might be having dead code containing no operation blocks within them. Leading to the motivation behind optimizing these fragments of code at a micro-operation level.

Slechta et al. [1] propose modifications to the rePLay framework which deals with the optimization in complex architectures. The micro-operations generated after the compilation are not visible to the compiler and hence have a tendency to possess inefficiencies. This framework focuses on the concept of "frames" containing mutually independent instructions along with every frame having the characteristics of atomicity, that is either no frame commits the changes or the frame commits the changes of all the instructions within it. This can be achieved with the help of removing all control dependencies within the frame, Slechta et al. improve this framework by providing a way to speculatively optimizing

around the memory dependencies and by providing a programmable optimizing engine datapath for the programmer to be able to use and configure. Some of the inefficiencies can be described by dead code elimination, NOP elimination, reassociation and store forwarding. However, the true nature of optimizations lies between the blocks, thereby dividing the optimizations into two categories of inter-block optimization and intra-block optimization.

1. Intra-block optimization

Reassociation would replace the usage of a particular micro-operation with the immediate result obtained from a previous instruction. Once the same result is no longer used by any of the instructions, dead code elimination comes into picture.

2. Inter-block optimization

Store forwarding associates the pair of load-store instructions present in the code. However, at times there's a need to preserve all live-out registers from the code which limits inter-block optimization via store forwarding as the optimization cannot remove the respective store.

These problems can be solved with the help of frame level optimization, which removes all control dependencies amongst the instructions present within various frames. The problems are solved with the help of converting branches into assertions which upon an untrue outcome schedule a hardware roll-over which is present in most of the modern processors. Along with additional optimizations of redundant code removal and dead code elimination.

For the evaluation of the same, they use a set of available benchmarks from AMD. From the results which they obtain by comparing ICache, Trace Cache, rePLay and rePLay with Optimization except gzip rePLay with optimization surpasses all other applications in the performance base. Given the fact that the rePLay optimizations reduce micro-operation count and the computation tree height for calculating the micro-operations. On an average the rePLay optimizer removed 21% of the micro-operations, 22% of the "load" instructions and demonstrated a 17% increase in the instruction-per-cycle(IPC).

C. Code Compression by reducing instruction bit width

In embedded processors which demands low power low cost reduced memory space techniques, having very long instruction word would increase power consumption and degrade efficiency. Instruction fetch is one of the primary contributor towards increased power dissipation in a processor. One of the method to improve this is by converting a long word instruction example 32 bit to compressed one example 16 bit with the help of remote operand array(ROA)

By this methodology all the instructions are converted into 16- - bit full instruction and 16-bit partial instruction.

The aim is convert the ISA into its equivalent reduced ISA format and not squeeze to them causing data loss. This compression technique is applicable only for instruction bits and not data bits. The ISA structure for this technique is modified to accommodate 8bit for opcode and D- bit and 8 bit is reserved for operand field. The full and partial instructions are distinguished on the basis of this D bit. This conversion is easier in case of R-type instruction as all the operands can be easily converted, but approach is different in case of I-type Jump and Branch instruction.

In this case the instruction is converted into equivalent 16-bit partial instruction and the immediate operand is stored in the ROA. At compile time the immediate operand value is bound to its 16-bit partial instruction and the complete 16-bit instruction is thus reconstructed. The remote operands stored in the ROA in orderly manner. When the compiler detects a D bit it extracts the first operand from the buffer and binds to the instruction. The hardware decides based on the opcode how many operands it has to collect from the ROA buffer. The ROA buffer works on the principle of FIFO queue. When the partial instructions are decoded, the corresponding remote operands are removed from the front end of the buffer and the instruction proceeds to the execution stage. The ROA slots that are added are obtained from the various NOP operation that are present in the code.

Thus using this approach effective die area with reduced power consumption and compressed code is achieved. The efficiency of this method could be scaled effectively by using aggressive compiler scheduling algorithms. The execution is increased by about 4% using this method but the code size is significantly reduced by 45% and for ideal cases it could be easily halved. The instruction fetch energy could be reduced to 45% compared to the normal execution of the energy.

D. Code Compression optimization using operand factorization

Program size can be minimized by designing a processor which can execute a compressed code. For the processor to do this, the decompression engine has to implement code decompression for the branch instructions as they cannot be compressed. One of the compression technique used is Operand factorization, a technique of separating the expression trees into (a) tree pattern(opcodes), a group of instructions in the expression tree after removing the operands and (b) a sequence of operands known as operand pattern, that consists of registers and immediate, resulting in removal of operands.

The efficiency of encoding technique is based on the compression ratio, ratio of size of compressed program to size of uncompressed program. The compression algorithm uses encoding techniques to encode the code patterns. There are total of four encoding methodologies, out of which two are Huffman and fixed length encoding and other two include consideration of effect of encoding in the performance of decompression engine. In bounded Huffman technique, an escape bit is added at the beginning of codeword, for

identification of type of encoding technique used. It is also used in MPEG-2 as a method for limiting the size of Huffman code word.in technique of VLC encoding, the bounded Huffman code-words are chosen such that the size of codeword can be known by leading zeros in it. This eases the extraction logic of code-words. The decompressed engine to decompress the code, works in two phases. Firstly, extraction of Tp and Op is done. Secondly, Tp is mapped into the sequence of uncompressed instructions while Opis utilized to generate registers and immediate bits. Then this data is fed to the instruction assembly buffer(IAB) where all the decompressed instructions are assembled. The decompression engine consists of three major modules:

Tree-Pattern generation:

The tree-pattern dictionary(TPD) is used to store the opcodes of each encoded tree-pattern code word. T_p is decoded and fed to TPD address Tpaddr and the opcodes encoded by T_p are fetched from TPD. This fetched sequence has three fields: OPCODE, ITYPE and END. The opcode consists of opcode bits, ITYPE encodes the type of instruction which is then used by IAB to assemble the decompressed instruction. The IAB combines the opcode and immediate bits to form an instruction. END is used to check the last instruction. On average, TPD is 1.7% of original program size.

Register Generation:

Register generation(RGEN) is a state machine that decodes the O_p field of the incoming compressed word into a sequence of operand registers needed by instructions in the tree-pattern. In RGEN, the number of states is determined by the largest tree-pattern in the program and the smaller tree-patterns are considered a s don't care. The average RGEN size of the uncompressed program is 4.9%.

Immediate Generation:

This module is used to store the immediate values. To minimize the use of bits to store different lengths of immediate values are used. Then these are store in memory banks in which the memory bank address and bank selection is done by IGEN from code word O_p. IGEN, TGEN and RGEN work simultaneously. Through this method, the average number of distinct immediate gets reduced and average compression ratio accounts to 2.2%.

Branch Target Address The branch instructions are compressed and are split into target address of 21 bits and offset of 5 bits. During the decompression of these instructions, the values address and offset are gathered from the IGEN and then the branch instructions are assembled.

Results:

The compression ratios are calculated using these two techniques by separating and encoding the patterns using combinations of fixed and Huffman code-words. At first 0%(50%) patterns were encoded using Huffman(fixed-length) and rest (50%)0% by fixed-length(Huffman). The point 0%,50% are combination of fixed-length and Huffman code-words with an escape bit to differentiate them.

The results of two encoding techniques are plotted by taking percentage of patterns on x-axis and compression ratios on y-axis. The above two images depict that, when more patterns are encoded using Huffman technique, compression ratio would be less. Use of VLC over Huffman causes a rapid increase in compression ratio. The average VLC compression for tree to operand patterns is 13.7%: 26.9%. By comparing the compression ratios of Huffman and VLC encoding techniques, the contribution of BH is 12% when the tree patterns are 5%. [3]

E. Using Instruction Recycling

Designing x86 decoders, to maintain backward compatibility with addition of new instructions would eventually: exhaust the range of unique instruction representations (opcode space), increase binaries and increase decoder complexity. This trade-off between extending microprocessor ISA efficiency and maintaining backward compatibility is referred as the ISA aging problem. This paper introduces SHRINK, an approach to address the harmful effects of ISA aging and reduce the ISA complexity without compromising backward code compatibility.

An instruction contains opcode field along with an extra prefix field, and sometimes a ModRlM field. The paper uses UNIQUE INSTRUCTION SIGNATURE (UIS), to denote the minimum combination of those fields necessary to identify an instruction. Over the years, average number of UIS bytes has grown from 2.7 to 4 bytes. Having fewer, shorter instructions, would simplify the processor's design, testing and validation, the effectiveness of the I-cache, and the final size of the binary executables. The total unused instructions on an average is 38% for all combinations of UIS's.

There are some radical approaches like: Reduce all UIS's to 2 bytes, reduce all UIS's to 1 or 2 bytes and Convert to a RISC-Iike ISA encoding, to reduce the ISA and Approach 2 of variable-length UIS's presents the best reduction in code size among them but SHRINK has more benefits.

Terminologies:

Processor revision (PR): An ISA version implemented in a CPU.

Software revision (SR): SR identifies the PR for which a program was compiled.

(If i=j then Software at SRi is fully compatible with a processor at PRj).

UIS Revision (UR): An array to keep track of number of instructions associated with a specific UIS over time.

Revision Vector (RV): It is defined as the set of URs of a processor.

Trap Mask (TM): It is defined as a vector produced by the element-wise < > operator between two RVs. TMx, y gives the list of all instructions that must be emulated when running SRx in PRy.

For revisions merging: The operator is defined as an element-wise OR between two revision vectors.

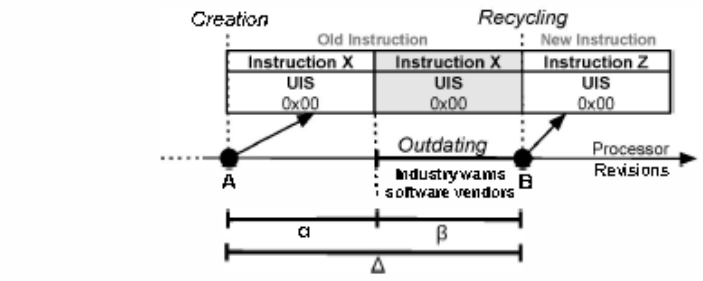

Figure: alpha is the time for adoption of new instructions, beta is the time to outdate instructions and delta is the minimum instruction lifetime. Hence Delta = alpha + beta [9].

In this paper Active Revision Vector (ARV) is calculated based on emulation technique using PR and SR emulation for at least a single instruction. ARV contains a TM which in turn decides which UIS's to emulate by deciding a trap vector. Using emulation, the processor can decide which version of the ISA to run from a variety of UIS's and find trap candidates. An Effective Revision Index (ERI) is an index to ARV which requires special bits explained in the hardware section.

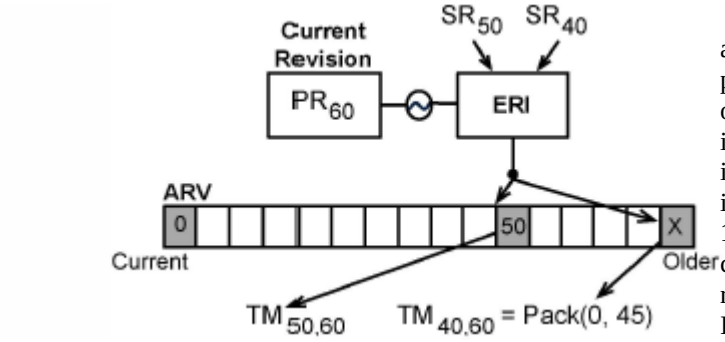

Figure: General trap mechanism using the ARV [9].

Modifications in Hardware/Software:

To implement SHRINK, ERI needs to be tracked as it decides which version of ISA to use. This can be achieved using Page table and by informing processor about additional ERI bits. Page table method is basically extending page table entries by 4 bit ERI which defines the versions (current and previous) the Page table, TLB and the Linker-Loader request mechanism. The processor functionality is increased to identify the instructions that require emulation and also issue a trap. The size of this extra circuitry is quite small if compared to the area saved in the decoder by removing instructions.

By extending Page Table, software needs to informed about the extension. Here the Linker and Loader use the ERI and SR values, track and load entries marked with them and then inform the processor. Decoding of the instructions is done by the OS and decides which instruction to outdate using trap value from the emulation.

The results showed that minimum DELTA with afforded overhead of 10% is 8 i.e. instructions created in last 8 years cannot be removed without incurring high overhead. There is reduction in critical path, area, and power consumption, respectively, by 23%, 48%, and 49%, on average when compared to Naïve decoders with an emulation overhead on legacy software of less than 5% when emulating up to 40% of the x86 ISA.

F. Code Optimisation using Instruction Register File

To determine the design requirements of a processor, a new architectural and compiler approach is developed by integrating an instruction register file (IRF) [7], in which frequent instructions are referenced in arbitrary combinations. The first structure is a IRF to hold common instructions and second involves immediate table containing most common immediate values used by instructions in IRF.

The ISA modifications to be made to support references to a 32-bit instruction register file are loosely and tightly packed instructions, using parameterized immediate values and positional registers. IRF can be integrated in two locations, at the end or at the start of instruction decode based on the methodology used to modify the ISA.

Loosely packed instructions:

A standard MIPS instruction is modified to contain an additional reference to an instruction from the IRF. In loosely packed format, the IRF instruction field is inserted in the place of infrequently used shamt field in R-type, and in I-type of instructions the length of immediate value field is reduced to include IRF instruction. No changes are done to J-type instructions. By doing this, immediate values are reduced from 16 to 11 bits and removing shamt from R-type, results in the change in the format of shift instruction by using the unused rs register when shifting by an immediate. When an individual IRF instruction is detected, neither of its neighboring instructions are available through the IRF. This redundancy cannot be captured by using loosely packed instruction format.

Tightly packed instructions:

This format has the capability to allow many RISA instructions to be accessed simultaneously from MISA. As IRF consists of 32 entries, a tightly packed instruction can consist up to 5 instructions from IRF. Tightly packed instruction can support fewer than 5 RISA by padding with nop references. When nop is encountered hardware stops execution, avoiding performance degradation.

Using Parameterized Immediate values:

By parameterizing the IRF entries those referred to immediate values, more instructions can be potentially packed. This requires additional encoded bits to fetch the necessary value. The immediate values are kept in immediate

table such that the packed instruction can refer to any parameter value.

Using positional registers:

Another method for increasing instruction redundancy is to remove the common patterns in register usage. The previously used registers can be referenced by current instruction through which RISA can have more flexibility as it can access both hardware and new registers.

Compiler modifications:

The GNU compiler allows use of pseudo instructions, which later are expanded by the assembler. To reduce this complexity, the pseudo instructions are expanded at compile time and are packed into the IRF. One of the limitation of IRF is space, so selected instructions used in IRF must provide great benefit to performance of program. The source program should be recompiled as packing is done by the compiler. It is done per basic block and packing across basic block boundaries is not allowed. The packing operates by examining a sliding window in each basic block in reverse order. This algorithm attempts to find each type of packing and finds the match. After packing is done for all the basic blocks, packing is again done for each instruction containing a branch or jump instructions that are not included in tightly packed instructions.

Several benchmarks from the MiBench suite are used to determine the benefits of using IRF with respect to code size, energy consumption and execution time the efficiency of packed instructions. Benchmarks such as blowfish and adpcm provide up to 2% and 1% reduction in code size.

On average, packing instructions reduces code size to 83.23% of its original size. Further it is reduced to 81.70% by parameterizing immediate values with IMM and incorporating positional registers reduces it to 81.09%. In case of energy, the consumption by I-Fetch due to IC misses is 30% - 45% approximately.

G. Code Compression using Instruction packing into Register Window

To obtain the instruction packing by placing the frequently occurring instructions into the Instruction Register File [5]. It proposes the extension of hardware and software to the IRF that supports multiple instruction register windows for allowing the large number of instructions to be packed in each function. The simple architectural enhancement is provided for reducing the instruction fetch cost.

Using Loop Cache

The general idea of loop cache mentions to keep the frequently used instruction into the register. This includes the Memory Instruction Set Architecture-MISA (Instruction referred from memory) and Register Instruction Set Architecture-RISA (Instruction referred from the register) instructions. In this approach the multiple RISA instruction are referenced into a single MISA instruction and they are known as the packed instruction.

Orthogonal Approach

This approach has been proposed and that specifies of placing the instruction into register. Thus packing of the most frequently executing instruction into an Instruction Register File is done by the compiler. This approach enables the single instruction to be fetched for the multiple instruction.

Using Code Compression

This approach proposes to reduce or compress the code in the program. This helps in decreasing the number of cache misses because of the less number of instruction being accessed during the execution.

Using Alternate Instruction Caching Strategies

This approach summarizes common sequences of code in the form of routines and each of this sequence is converted into the calls. In, this the summarized instruction sequence is meant to be overlapped and it eliminates the return of explicit instructions at the end.

Using Alternate Instruction Storage Strategy

This approach uses the Hardware dictionary wherein the duplicate sequences of the code are placed in the special control store in the processor and the various words of code(code-words) are associated with each of the sequences.

Using Dual Instruction Sets

This type of technique supports 16-bit and the 32-bit instruction format.

Using Zero Overhead Loop Buffer

In this approach the innermost loop is loaded explicitly and then executed. This type of loops is limited and so the limited number of instruction should fit in the buffer. Besides the Loop branch there can be no control transfers and the number of iterations that are executed must be known.

The following are the software windowing techniques to IRF that can reduce the fetch cost.

Using Greedy window approach:

Partitions are added and the most advantageous functions are selected to be either placed in the new partition or else they are placed in an old partition by merging with them.

Using Software window approach

Various benchmarked instructions are being packed in the software window along with the IRF and in some cases without IRF. Using instruction caching strategy and instruction storage strategy increases the execution time

because of the more transfer controls and this spoils the instruction cache performance due to diminishing spatial locality. Using an IRF with Loop cache reduces energy consumption than any other feature in isolation, it also reduces power consumption and incurs a lot more execution time penalty. Zero overhead loop buffers and the Loop cache helps in reducing the fetch cost significantly. Orthogonal approach not only saves energy but it also saves space by reducing the number of instructions that are stored in the memory. This improves IC hit rate and also improves the execution time. The software window approach obtains the average fetch cost by 54.04% hence it is not used with IRF. Unlike this, when it is used with IRF the fetch cost is 58.08%.

H. Code optimisation by reducing Instruction Cycle.

The novel ISA is designed that removes the gratuitous overheads and it helps in speeding up the performance of embedded DSP applications on resource constrained processors. This novel ISA also improves the energy efficiency by reducing the number of instructions. In comparison with the RISC processor, the simulation results that are obtained on the benchmark programs improves the performance too.

The cache-less dual-port tightly couple memory system with zero wait state and due to this the one read and one write operation can be performed within one single cycle. Using memory-ALU forwarding reduces the number of cycle by one. In this methodology, data is loaded from the memory and it is temporarily stored in separate register. Along with this the data is also written to the register file (RF) and that allows the instruction that is consumed to access the data without having it read from the register file. MAC operation is used along with the CISC and RISC version for the reduction of number of cycles. Also, different sizes of fast Fourier transform are used with the RCISC and RISC version for the reduction of number of cycles. The multiply accumulate CISC methodology (MAC-CISC) uses 4 cycles that is 2 cycles less than the RISC operation. This reduction of 2 cycles is because of the following reason:

(1) The memory locations of read and write are found to be same hence their effective address is not needed to be calculated

(2) The MAC instruction has no need to wait because the operator is directly loaded from the memory, then it is stored in a temporary scratch register which is then forwarded to the ALU. The RCISC implementation uses the minimum number of cycles that are needed to execute the load-mac-store sequence. The three cycles are used in RCISC implementation for executing complete MAC operation. This result shows that the number of instruction used by RCISC is half then the instructions used in RISC version and its 1 cycle less than the CISC version Using RCISC the performance has increased by 2 times. As the MAC-CISC operation uses less number of cycles than RISC but that reduces its energy efficiency of a processor to a greater extent. The RCISC is evaluated using the different sizes of the FFT function benchmark. The RISC version uses the 20 instructions and 23 cycles in total for performing a single radix-2 butterfly operation whereas the number of instructions and cycles used by RCISC for performing is 12 and 16 respectively, this shows almost the 40% and 30% reduction in number of instructions and cycles respectively. The performance of FFT functions was improved using RCISC by 3 times in comparison with the 3-stage low cost RISC processor. The RCISC processor achieves the speedup up to 3.5 times as compared to the RISC processor. The programs that accesses memory very less frequently gains performance by 2 times. The idle resources are turned off for longer time span or the speed of processor is lowered and this leads to the saving of energy.

I. Reducing instruction fetch energy in multi-issue processors

Gavin et al. [8] address the typical issue of processor power consumption. The growth of technologies allowed the microprocessor architects to deliver higher performance by increasing clock frequency which inevitably led to higher heat dissipation. However, the optimization which lied underneath this entire issue was to effectively perform operations with minimal power consumption. Their work presents us with various techniques to effectively reduce instruction cache power and techniques to reduce power consumption in Instruction Translation Look-aside Buffer (I-TLB), Branch Prediction Buffer (BPB) and Branch Target Buffer (BTB).

As the baseline for their study and work, Gavin et al. use the Tagless Hit Instruction Cache. They modify the underlying microarchitecture to support multiple instruction fetches in order to reduce the overall power consumption by altering the output port of the cache to allow adjacent instructions to pass through. A hit isn't guaranteed with the increase in the fetch width, that is the alteration of the output port and hence they suggest the cache line size to address the same issue. Furthermore, they experiment by disabling different buffers upon a cache miss for the instruction cache to estimate the overall performance of the system. For this evaluation they simulate the environment using Simple Scalar simulator and Wattch extensions and CACTI, using the MiBench Suite as the benchmark suite for their evaluations. The higher the leakage rate found in the tagless hit cache, the poorer was the performance.

From the same fact they conclude that lower sized tagless hit caches for instruction fetch are better at optimizing power consumption. With the same configuration they're able to avoid L1-IC access for 77% to 88% of instructions, for BPB and BTB the access is reduced for 77% and 83% of instructions in 1-wide fetch, 72% and 78% for 2-wide fetch and 64% and 68% for a 4-wide fetch. In the MiBench suite, they observe that only 14.2% of instructions are direct transfers of control (jumps and branches). In the 1-wide fetch configuration, they were able to come to within 9% of this

optimum for BPB and BTB target array accesses, and to within 3% of the optimum for BTB tag array accesses.

J. Transparent Instruction Set Customization

There is always an option to add custom instructions to a processor or design Application specific instruction processors (ASIP) to increase the performance but the time and cost of designing a new processor for each application is immense. An alternate for this is to integrate custom instructions into a general-purpose processor (GPP) and define a generalized framework for the same. This paper introduces, Transparent Instruction Set Customization on a GPP which enables many of the performance advantages of ASIPs without the associated overhead. The main idea is to identify subgraph at compile time and run time and feed it to the CCA subsystem to evaluate the subgraph using control registers.

Here, a configurable compute accelerator (CCA) is defined to execute selected computation subgraphs. The main idea proposed in this paper is to identify subgraphs or small section of code that can be computed once and can be used again without the overhead of processing but with the change in register values only (more applicable in looping situations).

The given code is analyzed in two ways: static method to identify the subgraphs on compile time and a dynamic method to identify and remap subgraphs to the CCA in a trace cache fill unit during run-time. This kind of hybrid approach, enables the combination of sophisticated offline subgraph detection algorithms at compile time with the flexibility of online realization of the customized instructions at runtime.

Additional hardware called the CCA subsystem consists of three major parts: the CCA, a configuration cache, and a control generator. This is added to the processor to customize it and hence help in both the static and dynamic subgraph analysis. The paper uses Branch and link instruction to explain the functioning and also provide means to identify subgraph to execute on the subgraph. Also used is BTAC, a branch prediction scheme that not only holds the PC for branch but also register numbers for the inputs to CCA instructions and an index into the CCA configuration.

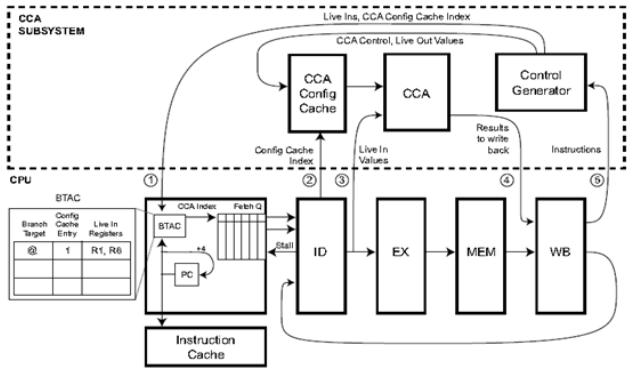

Figure: CCA SUBSYSTEM in addition to GPP CPU [10].

Transparent instruction set customization architectural framework:

An instruction when passes through the various stages of processing goes into both the CCA subsystem and the CPU simultaneously and hence both the CPU and CCA interact to process an instruction. At various stages control signals, index values, register values from the BTAC are exchanged between the two. The various steps begin from fetching the instruction and values, decoding the instruction using cache index value and the register values, provide values to the CCA subsystem, writing back to the CPU register values and then synthesizing CCA instructions using the control register from subgraphs. If at any point the subgraph cannot be mapped to the CCA index, the process is aborted and new subgraph is identified. The subgraph is evaluated in the CCA subsystem by separating logical and arithmetic instructions and placing them in the CCA structure at each level based on the code length.

Compiler Code Generation: Since we add new hardware, compiler should be modified accordingly. The various phases of compilation, are explained in the paper which flows in a vertical manner, passing through phases in a controlled manner. CCA compiler must determine which dataflow subgraphs should execute on the CCA.

The various steps are to use a branch and bound algorithm and subgraphs are enumerated within a basic block or superblock, integrate the subgraphs into a single instruction and make appropriate marking in the full code, Prepass/ Postpass Scheduling same as in a general compiler, subgraph expansion in order to allocate register values along with the non-subgraph register allocation, recompression of subgraph using a spill code and then passing the function onto the CCA subsystem to evaluate the subgraph and produce results using the cache index.

This method on evaluation led to average performance gains of 2.21x for domain specific CCA designs. There is a modest cost overhead beyond the original processor design. Also, there is an increase in the die area of the processor on the control generator by only 0.169mm and an additional delay of the control generator is only 0.46ns plus latch setup time. Since these values are effectively small enough, CCA subsystems can be latched onto various processors.

III. DISCUSSION

Code compression for processor optimization certainly has its benefits, the call to a dictionary after the compilation of the program not only requires us to have a post compilation analyzer but also a modified processor design which supports the compressed code. The question such a design imposes is about optimizing the code further with some other approach previously discussed in this document, the micro-operation optimization which works with the rePLay framework would require us to modify the processor design as well as the

framework data path to a certain extent. These modifications scale up too many components while applying some techniques to optimization, at the same time giving an indication of a not so modular design. Wherein, different modules can be added to the system or removed from the system. Another perspective to take a look at is the micro-operation optimization approach, this approach comprises the use of another framework and a modified datapath for the same putting one-too-many hardware primitives on top of the processor for improving its performance. It is without a doubt that the processor is optimized by 39% reduction of micro-operations but at the same time, it's not really desirable to have a processor with such hardware primitives. Also, processor execution time is traded off for performance using the dictionary, this makes it less likely for the call-dictionary algorithm to be applied to general purpose computing processors as the processor performance time cannot be compromised at all. At the same time both these approaches bring to light many redundancies in the compiler generated code as well as redundancies in the sub-operations which are used by the compiler to address the same issues. These two aspects provide us a background to think about instruction set architecture optimization in a different manner altogether, giving light to an aspect of the architecture which addresses these issues without having to modify the processor on such a large scale.

The advantage of code compression using bit width reduction is reduced code size and power consumption by 45%. The disadvantage is that not all instruction can be converted to the equivalent compressed instruction due to encoding restrictions and restriction in terms of register accessibility. Also the reduced power and die area come with an increased program execution time due to the maintenance of extra decoding logic for the ROA buffer.

Using old UIS's rather than adding new instructions helps SHRINK perform better when compared to Naïve decoders. It also allows to remove several instructions from the x86 ISA and improve the instruction decoder. Using Transparent Instruction Set Architecture increases performance of 2.21x from general purpose processors but there is a slight increase in die area and delay is introduced in the system. Since this increase and delay is marginal, it can be neglected.

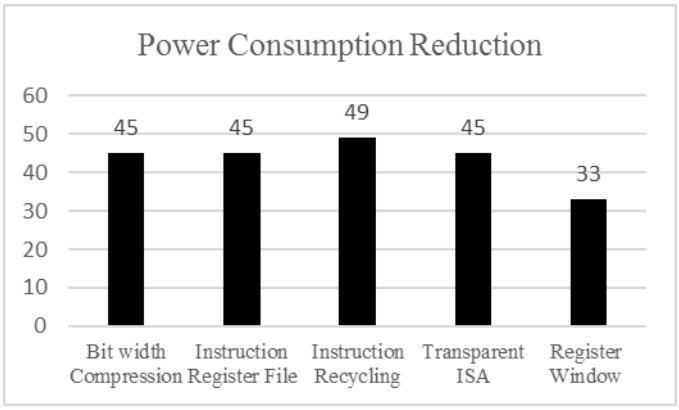

Figure: Comparison of Processor Power Consumption using various approach.

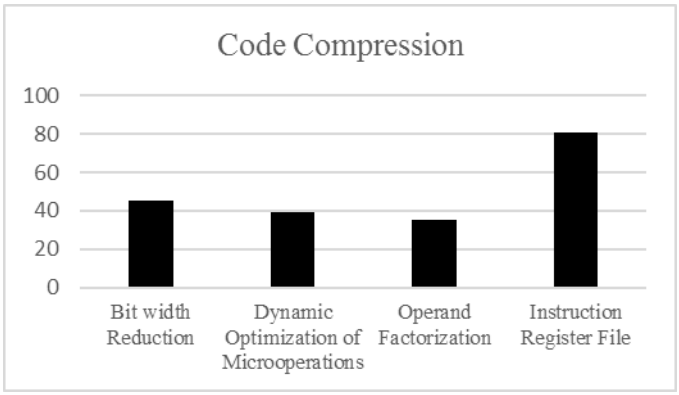

Figure: Comparison of Code of Compression using various approach.

IV. Future Scope

To achieve a larger set of window [7] than there are existing hardware windows. For some larger applications this increase in the window size helps in packing more instructions.The another area that can be included for future research includes the expansion of the evaluation of IRF [7] with the existing code compression technique.

The effectiveness of IRF [5] can be improvised by changing the architecture, by adding more compiler optimizations and also by adding both. Instructions can be loaded dynamically with varying phases into the IRF [5] at different points and then packing instructions into application more effectively.

Code Compression technique [3] can be improved by operand patterns that encodes the same temporary register by 2 times. This could become possible when the decompression engine is able to keep track of the temporary registers. The

another way by which the code compression could be improvised is by analyzing the co-relation between the operand patterns and the tree.

Using techniques like I-TLB, BPB and BTB in the hardware description language a novice processor can be constructed and using EDA design tool along with this for implementing a hardware design. By this the power consumption will be reduced.

The code compression dictionary algorithm [2] can be used with a better solution of the dictionary placement, the search and hash technique which might expand the overall algorithm. But it may or may not serve as a tool to increase the performance at all times, it'll be contingent upon the processor specifications.

As paper [9] proposes to change the software and hardware in a processor, this may cause security implications. Changing files that require higher privileges on machine, could harm the system in other ways. Thus for future purposes, security of higher privileged files can be looked into.

In the paper [10], the instructions are fed into a compiler and then processed parallel in the CCA subsystem, hence the instructions required by the CCA require additional cycle to transfer. In the future the subgraph instructions can directly be fed into the CCA substructure and all control the spill code analysis. This will lead to less delay for CCA processing.

Microarchitecture modifications not only allow a better micro-operation reduction [1] but also allow us to inspect the idea about monitoring the underlying processes which occur during the execution period. These observations can lead to some discoveries in the way hardware operates on the operations. Reduction of micro-operations [1] is one of the many inconsistencies about which we're aware of, the datapath for the rePLay framework can be modified to include specific type of instructions which utilize the underlying hardware specifically. Certainly this aspect would be processor dependent but it can deliver higher performance for the specific tasks such as matrix multiplication.

The research which can be carried out on efficient processor utilization may or may not be limited to the points which we were able to address and at the same time, these are some of the points which we were able to deduce from our study of such reputed works.

V. Conclusion

Given the fact about various approaches to processor optimization, the factors of code size, power consumption and the manner in which particular instructions are stored lead out to be areas which can be optimized to achieve an overall optimization of the processor itself along with their dependencies imposed on the processor, or in some cases on the underlying instruction set architecture. Certain techniques work on certain situations only, however an approach which might be able to expand the positive outcomes of the various methods discussed in this document might prove effective for most of the programming problems faced by microprocessor designers and programmers.

Acknowledgment

We would like to thank Dr. Hyeran Jeon, Computer Engineering Department, San Jose State University for her continuous guidance and support as well as her being the motivation for us to select this field for conducting a survey. Her timely and insightful inputs to our questions as well as to the paths which we were exploring have helped us in a very effective manner.

References

[1] Slechta, Brian, et al. "Dynamic optimization of micro-operations." *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*. IEEE, 2003.

[2] Lefurgy, Charles, et al. "Improving code density using compression techniques." *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*. IEEE, 1997.

[3] Araujo, Guido, et al. "Code compression based on operand factorization." *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society Press, 1998.

[4] Lee, Jongwon, et al. "Dynamic operands insertion for vliw architecture with a reduced bit-width instruction set." *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012.

[5] Hines, Stephen, Gary Tyson, and David Whalley. "Reducing Instruction Fetch Cost by Packing Instructions into RegisterWindows." *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2005.

[6] Lozano, Hanni, and Mabo Ito. "A Reduced Complexity Instruction Set architecture for low cost embedded processors." *High Performance Computing & Simulation (HPCS), 2015 International Conference on*. IEEE, 2015.

[7] Hines, Stephen, et al. "Improving program efficiency by packing instructions into registers." *ACM SIGARCH Computer Architecture News*. Vol. 33. No. 2. IEEE Computer Society, 2005.

[8] Gavin, Peter, David Whalley, and Magnus Själander. "Reducing instruction fetch energy in multi-issue processors." *ACM Transactions on Architecture and Code Optimization (TACO)* 10.4 (2013): 64.

[9] Cardoso Lopes, Bruno, et al. "SHRINK: reducing the ISA complexity via instruction recycling." *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 2015.

[10] Clark, Nathan, et al. "An architecture framework for transparent instruction set customization in embedded processors." *ACM SIGARCH Computer Architecture News*. Vol. 33. No. 2. IEEE Computer Society, 2005.