MICROCODE COMPRESSION FOR TIPI

by

Nadathur R. Satish and Pierre-Yves Droz

Memorandum No. UCB/ERL M05/29

29 August 2005

MICROCODE COMPRESSION FOR TIPI

by

Nadathur R. Satish and Pierre-Yves Droz

Memorandum No. UCB/ERL M05/29

29 August 2005

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Microcode Compression for TIPI

Nadathur R Satish and Pierre-Yves Droz
EECS Department, University of California Berkeley

ABSTRACT

A lot of applications need dedicated datapaths to be computed fast. While building the datapath is usually simple, designing and debugging the associated control logic can be a long process. The Mescal group is working on a development environment which automatically generates this logic. Traditionally, a register-transfer level vertical instruction set is decoded into pipelined control bits with forwarding and hazard detection logic. However, for novel, low-power, parallel, deeply pipelined, heterogeneous embedded processor architectures, such an ISA may not be easy to formulate, presents serious difficulties for compilation, and requires excessive dynamic control logic for forwarding and hazards. An alternate approach, which can be viewed as an extension to the VLIW style of processors is to use a control scheme that retains the full flexibility of the datapath. Thus, the resulting processor accepts a trace of horizontal microcode instructions that can be automatically generated. This eliminates the need to create an ISA (manually or otherwise), but requires a large number of bits for each 'instruction' in the microcode trace. Therefore, the microcode needs to be compressed after the compilation and decompressed on the fly. A high compression rate can be achieved using a trace cache to exploit the redundancy in the microcode.

1. Introduction

The background to our work is the Tipi micro-architecture design tool [1] being developed at the University of California, Berkeley. Their goal is to provide an architecture development system to specify, evaluate, and explore fully-programmable (micro-) architectures using a correct-by-construction method to ease the definition of customized instruction sets.

Traditional design flows define the instruction set architecture (ISA) first and then implement a corresponding micro-architecture. In this situation, the problem that arises is that the actual architecture must be checked against the original ISA specification to see whether it implements the ISA. In Tipi, The design flow in Tipi encourages the designer to think of the data-path micro-architecture first. He/she lays out the elements of the data path without connecting control ports. For unconnected control ports, operations are automatically extracted which are supported by the data path. These operations form the basis for more complex, multi-cycle instructions that are composed of several primitive operations. The construction of complex, multi-cycle instructions is supported by defining temporal and spatial constraints, i.e., automatically extracted operations from the data path can be combined in parallel and in sequence as long as resources for these operations do not conflict with each other, which is checked by Tipi.

The Tipi framework extracts the operations and produces a file with the horizontal microcode for each of these operations. It thus generates a horizontal microcode code-generator, which can take in an assembly file and produce the microcode equivalent of it.

The main problem with this approach is that the microcode produced is very large in size but is highly redundant i.e. the entropy is very low. This leads to problems of memory size and memory bandwidth required to read these microcode sequences from the memory. This leads to the idea of compressing the microcode in order to reduce these requirements on the memory.

It is of interest that the compressed stream need not be visible to the architecture designer since the designer will use the framework of the encoder-decoder pair which acts as a layer to abstract the generated microcode. Thus we are free to choose the encoding/decoding scheme.

The paper focuses on multimedia and signal processing benchmarks which have kernels with long pieces of linear code with few branches. The scheme proposed is not efficient on code that has a lot of branches.

The rest of the paper is organized as follows: In Section two, we present the encoding and decoding scheme that we use. In Section three, we describe the decoder architecture. In Section four, we describe the encoder architecture. In Section five, we present the results of our work. In Section six, we present our conclusions.

2. Coding/decoding scheme

The scheme that we choose is limited by the consideration that the decoder has to be implemented in hardware in order to avoid being on the critical path of processor execution. Thus software-based compression schemes like gzip cannot be used. Dictionary based compression schemes like Huffman exploit coding redundancy in the microcode but do not give sufficient performance to make storing microcode feasible.

We propose to use a L0 cache to exploit redundancy in the microcode. The cache will contain microcode traces. Since the horizontal microcode inherently controls a number of pipeline registers, and a number of multiplexers and register enables, it is reasonable to expect that parts of the microcode will have good locality, enabling us to obtain better compression for portions of the microcode. Thus we support the existence of multiple

trace caches, each of which contains a portion of the microcode.

The trace caches will be controlled by the decoder, which is in our case a processor that is specialized in trace cache operations. The motivation for this is described in Section three. The operations on the trace cache are of two main kinds: filling the trace cache with the microcode and reading the contents of the trace caches in the right order. The decoder has to be capable of performing these two functions. The decoder performs these two functions when it receives WRITE and SEQUENCE instructions respectively. The encoder is then a compiler that produces WRITE and SEQUENCE instructions. The WRITE instruction directs the decoder to fill in a particular trace cache line, and the SEQUENCE instruction provides the order in which such written values are to be read out.

The reading of the cache has to be done at the micro-architecture speed in order to avoid being on the critical path. This however implies that the decoder clock speed has to be much higher than the micro-architecture speed because it has to write the cache in addition to reading it. In order to keep the decoder clock speed low, we use a sequence manager for each cache. The sequence manager is autonomous of the cache writing mechanism, and is in charge of reading the cache in the correct order to regenerate the microcode trace.

Since each sequence has to be stored in a buffer that is of finite size, we can only write a limited number of microcode sequences before the reading begins. In order to further pipeline the reads and writes, it is essential that a mechanism be provided to continue storing the indices for the next sequence while the current sequence is being executed. If this is not done, the sequence buffer becomes a resource shared between cache reads and writes, eliminating the possibility of doing them in parallel. For this reason, each sequence manager has two sequence buffers which store the order in which the cache indices are to be read. One of the buffers stores the sequence in which the sequence manager is currently reading out data. The other buffer stores the indices for the next sequence. Whenever a sequence is complete, a START instruction is issued to the decoder in order to start up the sequence managers. At this point the new buffer is copied into the current buffer and execution starts.

Another point of interest in our scheme is its need to be applicable to various architectures. We require a scheme that is flexible for this purpose. We thus has a number of parameters like the number of caches and the line size of each size, total cache size, the size of the data bus, and other parameters that arise due to fact that the decoder can do multiple issues on each cycle.

The next section describes the architecture of the decoder and its implementation details.

3. Decoder

The decoder is a piece of hardware responsible for the decoding of the compressed microcode located in the program memory to a horizontal microcode that will be used to drive the architecture.

3.1 A processor and its ISA

The decoding scheme presented earlier requires doing various operations during the decoding: Writing a line of microcode in the trace cache, preparing a sequence of indexes that will be used to access the cache, output the microcode and starting using a previously prepared sequence. In addition, to control the execution of the code we need a jump instruction. We decided to implement the decoder as a processor specialized in these operations. It executes a binary code stored in the memory and complying with its ISA. The instructions in this ISA are listed in Table 3.1.

The number of parameters of these instructions is diverse; therefore their sizes are very different. Using a fixed instruction format would force us to add unnecessary zeros at the end of the shortest instructions, which would lower the encoding scheme compression

Table 3.1

NOP	No operation
WRITE C,I,D	Write data **D** in cache **D** at index **I**
SEQUENCE C,S	Prepare sequence **S** for cache **C**
START	Start executing all the ready sequences
JUMP A,O	Fetch next instructions from address **A** and offset **O**
SEQLENGTH SL	Use sequence length **SL** when executing the next sequences
STOP	Freeze the decoder on its actual state
COPY C,DI,SI,BC	Read data at index **SI** in cache **C**; Flip the bits **BC**; Write at index **DI** in same cache.

performance. Obviously, using variable length instructions would solve this issue, but with the cost of additional complexity in the fetch unit. A rapid evaluation of the gain in term of compression ratio shows us that the compression performance is divided by up to 4 using fixed-size instruction: using variable-length instructions is unavoidable.

3.2 Implementation

In order to avoid introducing stalls in the main architecture, the decoder must have finished preparing the next sequence before the current is finished. In most cases, this requires an IPC greater than one. The flow of instructions that will enter the decoder has two properties that will greatly help us to achieve this goal: there are no dependencies between instructions and we do not often execute branch instructions. With these assumptions, an architecture composed of several independent Fetch/Decode/Execute pipelines seems to be the most efficient way to achieve a high IPC.

The architecture we have chosen is presented in Figure 3.1

The decode unit of each pipeline has to be able to communicate to all the caches the operations they have to do. Thus, each pipeline has one bus on which the decode units can put the data to be written and the index at which it has to be written. It activates the cache that has to write the data through an enable signal.

The selection of the cache line that has to be used to generate the microcode at each cycle is done by a unit independent from the processor and attached to each cache: the sequence manager. When a sequence instruction is executed, it receives a sequence of indexes from the decode stage, and stores it to a temporary buffer. When a start instruction is executed, the content of this temporary buffer is copied to a working buffer, and the sequence manager starts to output the cache lines indicated in this sequence. The execution of the processor and of the sequence manager is synchronized by the start instructions: If a start is issued when the sequence managers have not finished executing the previous sequence, the processor stalls its fetch stage and waits until the sequence is finished. If a sequence manger has finished executing its previous sequence, but no start is pending, then it stalls the main architecture, and waits for a start to be issued.

Fetching an instruction on each cycle where the size is only known a-posteriori presents a performance issue: The size is actually calculated in the decode stage, or one cycle after the instruction has been fetched. The fetcher can not compute the address at which the next instruction is located in memory until it knows this size. This prevents the pipelines from fetching more than one instruction every two cycles. Several methods have been created to allow fast fetching of variable length instructions in normal processors, as [5] or [6], but these solutions are not really well suited in our case. As we do not have any dependencies between instructions, a more efficient method exists. The solution to this problem is interleaving two instruction streams in the same pipeline. On even cycles, the fetch unit takes instructions from stream 1 located at address 1 in the memory, and on odd cycles, it takes instructions from stream 2 located at address 2 different from address 1. With this interleaved scheme, an instruction is issued at every cycle. The task to split the instruction stream in two different streams is given to the encoder, which has to place the instructions at the right positions in the memory.

The datapath from the memory has a fixed size. This means that the fetch unit has to buffer the data coming from the memory in order to select the right bits that form a valid instruction. When a jump is decoded, its address is forwarded to the fetch unit, which flushes its buffer, changes its program counter to the new value, and starts making request to the memory to refill its buffer.

An issue specific to the decoder forces the encoder to care about the cycle at which an instruction is executed: If the decoder executes two writes and a start in the same cycle, one write can belong to the sequence located before the start and the other can belong to the sequence located after the start. This violates the correctness of the decoding: the start will be executed too early and the first write will be considered as a write in the second sequence. A piece of hardware could be added to deal with this case, but this would drastically increase the complexity of the decoding. We preferred to transfer this complexity to the encoder: it detects this special case and adds extra NOPs between the first write and the start to be sure that they will not be decoded in the same cycle.

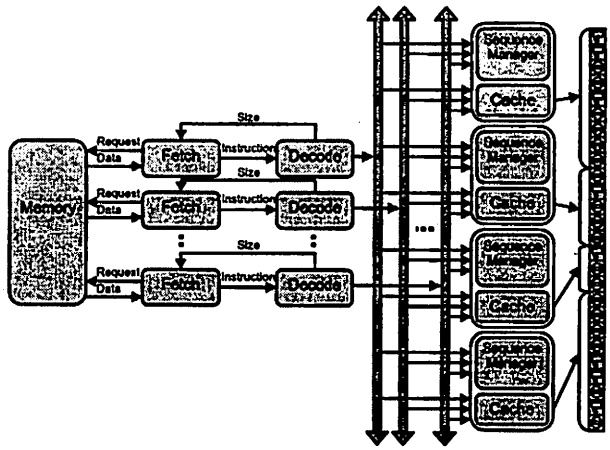

Figure 3.1 : Decoder architecture

4. Encoder

The encoder is a compiler that generates instructions for the decoder. The code generated by the encoder forms the compressed microcode stream that is stored in memory.

The encoder functions in two stages. The first stage involves generation of a linear instruction sequence and mapping the bits of the microcode that are "don't care's" in such a way as to minimize the code size. The second stage involves packing the linear instruction stream as per the requirements of the decoder.

4.1 Generation of instructions

The encoder performs the first stage by keeping a simulation of the trace caches. The encoder first performs a preprocessing step by taking in the input assembly code and the XML file giving the microcode representations of each instruction and substitutes them to obtain the input microcode. This microcode trace is then examined linearly. A single line of microcode is first broken up as per the length and number of trace caches. The number and length of trace caches are part of the parameters of the scheme that can be varied. After the microcode is split up, each trace cache is checked for a cache hit. If there is a cache miss, a WRITE instruction is generated. This implies either that one of the lines of the trace cache has to be empty, or that one of the lines has to be replaced. While replacement is done, care has to be taken that none of the trace cache lines involved in the current sequence being executed by the sequence manager is being overwritten. The WRITE instruction takes as operands the line so chosen and the microcode value to be written. In case of a cache hit, no WRITE instruction is generated. In either case, the cache index in which the value is either present or written into is added to a list which is issued to the sequence manager in charge of the respective cache. The list is passed to the manager in the form of a SEQUENCE instruction to the decoder. The next line of the microcode is then considered. As the number of bits that a SEQUENCE instruction can occupy is limited by the size of the sequence buffer, only a certain number of

indices can be added. Once this limit is crossed, the encoder tells the sequence managers to start their sequencing by issuing a START instruction, and the sequences in the simulation are emptied.

4.2 Mapping of don't cares

The original microcode has a lot of bits that are don't cares and which can be set to either one or zero. We decide which bit to use depending on which mapping will yield a cache hit. If none of the possible settings of the don't cares yields a cache hit, then we set all don't cares to zero before issuing the WRITE instruction. This is motivated by the fact that most bits in the microcode are zeros and we want the trace cache to also be similar, in order to have a better hit rate.

4.3 Insertion of NOPS and packing of instructions

Once the linear code is generated, it has to be split into a number of instruction streams. The motivation and scheme for interleaving instruction streams is described in Section three. Here we confine ourselves to how the encoder generates these streams. The special case of a START instruction being issued in the same cycle as a WRITE instruction for the previous stream is handled by padding the instruction stream with the requisite number of NOP instructions in order to ensure that the START instruction is always issued first.

After NOPs are inserted, the entire linear code is split into blocks of size ISSUEWIDTH. Each block is further subdivided into two interleaved streams. Alternate instructions are issued to the interleaving streams until all instructions in the block have been issued. Then the next block is taken up and the process repeated. This sequence of instructions generated is the packed sequence.

Since the decoder initially does not know at which point in memory each of the streams start, the first few instructions are JUMP instructions to the start of each stream.

4.4 The COPY instruction

The WRITE sequences that are issued form the major part of the total code size. An attempt made to reduce the number of WRITE instructions was to introduce a form of delta coding through a COPY instruction. The motivation behind this was that in many cases, the new microcode lines differed only slightly from an existing cache line with only a few bits different. By using a COPY instruction which only encoded the original index and the bits to be flipped, it was hoped to reduce the total code size. In order to optimize the use of such COPY instructions, such instructions are issued only when the corresponding WRITE instruction occupied more bits than the COPY instruction. The decision whether to use COPY or WRITE is made by the encoder. Since the encoder is meant to be in software, it typically can do expensive operations.

5. Experimental results

5.1 Evaluation methodology

We built a simulation environment to evaluate our scheme: The encoder has been implemented as a JAVA application, and the decoder as a C++ cycle-accurate simulation. Three test architectures have been used: RSA is a hardware implementation of the RSA coding/decoding algorithm, CC is a Convolution Coder processor, and DLX is a processor compliant with the DLX ISA. For the DLX example, we use five test applications, that cover a wide range of multimedia applications, all built out of C code, and taken from [2]: des_branch is an implementation of the DES cryptology algorithm, des_unrolled is the same implementation, but with the main loop (which contains 8 iterations) unrolled. Fft is an implementation of the FFT algorithm, gsm_encode and gsm_decode are the two main kernels of the voice codec for the GSM standard. Finally, Idct is an implementation of the inverse discrete cosine transform, used in the JPEG 2000 standard.

All these programs have been compiled to ASM using GCC 2.7.2 cross compiled for PISA. The TIPI microcode generator has been used to generate the microcode that will be used by the encoder.

5.2 Optimization of the codec parameters

Before using our codec, any user has to decide the parameters that will be used to generate the codec, such as the size of the caches, their number, the number of pipelines in the decoder, and the length of the sequence used to index the cache. In the next paragraph, we will study the variation of these parameters on the global performance of the codec.

One could expect that increasing the cache size would allow for a better compression rate by increasing the hit rate. This is true, but another effect has to be taken into account: each time that this size crosses a power of two, one extra bit is needed to write an index into the cache, which drastically increases the size of the sequence instruction. Figure 5.1 shows the result of the measure of compression ratio when the cache size varies. We can see the gaps at size=32 and size=64. For the DLX example, increasing the cache size over 64 does not increases the hit rate. The IDCT example has the worst compression rate, fact that we will find again in the other measures. This is mainly due to its small size (541 instructions), which does not allow to take full advantage of the cache.

As shown by Figure 5.2, increasing the sequence length has always a positive effect on the compression ratio: By having longer sequences, the total number of sequences in the packed file decreases. As each sequence write is associated with a START, we decrease the number of these instructions, and the bits associated with these are not written in the packed file.

To avoid stalling the main architecture, the decoder has to finish writing the data in the cache and the sequence in the sequence manager before the start is issued. For a short sequence length, this means that we have to guarantee a high IPC. The Figure 5.3 shows the result of a variation in the issue width (=IPC for us) on

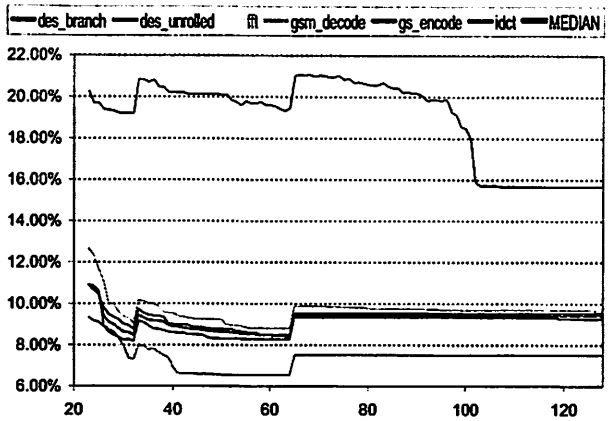

Figure 5.1: Compression ratio / cache size

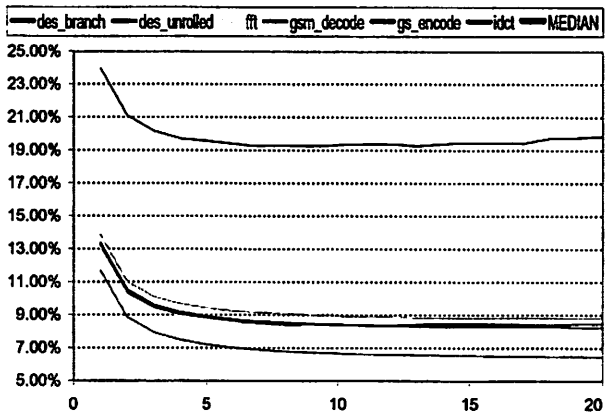

Figure 5.2: Compression ratio / sequence length

the number of stalls at the output for a sequence length of 7. An IPC of one is definitely too small for this sequence length, and up to 7% of stalls are introduced. However, above IPC=2, the decoder is fast enough to be able to decode the whole stream without any stall. The remaining 1% of stall is due to the stalls introduced during the initialization, when the decoder prepares the first sequence.

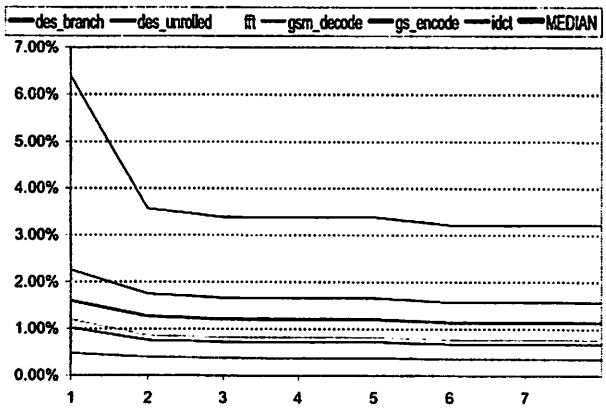

Figure 5.3: Stalls in the microcode / issue width

5.3 Comparison with other schemes

Figure 5.4 presents a comparison in terms of compression ratio between our codec (MT-codec, for Microcode Trace Coder Decoder), and other reference schemes: Huffman is a standard dictionary based method widely used to compress standard code. We use it to compress the microcode and we measure the size of the compressed file its produces. We have taken the source code for this application from [4]. DLX is a classic instruction decoder for a DLX architecture. The compression ratio for DLX is defined as the ratio of the size of the binary object file divided by the microcode size acquired using the Simplescalar [3] simulator. We also provide the compression ratio that gzip can achieve on the microcode. We consider that to be the final boundary that a compression scheme can reach (but gzip is not implementable in hardware). MT-codec does better than Huffman on any example. It is also nearly equivalent to the DLX instruction decoder, although the DLX ISA has been carefully designed to save room in memory, compared to MT-codec, which is an automatically generated coding scheme. MT-codec is not so far from the gzip limit, and can even do better in the des_branch example.

	RSA	CC	DLX / idct	DLX / des_branch	DLX / fft	DLX / gsm_decode	DLX / gsm_encode	DLX / des_unrolled	Avg
huffmann	165.33	30.14	39.21	34.50	32.18	34.23	34.44	32.71	34.55
DLX			11.05	10.29	10.04	10.17	10.13	10.35	10.34
MT-codec	58.22	8.17%	19.85	8.47%	8.38%	8.80%	8.24%	6.47%	10.03
gzip	85.33	10.68	17.53	9.69%	7.78%	6.35%	6.30%	4.01%	8.61%

Figure 5.4: Comparison with other schemes

Figure 5.5 presents the data rate generated by the instructions on the memory bus during the execution of the des_branch program. We used as a reference a DLX processor simulation (simplescalar tool suite), having a L1 instruction cache of size 64 lines of each 128 bits, for a total of 8KB. The replacement policy is LRU. In average, our scheme generates less traffic on the memory bus (7.2 bits/cycle) than the DLX processor, which generates a traffic of 8.3 bits/cycle. The main core of des_branch is a loop of relatively small size and can entirely fit in the instruction cache of the DLX processor, which explains that a big part of the execution does not generate any traffic. In comparison, our scheme permanently needs some SEQUENCE instructions, which keeps a residual traffic on the bus. The peak traffic is much lower with our scheme (20.7 bits/cycle versus 38.8 bits/cycle), which can be explained by the fact that when the caches are totally empty, or do not contain any interesting data, DLX has to

load the full instructions, whereas our decoder use delta-coding (COPY instruction) to modify the non-valid instructions that are in the cache.

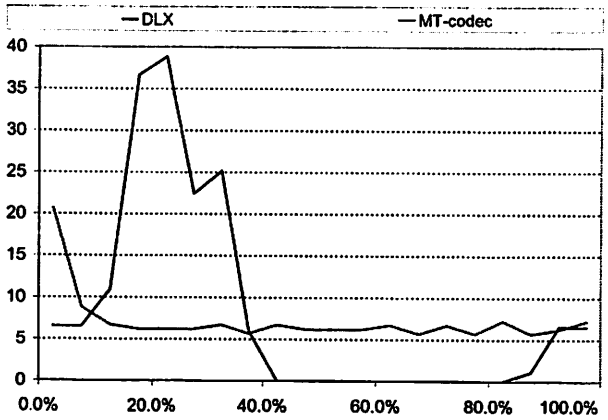

Figure 5.5: Memory data rate (bits/cycle) during an execution

5.4 Efficiency of the scheme

To see how we could still improve our scheme, we measured the contribution, in size, of each instruction in the packed file. On average, SEQUENCE represents 66% of the size, COPY 20%, and WRITE 12%. This shows that, for the kind of scheme that we have chose (trace cache operations), we are close to the optimum, which would be a packed file composed uniquely of SEQUENCE, and where the writing of data would not account for a big part of the file size. These figures also show that the COPY instruction is an efficient way to compress the WRITE instruction, as COPY accounts for 34% of the number of instructions, but only 20% of the size, while WRITE accounts for 5% of the instructions, but 12% of the size.

6. Conclusions

In this paper, we presented a scheme which allows for compression of microcode traces. In our scheme, the encoded stream is a set of instructions for a processor that operates on a set of trace caches and reads the values out in the right order. We presented details of the decoder and the encoder and compared the performance of our scheme with respect to the hand-coded ISA of the DLX processor, and to Huffman encoding of the original microcode.

The results show clearly that our scheme is competitive against the hand made encoding for the DLX. On average it performs slightly better. Our scheme does better in all cases than the Huffman encoding scheme, which does not fully exploit the redundancy of don't cares and does not do delta coding.

Our scheme also provides a way for the architecture designer to explore various parameters and choose the one best suited to the architecture.

Since in many cases the ISA encoding does not matter to the architecture designer, and in fact hand-coding involves significant effort and verification time, we believe an automatically generated encoding will be of great value. Our scheme provides an automatic way to produce an ISA which is the encoded stream.

Future Work

Our scheme does not provide any efficient branch mechanism. The JUMP mechanism is a slow mechanism used to initialize the state of the processor and to jump from one big linear block of code to another one. The decoder architecture could be used to provide a really efficient high granularity branch mechanism: on a branch, the processor starts filling the caches with the microcode needed for the two possible outcomes of the branch. The sequence managers receive the feedback information from the main architecture, and decide which sequence they have to execute. This would allow a fast branch with no extra cycle introduced by the decoding.

The COPY instruction provides an efficient way to compress the writes to the caches. After adding COPY, the SEQUENCE instructions represent the biggest part of the compressed file. Some similar instruction could be found which would be a compressed form of the SEQUENCE instruction.

For each architecture, the user still has to decide all the parameters for the encoding/decoding scheme. We provided in this paper results that can be used as a guideline to choose these parameters, but a more efficient way would be to generate automatically these parameters, using an analysis of the topology of the main architecture.

References

[1] Matthias Gries, Scott Weber and Christopher Brooks: *The Mescal Architecture Development System (Tipi) Tutorial.* Electronics Research Lab, University of California at Berkeley, UCB/ERL M03/40, October, 2003

[2] OpenMash.org: source code for vat-gsm, des

[3] SimpleScalar.org: PISA instruction set compiler and simulator

[4] DataCompression.info: Huffman source code

[5] Shai Rotem, Ken Stevens et al.: *RAPPID: An Asynchronous Instruction Length Decode.*

[6] Heidi Pan, Krste Asanović: *Heads and Tails: A Variable Length Instruction Format Supporting Parallel Fetch and Decode*

Acknowledgments

Thanks to Scott Weber and Matthew Moskewicz for their help in this project, and the time they have spent with us.

Thanks to Bongo Burger for the energy they provided us during many lunch breaks.