A 500MHz, 8-Stage Pipeline RISC Microprocessor Design with Sub-Computing

Ya-Lun Hu, Ming-Ku Chang, Fang-Yu Shiu, You-Hsiang Cheng, Tien-Fu Chen
Department of Computer Science and Information Engineering,
National Chung Cheng University
Chia-Yi, Taiwan (R.O.C)
hyl93@cs.ccu.edu.tw, cmk94@cs.ccu.edu.tw, sfy94@cs.ccu.edu.tw, cyh95@cs.ccu.edu.tw, chen@cs.ccu.edu.tw

ABSTRACT

In this paper, we have demonstrated RISC of embedded system design. Under performance and power consumption premise, Instruction Set design, like sub-computing instruction, load and store mask instruction, repeat instruction and delay branch instruction, has many characteristics in increasing instruction Powerful. Sub-Computing processes the simple operation first using a stage in the pipeline, and then using ALU executes the complex operation. It increases 20% performance in Instruction Set. There are 2 critical paths on Architecture. Improving them can increase frequency from 300MHz to 500MHz and pipeline stage will up to 8. Although increasing pipeline stage has the contradictory place with Instruction Set, we are trying to find out the balance point in the architecture. And we propose the best design way in the paper.

1: INTRODUCTION

As more and more complex applications are being ported to embedded system, speed of execution is becoming quite important. So we designed a more powerful computing way called sub-computing to improve the performance. It can achieve about 20% performance improvement. We have to meet the constraints of limited memory. Reducing code size and not sacrificing the execution time is our design objective. Therefore we design compression instructions which include 16-bit instructions and 8-bit instruction. They reduce code size effectively without adding any cycles to the execution time. It can achieve about 20% to 30% reductions of code size. Furthermore we devise the Load/Store mask for multimedia concerning. It can reduce significantly the movement of data. We also implement repeat instruction and delay branch to reduce the branch penalty. We can increase frequency from 300MHz to 500MHz by modify Architecture, and balance eight-stage pipeline between each stage. We revised partial Architecture to reduce power consumption and improve performance. In a word, we design a high performance and low code density instruction set. In addition, we develop an instruction level cycle accurate C simulator with inline debugger to evaluate our ISA design. It includes the ability of handling I/O.

2: RELATED WORK

2.1: Improve the performance

More recently, Clark et al. proposed configurable compute accelerators (CCA) [4] to speed up the execution of dataflow sub graphs, and an interface [3] that facilitates integrating different CCAs into the baseline processor. Figure 1 shows the diagram of the depth 7 CCA. Sub graphs are identified and replaced with new microcode instructions at run time. These microcode instructions control the CCA, which is a big array of function units that generally can capture sub graphs of depth from 4 to 7. The microcode is then invoked when a reserved instruction is executed in the program code. The compiler is responsible for what code sequences should be mapped to CCA sub graphs and at what locations in the program these sub graphs should be loaded into a CCA configuration cache. The other approach exploited the dataflow sub graph from sequences of dependent instructions to build up Functions expressing these sequences offline using replay framework [5] [9] is proposed to improve performance [10]. Figure 2 shows the implementation of functions using a set of 64 1-bit chained operators.

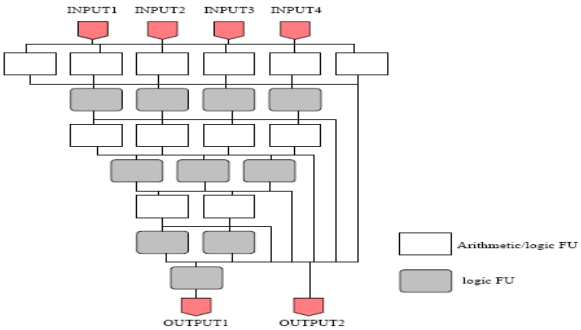

Figure 1: Block Diagram of the Depth 7 CCA

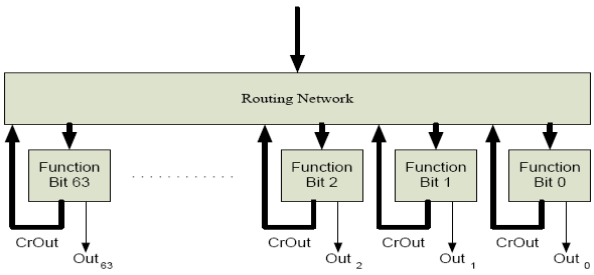

Figure 2: Implementation of functions

2.2: Improve the code density

Although a RISC instruction set is easy to decode, its fixed-length instruction formats are wasteful of program memory. Dual-width ISA designs, such as Thumb [1], Thumb2 [2], and MIPS16 [8], have been proposed to reduce code size to address the issue of limited memory bandwidth in embedded media computing. For example, Thumb and MIPS16 are defined as subsets of the ARM and MIPS-III architectures.

A wide range of applications were analyzed to determine the composition of the subsets. The instructions included in the subsets are either frequently used, do not require a full 32-bit, or are important to the compiler for generating small object code. The original 32-bit wide instructions have been re-encoded to be 16-bits wide. Thumb and MIPS16 are reported to achieve code reductions of 30% and 40%. Thumb and MIPS 16 instructions have a one-to-one correspondence to instructions in the base architectures. In each case, a 16-bit instruction is fetched from the instruction memory, decoded to the equivalent 32-bit wide instruction, and passed to the base processor core for execution.

Reducing code size also can be achieved by compiler techniques with hardware support such as echo instructions. The idea behind the echo instruction is to compress these repeating sequences of instructions by ”echoing” existing code sequences in the program [7]. If two sections of code are found to be the same, there is no reason to include both. The single copy is left in the location of one of the original sections of the code. All the other sections are replaced with a single echo instruction that tells the processor to execute a subset of the instructions from the single copy.

3: Instruction Set Architecture Design

3.1: Sub-computing

The traditional operations such as addition alone and subtraction alone can not meet the demand of complex application. So we propose a new approach to improve the performance called sub-computing. Sub-computing means it can do extra arithmetic operation (shifter, and, or etc.) to either the source operand input to ALU or the destination operand output from ALU. Figure 3 shows the diagram of sub-computing.

So we can perform two operations or three operations at most in one instruction. Such as “a = b + (c << d)” and “a = (b + c) - d” all can be done in a single instruction. Not only can the cycle counts be reduced, but the code size also can. Furthermore, sub-computing can be done on the source operand as well as the destination operand at one time. In this case a sub-computing instruction can achieve three arithmetic operations.

By the way of carrying the sub-computing information, it can be classified into two groups. One is carrying by the specific register, and the other is encoding in the instruction itself. These two types of instruction are different in its encoding.

In type one sub-computing, it carries sub-computing information in specific register, so requiring one additional instruction to put the sub-computing information into the specific register. There is a simple example for that.

r5 = (r3 + (r4 * 8)) / 2

The c code above-mentioned can translate into the following assembly code.

```
movimc .r0   r31,          0x341
add    .r0   r5,ashr #1,   r3,      r4,shl#3
```

The first instruction is used to put the sub-computing information into the specific register. Each sub-computing operation is described by 8-bits, 3-bits for function and 5-bits for operation amount. All of the information is occupied 16 bits totally in a register. Figure 4 shows the details of the specific register. Table 1 shows the eight kinds of functions which sub-computing unit can do. And then the second instruction performs the sub-computing operation by reading the specific register value. In the traditional arithmetic operation it uses three instructions and takes three cycles to complete the job, but in our approach it only uses two instructions and takes two cycles to complete the job. It gains the 1.5 times speedup. If the job (r? = (r? + (r? * 8)) / 2) is done for many times, the speedup we gained is more than 1.5 times. Because setting specific register needs to be executed only one time and it can be reused in the further sub-computing. In other words, if the job is done for four times, we can gain the 2.4 times speedup (4*3/ (2+1+1+1)).

In type two sub-computing, sub-computing information is encoded in instruction itself, one additional instruction is unnecessary. In order to encode the sub-computing information in the instruction, it sacrifices an operand filed. Compare to type one sub-computing, this is a two operand instruction. That is, one of the source operand shares the same operand filed with the destination operand. The other limitation is that only one operand can perform sub-computing at one time, either source2 operand or destination operand.

There is another simple example for that.

r5 = r5- (r4 + r3)

The c code above-mentioned can translate into the following assembly code.

```
sub .r0   r5,   r5,   r4,   add   r3
```

In the traditional arithmetic operation it uses two instructions and takes two cycles to complete the job, but in our approach it only uses one instruction and takes one cycle to complete the job. It gains the two times speedup.

Table 1: Improvement of Sub-computing

	ARM	RISC	Improvement
Code Size(bytes)	56	40	Reduction = 0.286
Cycle Counts	14	10	Speedup = 1.4

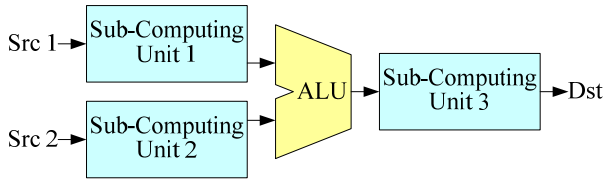

Figure 3: Diagram of Sub-computing

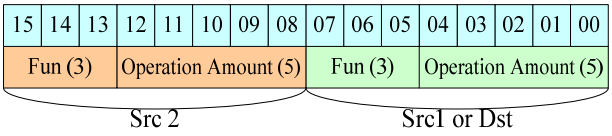

Figure 4: The Details of the Specific Register

3.2: Compression Instruction

Our RISC is designed to accept different width ISA executed concurrently; we choose the frequently used instruction to make up a short ISA to reduce code size. The 16-bit short instruction provides a subset of the functionality provided by the 32-bit instruction set. Nevertheless, the 16-bit code expends lesser energy and has a smaller memory footprint. Unlike 16-bit Thumb instruction of ARM, our short instruction can be executed interlacing with the 32-bit instructions. There are two benefits about that. First, the special branch instructions to change between 32-bit and 16-bit mode are unnecessary. Second, the 16-bit instructions can only capture part of expressibility and functionality of the 32-bit instruction. In some case two 16 bit instructions or even three are required to perform the same task that can be accomplished by a single 32-bit instruction. If the 32-bit instructions can be executed interlacing with the 16-bit instructions, there would be no increase in instruction counts. For this reason the compression instruction we proposed would not lead to the increase of executing time as Thumb instructions of ARM.

In order to reduce code size more and more, we propose an 8-bit instruction set called extremely short instructions. It makes use of the destination register of the previous instruction instead of encoding it in the instruction space. Here is a simple example of the extremely short instruction.

```
Ldw     .r0     r5,     *+r2[#4]
add     .rx     p1,     #7
```

The first instruction is a 32-bit instruction that loads the data from the specific address of memory to the 5th register. The second instruction is an extremely short instruction that add the immediate value to the destination register of the previous instruction, i.e. r5 = r5 + 7. In this example it achieves the 37.5% reduction in code size.

3.3: Load and Store Mask

Efficient load and store instructions are important. In some multimedia application, SIMD operation is useful. In some case we need to combine two half word data into a 32-bit register. With the traditional load and store, we need consecutive load operations, shift the data, and then do logic OR to complete the task. It's tolerable if the operations appear infrequently, but it gets worse when the function block containing this kind of operation mainly, such as DCT. So load and store mask is proposed for solving these problems.

As Figure 5, the data requires four instructions to complete in ARM. It is a tedious job and it needs a temporal register to hold one of the half word data. In a word, it is inefficient. Figure 5 is the example of load mask we proposed. The data requires two simple load mask instruction to accomplish.

```
ARM
Ldrh    dst1,   *mem[address1]
Ldrh    dst2,   *mem[address2]
Mov     dst2,   dst2,           lsl#16
Orr     dst1,   dst1,           dst2
RISC
Ldhw.L  .r0     dst,            *mem[address1]
Ldhw.H  .r0     dst,            *mem[address2]
```

Figure 5: Comparison of the Data Movement in ARM and RISC

3.4: Repeat

For many applications, a large percentage of the dynamic program execution time is spent in small program loops. These loop execution incur significant overhead due to the updates of loop counters and the branches to initiate a new iteration. Hence the other method is to provide zero-overhead looping by repeat a block of code to improve loop execution time. In our approach we introduce three registers, repeat counter register (RC), repeat-block start address register and repeat-block end address register to repeat a block of instructions.

Zero-overhead looping is achieved by comparing the PC value with repeat-block end address. If the value is equal and repeat counter is greater than or equal to zero, repeat-block start address is loaded into the PC to restart the loop.

Figure 6 illustrates the usage of repeat instruction. It repeats the block with the six instructions eight times.

3.4: Delay Branch

The branch penalty is a side-effect of pipelined architectures. When the branch is resolved the succeeding instructions is already fetched and decoded in the pipeline. If branch is not taken, the succeeding instructions continue executed. Otherwise the succeeding instructions need to be flushed. Consequently it causes the branch penalty. To compensate the branch penalty we implement the delay branch in our design. A delay branch always executes the following instructions which are independent to the result of the branch instruction.

Figure 7 and Figure 8 show the difference of ordinary branch and delay branch. We can observe that the throughput of ordinary branch is only two instructions, but of delay branch it raise to six

instructions. Hence the performance will be improved by replacing ordinary branch with delay branch.

```
Matrix Multiplication
  for ( i=0; i<8; i++ )
     for ( j=0; j<8; j++ )
        for ( k=0; k<8; k++ )
           c[i][j] += a[i][k] * b[k][j];
assembly
RPT   .r0   #8      #6
ldw   .r0   r3,     *r0[#256]
ldw   .r0   r2,     *r12[#512]
mp    .r0   r1,     r2,     r3
add   .r0   r4,     r4,     r1
add   .r0   r0,     r0,     #4
add   .r0   r12,    r12,    #32
```

Figure 6: An Example of Repeat Instructions

```
B       .r0   Label_1
Add     .r0   r1,   r2,   r3
Sub     .r0   r1,   r1,   r4
Xor     .r0   r5,   r6,   r7
Mp      .r0   r2,   r1,   r5
Label_1:
Mov     .r0   r3,   #0
```

PC gen	fetch	Dispatch	decode	Operand fetch	Ex1	Ex2	WB		
B	B	B	B	B					
	Add	Add	Add	Add	flush				
		Sub	Sub	Sub	flush	flush			
			Xor	Xor	flush	flush	flush		
				Mp	flush	flush	flush	flush	
					Mov	Mov	Mov	Mov	Mov

Figure 7: An Example of ordinary branch

```
BD      .r0   Label_1
Add     .r0   r1,   r2,   r3
Sub     .r0   r1,   r1,   r4
Xor     .r0   r5,   r6,   r7
Mp      .r0   r2,   r1,   r5
                          Delay Slot
Label_1:
Mov     .r0   r3,   #0
```

PC gen	fetch	Dispatch	decode	Operand fetch	Ex1	Ex2	WB		
BD	BD	BD	BD	BD					
	Add	Add	Add	Add	Add				
		Sub	Sub	Sub	Sub	Sub			
			Xor	Xor	Xor	Xor	Xor		
				Mp	Mp	Mp	Mp	Mp	
					Mov	Mov	Mov	Mov	Mov

Figure 8: An Example of delay branch

4: ARCHITECTURE

For the target of a 500 MHz processor, so we design the RISC architecture which consists of the 8 stages pipeline (PC Generation, Instruction Fetch, Instruction Queue, Instruction Decode, Operand Fetch, Execution1, Execution2, Write Back).Figure 9 shows 8 pipeline stages architecture. It avoided complex fetch instruction in Instruction Cache which temporal store instruction use the queue of 128-bit from the third stages.

Instruction Decoder can decode long, short, and external instructions in the same time. Operand Fetch includes both reading and dispatching data in the register file.

It can increase parallels from 5 function unit of Execution1 stage running in the same time. Its main tasks of Execution 2 are ALU execution, Multiplier process, and Data Cache storage.

Table 2: Critical Path Timing Analysis

Component	Timing(ns)
ALU & Sub computing	2.85
Decoder & Operand Fetch	2.6

The 6 stages of initial design have higher IPC, but it has the longer critical path and the unbalance of pipeline cut. Table 3 introduces the two critical path latency out of 1.3ns a lot a lot, but most latency of the component are under 1.3ns.The frequency to upgrade and balance pipeline cut, so we divided stage the ALU and Sub Computing into 2 stages. It adjusts two stages in original Decode and Operand Fetch. Although above adjust can tend to descend IPC, it is worth increasing frequency. The margin upgrades 60%, and the most latency is not more than 2 ns in original design architecture, the frequency is next to 300 MHz, to goes to 500MHz after above adjustment.

Another reason of adjusting pipeline is that the instruction designs use Sub Computing operation to increase operation performance. Instructions can assign Source/Destination to do Sub Computing operation. If the sub Computing was interacted in ALU unit that all components could extensive and loss flexibility.

On the other hand, if we assign individual Sub Computing Unit, then we'll get accord with need and higher flexibility. More instructions are higher used the Source section more than Destination section in the Sub Computing, so Sub Computing of Destination section and Writing Back in the same stage.

It could decrease IPC that increase one Stall after the instruction when Sub Computing in Destination section, but that instruction are less and IPC impact less.

Another reason of Load/Store Multiple instruction that waiting for complete Load/Store Multiple stall pipeline in a while and other instructions could not process.

We operate Load/Store instruction a lot by separate from Load/Store Path and ALU path. It improves to increase IPC that the Data Processing instructions of no hazard into ALU early.

It is an important of all performance that instruction queue quickly than cache speed to processor. If it fetches a lot of data and temporal store, and it is necessary to decrease Cache access overhead. The primary method of every fetch come from data of one cache line (128 bits), and store in instruction queue. It depended on the instruction length (8/16/32 bits) sent instruction every time.

It would fetch the next data of the cache line when there is less than or equal to 64 bits of data in queue to avoid every instruction fetch from cache, unless branch instructions.

It has decreased impact to frequency of cache access that we have design the compress instruction in instructions set.

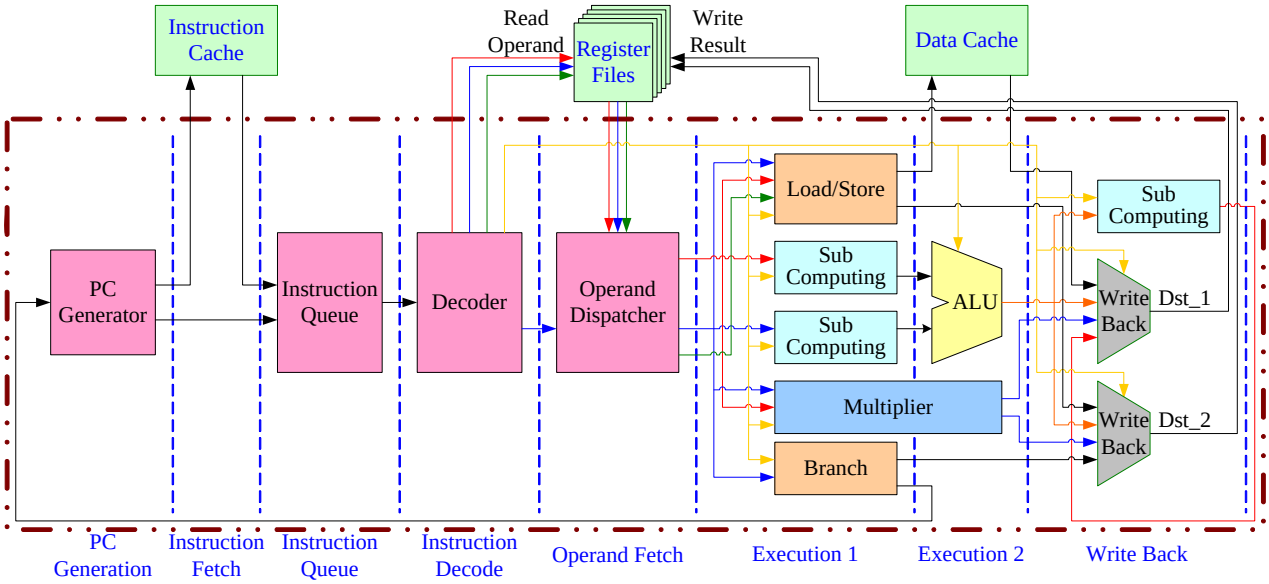

Figure 9: Architecture Block Diagram

Above, we see the same effectiveness of instruction queue. The other advantage is the code size decreases. It makes upgrading performance and limited of the complex that the same program in smaller memory and more program of memory could exist in the same time.

5: Experiment Environment

We evaluated the ISA across a wide range of embedded applications contained in the MiBench benchmark suite [9]. MiBench consists six categories including: Automotive, Network, Security, Consumer, Office, and Telecommunications. These categories offer different program characteristics and let us review our design. MiBench consists of twenty-three different programs totally. Because the RISC compiler is still progressing now, some programs are not executed correctly. So we only can show some result of Mibench.

We compiler these program to ELF (Executable and Linkable Format) file and simulate the program on an instruction level cycle accurate C simulator. And we compare the simulation result to ARM7TDMI, ARM940T and ARM1022E. Table 3 summarizes cache size.

Table 3: the information of cache size

Cache size	Instruction cache	Data cache
ARM7TDMI	none	none
ARM940T	4KB	4KB
ARM1022E	16KB	16KB
RISC	8KB	8KB

6: Experiment Result

Figure 10 shows the cycle counts of each processor. RISC is better than ARM940T in four programs. Furthermore RISC is better than ARM1022E in three programs. And it can get much superior if the compiler can fully support the all feature of RISC.

Figure 11 shows the instruction counts of each processor. We did not compare to ARM because they are executed on different environment.

When analysis architecture, measure Static Timing of component by synthesis. There are 2 critical path be found, that is ALU & Sub-Computing (2.7ns) and Decode & Operand Fetch (2.5ns). Divide ALU & Sub-Computing to ALU (1.65ns) and Sub-Computing (1.2ns), also divide Decode & Operand Fetch to Decode (1.1ns) and Operand Fetch (1.5ns). Table 4 summzrizes Timing Analysis.

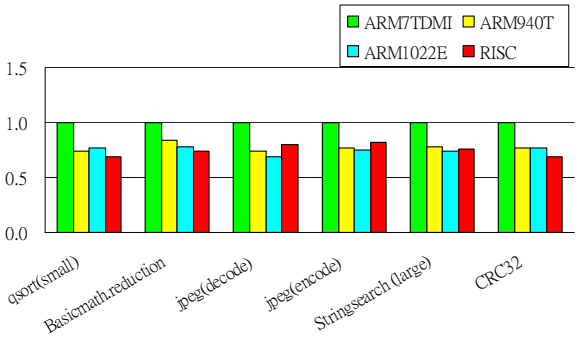

Figure 10: Normalized Cycle Counts Result

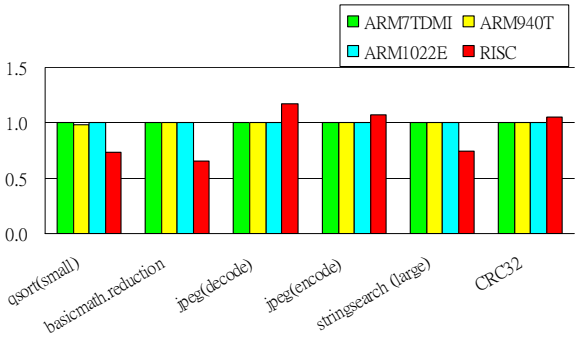

Figure 11: Normalized Instruction Counts Result

Table 4: Divided Component of Critical Path

Component	Sub Component	Latency (ns)
ALU & Sub-Computing	ALU	1.65
	Sub-Computing	1.2
Decode & Operand Fetch	Decode	1.1
	Operand Fetch	1.5

7: CONCLUSION

In this paper, we proposed an instruction set architecture of advanced RISC. It is a general purpose embedded processor. In our design it can achieve about 20% performance improvement and about 20% to 30% reductions of code size in hand-coding program. The compiled program can't gain as much improvement as handcoding program because still some features of instruction set architecture we proposed are not supported by compiler. The situation can be improved when compiler fully support.

REFERENCE

[1] ARM Limited. ARM7TDMI (Rev. 4) technical Manual, 2001.

[2] ARM Limited. ARM Thumb-2 Core Technology, 2003.

[3] Nathan Clark, Jason Blome, Michael Chu, Scott Mahlke, Stuart Biles, and Krisztian Flautner. An architecture framework for transparent instruction set customization in embedded processors. In ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture, pages 272–283, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, pages 30–40, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Brian Fahs, Satarupa Bose, Matthew Crum, Brian Slechta, Francesco Spadini, Tony Tung, Sanjay J. Patel, and Steven S. Lumetta. Performance characterization of a hardware mechanism for dynamic optimization. In MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture, pages 16–27, Washington, DC, USA, 2001. IEEE Computer Society.

[6] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[7] Jeremy Lau, Stefan Schoenmackers, Timothy Sherwood, and Brad Calder. Reducing code size with echo instructions. In CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems, pages 84–94, New York, NY, USA, 2003. ACM Press.

[8] MIPS Technologies. MIPS32 Architecture for Programmers Vol. IV-a: The MIPS16 Application Specific Extension to the MIPS32 Architecture, 2001.

[9] Sanjay J. Patel and Steven S. Lumetta. replay: A hardware framework for dynamic optimization. IEEE Trans. Comput., 50(6):590–608, 2001.

[10] Sami Yehia and Olivier Temam. From sequences of dependent instructions to functions: An approach for improving performance without ilp or speculation. In ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture, page 238, Washington, DC, USA, 2004. IEEE Computer Society.