Static Classification for Dynamic Decisions Using Assembler Instrumentation

Sylvain Aguirre
University of Applied Science
EIVD
1400 Yverdon, Switzerland
sylvain.aguirre@eivd.ch

Harsh Metha
Indian Institute of Technology
Kanpur - 208 016 (UP)
Kanpur, India
hmetha@iitk.ac.in

Daniel Mlynek
Swiss Federal Institute of
Technology
EPFL - station 11
Lausanne, Switzerland
daniel.mlynek@epfl.ch

ABSTRACT

Being able to predict or anticipate instructions that will be executed can help increase processor performance. For instance, static or dynamic branch prediction techniques—which respectively have limited efficiency and require silicon area—can be used to reach this goal. The principle of our approach is based on the sharing of tasks both at the compilation time by the assembler and dynamically at the hardware level. The main idea is to extract, at the compilation time from the assembly source code, relevant information from a hardware viewpoint and transmit it to the hardware through common but tuned Instruction Set Architectures (ISAs) for various purposes, such as selecting the more suitable branch predictor, saving silicon area or even reducing critical timing paths identified by the microarchitects. This approach implies a symbiosis between the assembler for the encoding of the information within the instruction and the hardware which needs appropriate logic to take advantage of that information. The first step must be carried out at the architecture or microarchitecture level, in the sense that the processor designer must identify the information he would need to improve the performance of a critical part of his design or to reduce a penalty. Then, the main step consists in identifying free bits within interesting instructions or reorganizing the instruction set in such a way that bits can be released in the considered instructions to introduce the information that the processor needs. An application dealing with the instruction flow has been studied using the GNU C Compiler (`GCC`) tool chain targeted for the MIPS 1 ISA. Our simulations using a MIPS R3000 virtual processor and EEMBC's integer benchmarks show that a speedup of up to 7% can be obtained for the chosen application.

Categories and Subject Descriptors

C.5.3 [**Computer Systems Organization**]: Computer System Implementation—*microprocessors*

General Terms

Design, Experimentation, Performance

1. INTRODUCTION

Generally,there are three ways to improve the performance of a processor for a given ISA. One can tackle the CPU frequency, the number of instructions performed or the `stall` cycles [9]. The number of instructions depends on the ISA and the compiler used—that is to say it is a pure software deal—whereas the clock cycle is a pure hardware affair. `Stall` cycles, on the other hand, can be handled either by the compiler tool chain or by the processor itself: in the former case, delay slots can be used and for the latter case, an interlock system is generally implemented.
What preceeds illustrates the fact that, even if ISAs have been designed to link in a smart way processors and programs which are foreseen to run on, those three factors are treated separately. Pushing all we can on the compilation time can help reduce the chip size, complexity, power consumption and increase the reliability of processor without forgetting that it is much less expensive to provide software patches than to fix hardware bugs.On the other hand, we know that some operations can only be performed at the run time and consequently managed by the hardware. Nevertheless, some hardware tasks, or let us say, some kind of insight could be shared with the software as we will see in section 5. This requires from the designer to have both a deep knowledge of the compilation tool chain and of the pipeline, in order to identify what can be done by the software chain instead of the processor or what kind of information can be useful to the processor.

Section 2 introduces works done in the static domain, whereas section 3 exposes how to integrate some extra information within an instruction using the GAS assembler. Section 4 describes the methodology followed to obtain results presented in section 5 for a chosen embedded application. Finally, section 6 concludes this paper in suggesting some applications that can take advantage of our approach and proposing to spread this work to other architectures.

2. BACKGROUND AND PRIOR ART

How to decrease the complexity of microprocessors while maintaing high performance has emerged as a big challenge, particularly in case of embedded processors. One way to deal with this problem is to shift some of the hardware based

operations to the software side.

Recent work on the static classification of instructions [18] illustrates an example of this technique. This work has proposed a method to move the hardware operation of choosing the correct branch predictor from a group of available branch predictors into software. The compiler statically classifies the instruction into various groups depending upon the predictor they should use at run time. This is done by both profiling as well as checking dependencies between the instructionsh. The compiler then adds extra bits containing this information to the instruction field. During runtime these extra bits are decoded by the hardware and are used to select the appropriate predictor for that instruction. Thus it reduces the hardware complexity by shifting the decision operation previously done by the hardware to the software. The proposed hybrid method in our work is another compiler-directed classification scheme based on a repartitionning of extra tasks between the hardware and the software by using the already existing ISA and its instruction format. Instead of adding extra bits to the instruction we try to exploit the fact that not all the combinations of bits are used by standard architectures. For example, in the MIPS-I architecture not all 32 bits are used for the `branch` instructions. Indeed, when a branch opcode is defined as a `REGIMM`[1] class of instructions, four bits in the new referenced 32 bit instruction field are unused. Our scheme targets such cases. These bits are then used to send information such as value predictibility pattern mentioned in [18].

3. STATIC CLASSIFICATION

3.1 Principles

We propose to supply extra information to hardware from software by using unused bits of ISA's instructions. Consequently, no extra bits are added to an instruction and the decoding stage is kept unchanged. Thus we are able to provide information to the hardware without having to increase the bandwidth as opposed to the approach proposed in [18]. Futhermore, this process ensures that no other information contained by the instruction is modified.

Once the instructions that can transport extra information have been determined, the user must first either create an independent software tool or hack the assembler to locate these instructions at compilation time within the assembly file, and then set the bits that are not yet used according to his convenience.

The next two paragraphs give more details of the two different implementations which, somehow, are made up of two tasks: identification of released bits and information adding.

The GCC [14] target for the MIPS 1 ISA has been used for this study.

3.2 Basic Implementation

This implementation consists in hacking the C GNU assembler code (GAS) and is divided into two tasks:

[1]REGIMM is a special MIPS 1 opcode that represents a branch type and refers to another field inside the instruction to determine precisely what sort of `branch` we are facing.

```
static void append_insn ( ip, …)
     struct mips_cl_insn *ip;
     …
{
 …
 if(!mips_opts.mips16)
 // checks if it is not mips16 instruction
  {
   if ( ((*ip).insn_opcode>>26)==(unsigned)1 )
   // checks if it is a REGIMM instruction,
   // i.e. opcode is 000001
    if(((((*ip).insn_opcode)>>16)&0x001f)==(unsigned)1)
    // checks if it is BGEZ  MIPS 1 instruction
      (*ip).insn_opcode=(*ip).insn_opcode|0x00080000;
      // flips the 18th bit which indicates the type
      // of REGIMM instruction
   }
 …
}
```

Figure 1: Adjustment—in bold—of the `tc-mips.c` file, part of GAS targeted for MIPS, in order to change the setting of an instruction. By way of example, the 18th bit of all the BGEZ instructions is set to one instead of zero. Indeed, the opcode points to five bits that model only a group of four instructions, leaving the 18th unused.

Step 1: Identification of released bits.

This is in fact a pre-implementation step which appears to be the relevant and difficult part of the process. First, the designer has to identify what sort of information—determinable at compilation time—the processor would need to improve its performance or reduce its complexity for instance, and then find out the most suitable instruction to carry that information—i.e., find an instruction that may contain unused bits or find a way for which it could be the case—e.g., tailoring the ISA.

Step 2: Adding Information

At this step, we make changes in GAS. This is achieved by adding C code in the `append_ins()` function of the `tc-target.c`[2] file (`tc-mips.c` file in our case). The nested `if` loops are now also used to check if the current instruction is one of those you are interested in—which can turn out to be difficult—and when it is the case the content of the instruction is adjusted like you want it to be. Figure 1 shows how GAS is tuned to both locate the BGEZ instruction and introduce inside it a single bit of information by using the unused 18th bit. Indeed, the REGIMM opcode refers to a 5-bit field that contains the concerned type of `jumps` among four possible `branches`. Consequently, at least one bit is never set.

This method is well suited when all instances of a concerned instruction have to be modified. Otherwise, all checks like dependencies have to be implemented inside the assembler itself which can be difficult.

3.3 Independent and Fine Grain Implementation

[2]This file is part of the binutils package files from GCC: binutils-version.no/gas/config/tc-target.c

```
struct mips_set_options {
...
/* This variable will be set thanks to the
.set add_information label in the .s file */

  int add_info;
}
```

Figure 2: Definition of a new member in the `mips_opts` structure—part of the binutils-version.no/gas/config/tc-mips.c file—which will hold information about its corresponding `.set add_information` label.

This implementation differs from the previous one mainly because all the tests consisting in identifying the instruction you want to modify are done by a standalone software. Thus, the assembler is quasi only tuned to change the instruction code, keeping the robustness of the tool. This novel method gives much more flexibility to the user and allows more accurate operations. Indeed, we add a new simple software layer between the cross compiler and the assembler. This layer identifies the instructions the user is interested in. Once such a dependency is located, a label is added in the assembler input file just before the considered instruction. For example, we may use a label like `.set add_information`. This file is then given as input to the assembler that will set the identified unused bits—as you specify them to be set—each time the label is encountered.

The first step described in section 3.2 is still required.

Then the designer must create a parser that will instrument and perform static classification on the assembly code by tagging it thanks to newly defined labels relevantly put in the code. The algorithm to implement these labels depends on the microarchitect's requirements and specification and he can create this parser with the language he wants like lex & yacc [12] or perl [17].

Finally, some surgical changes have to be done in the assembler:

1. A new variable per label must be defined (figure 2) which will act as a flag to indicates whether the associated label is currently seen.

2. This variable will be set by the `s_mipsset` function[3] each time its corresponding label is encountered in the assembly code as shown in figure 3.

3. Using the state of the variable, the part of assembler code which inserts opcodes for instructions will add information to the bits identified in step 1. Figure 4 shows how to do so. Indeed, code is introduced in the `append_insn` function[4] that checks if the variable has been set and whether the current instruction is the one in which the information should be added.

[3]the `s_mipsset` function belongs to the binutils-version.no/gas/config/tc-mips.c file.
[4]also part of the tc-mips.c file.

```
static void s_mipsset (x)
     int x ATTRIBUTE_UNUSED;
{
  ...
  else if (strcmp (name, "noreorder") == 0)
    {
      mips_emit_delays (true);
      mips_opts.noreorder = 1;
      mips_any_noreorder = 1;
    }
  /* This checks if this .set add_information
     label is present in the assembly  file and
     sets the mips_opts.add_info variable */
  else if (strcmp(name, "add_information") == 0)
    mips_opts.add_info=1; //.set add_information
  ...
}
```

Figure 3: Update of the new variable—`add_info`—to one each time the corresponding label is met within the assembly code.

4. Finally the variable is automatically reset just after the information has been added.

This second method provides more flexibility to the user. Dependency checking is implemented in a different layer thereby avoiding the cumbersome task of modifying assembler code for this purpose. Also the changes which are to be made in the assembler code for this method are simple as compared to those in method 1.

Figure 5 shows how our approach is integrated into a standard processor environment, thus proposing a flow to have an influence on various metrics (instruction throughput, area, ...) by making complementary hardware and software efforts.

4. EXPERIMENTAL METHODOLOGY

This Section describes the environment in which the results presented in section 5 were obtained—i.e., the compiler toolchain and its configuration, the chosen processor and its models—and finally the benchmark used to validate our work.

4.1 The Cross Tool Chain

In many system contexts, notably embedded ones, a cross tool chain may be necessary to compile a high level user code dedicated to run on a processor different from the host machine.
The cross tool chain we are talking about in this paper is based on GCC, a widespread free C compiler [3]. It is made up of six major components: the cross compiler itself (e.g., GCC version 2.95.2), an assembler (GAS) and a linker (GLD) part of various utilities (e.g., Binutils version 2.13), a library (e.g., newlib version 1.12.0) and a debugger (e.g., GDB version 6.0).

The standard compilation flow which consists in compiling, assembling and linking user programs is a bit different of the one we propose, as we will see section 4.6.

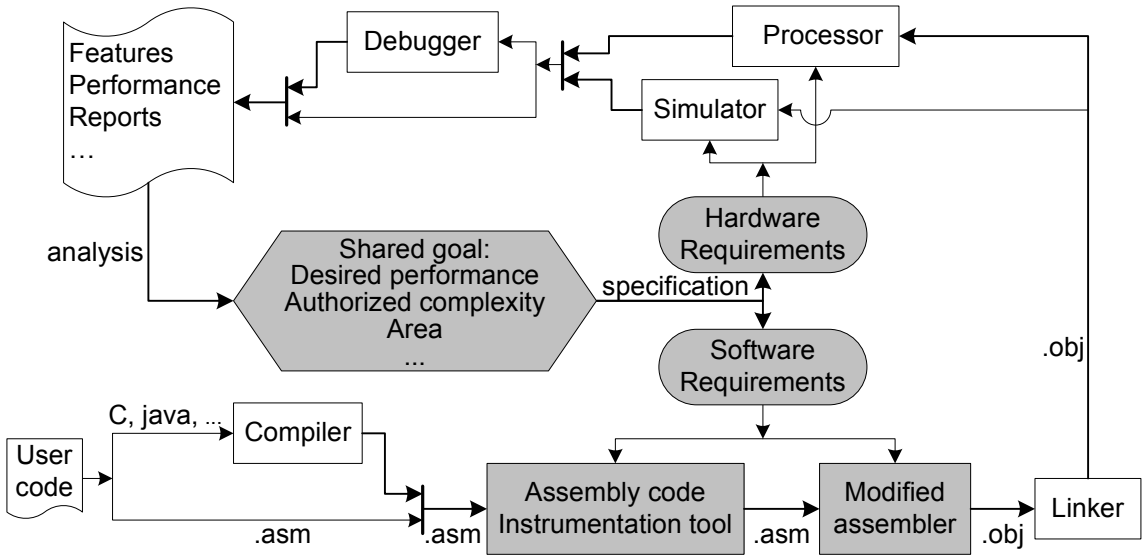

Figure 5: Overview of a processor environment including our proposal (in grey) independently of the compiler tool chain. What is called the `shared goal` represents the essence of this flow in the sense that the microarchitects have to analyze the current processor features and see if they fit to what they would like. From that analysis, hardware and software tied specification must come out to detail what must be modified in the processor (and the simulator), what instrumentation tool must be created and how the assembler must be tailored.

```
static void append_insn ( ip, … )
       struct mips_cl_insn *ip;
{
 …
 if (!mips_opts.mips16 && mips_opts.add_info)
 // checks if it is not mips16 and add_info variable
 // has been set by .set add_information label.
  {
   if  ( ((*ip).insn_opcode >> 26) == (unsigned)1 )
   // REGIMM instruction i.e. opcode is 000001
    if(((((*ip).insn_opcode)>>16)&0x001f)==(unsigned)1)
     // checks if it is BGEZ instruction
       {
        (*ip).insn_opcode=(*ip).insn_opcode|0x00080000;
         // flips the 2nd bit which indicates the type
         // of REGIMM instruction.
        mips_opts.add_info = 0;
         // reset the variable after the information
         // has been added.
       }
  }
 …
}
```

Figure 4: Effective static classification of instructions. As an example, a 5-bit field of the BGEZ MIPS instruction code, called the REGIMM field, is tranformed from x1 to x9 each time our `.set add_information` label is encountered just before the standard BGEZ instruction, letting so the hardware interpret the extra data.

4.2 Settings

All of the benchmarks were compiled with the `O3` and `funroll loops` settings which try to improve the program time execution. Due to "relocation truncated" errors, the `-G 0` option was applied in all cases as specified in [16]. The libraries were generated in the same way. One more run, which included the instrumentation tool this time, was also performed.

4.3 Reference Processor

The case study is a 5-stage single instruction issue, in-order processor, the MIPS R3000 based on a RISC architecture. This 32-bit load/store architecture does not contain Floating Point capability and it is made up of 32 32-bit general-purpose registers.

4.4 VMIPS Modifications

VMIPS is MIPS R3000 simulator [8] that we have improved to make it compatible with any integer programs in order to validate our work and the new toolchain through the EEMBC benchmarks [2]. Furthermore, it acts as a profiler by achieving control dependence analysis and delivers many other statistics.

4.5 EEMBC's Benchmarks

Only the EEMBC integer programs were used in this study to validate the new compilation flow and to demonstrate the kind of gains that can be obtained through our proposal. The results described in section 5.2 are based on what EEMBC calls the `regular` programs.

4.6 The Compilation Flow

Our approach has modified the standard cross tool chain. Indeed, an instrumentation tool has been introduced in between the compiler or the user assembly code and the assembler.

The input of this tool is an assembly code that can be obtained either by compiling a user's high level language such as C or directly provided by the programmer. After this step, the generated code feeds the assembler that has been previously modified to interpret the changes made by the instrumentation tool. Then, the generated object code could be linked to other object files by the linker, as usual. This flow is illustrated in figure 5.

5. APPLICATION

In this section, the application that has been chosen to use the proposed static classification is presented first—knowing that processor microarchitects are best to find out what application is the best suited for their processor—and then, we analyze for all EEMBC's integer benchmarks the performance of VMIPS used with the standard GCC tool chain and with the flow that takes into account the instrumentation tool. Obviously and as shown by the simulations, the programs have an impact on the speedup that the static classification can bring for a given application. The chosen evaluation metric we worked on is the overall execution time. Indeed, the simulator is clock-accurate and provides the number of clock cycles needed to perform an integer program. Finally, other potential applications in the domain of the data dependency, branch prediction and superscalar ressources are exposed.

5.1 Description

In older processors, branch instructions took more than 1 clock cycle because some time is needed to calculate their target addresses [9]. Indeed, to deal with target address computation, architectures use delay slots [11, 15] or interlock the pipeline [1, 4], which have a harmful impact on the processor's performance. To resolve this issue, many branch predictors based on various algorithms, of varying efficiency and complexity have been studied [5, 7, 10]. In any case, these proposals, based on static or dynamic approaches, need a dedicated piece of hardware and are mostly based on predictions. Static branch strategies—based on profiling or even branch direction—have poor efficiency and dynamic branch predictors use large amounts of hardware which also negatively influences performance. Hybrid solutions have been submited, with the drawback of increasing the `branch` Clock Per Instruction (CPI) [13].

This proposal does not deal with predictability, which can lead to high misprediction penalties even if many efficient value prediction mechanisms have been proposed, but more on anticipation by giving an effective information on the context of an instruction. Furthermore, the application we propose is based on control dependency handled by `delay slots` and combines branch prediction techniques with our static classification to improve the performance.

Imagine that we are able to know in advance that the next instruction is a stall cycle thanks to the value of one bit present in the current instruction. So, part of the effort would be pushed from hardware to software, which leads to a less expensive solution because silicon would be saved.

Usually programs of embedded systems have modest sizes. The intruction addresses are managed by control instructions like conditional branches and jumps that handle the program counter. Conditional branches make smaller jumps than `jump` instructions because they are intra-routine instructions whereas `jump` instructions are extra-routine instructions. Consequently, less bits are dedicated to compute an instruction address in a `branch` instruction than in a `jump` one. Now, if we come back to our 32-bit MIPS 1 architecture, 16 bits are reserved within a conditional branch instruction for the program counter computation and 26 bits in the case of a `jump` instruction. Then, we can easily imagine that embedded programs of 256 Kbytes—what it is quite important for embedded programs—use no more than 15 bits out of the 16 bits available in the `branch`'s offset field. So, for such programs the 16th bit of `branch`'s offset is free and could be used for doing something else like providing an extra information to the hardware as it is the case in the example of application we propose here.

Moreover, let us assume that the system uses branch prediction techniques. So, if a branch predictor foresees a branch not taken decision and, in addition, if the single bit decoding of the previously mentioned extra information indicates that the next cycle is a stall cycle, the processor then can jump directly to the second next instruction after the `branch`, thus avoiding to waste a clock cycle. So, in this example, the overall gain will be the unfilled `branch` not taken delay slot rate. Consequently, `branch likely` instructions[5] would not be needed any more since the Thumb[6] instruction set could be able to do their tasks more efficiently.

5.2 Simulations

Only the results of integer EEMBC benchmarks that are improved by the optimizer are presented in this paper, others have a gain inferior to 1% and are not mentioned further.

In our simulations, the multiplication and division latencies were set to 5 clock cycles and 35 clock cycles respectively, as suggested in [15].
For each benchmark, two simulations were run: one with the standard cross compiler tool chain and a other based on our flow.

The integer EEMBC benchmarks that take advantage of the instrumentation tool are mostly the Open Shortest Path First (OSPF), the Pointer Chasing (pntrch01), the Dithering (dither01) and the Text Processing (text01) programs with an average speed up of about 4%.

The best improvement was obtained for the OSPF program[7]. Indeed, our proposal allows to save up to 7.3% of the OSPF execution time because this benchmark is based on

[5]In the MIPS ISA, `branch likely` instructions are branches that nullify the instruction that comes just after—i.e., in their delay slot—if the `branch` is not taken.
[6]i.e., the bare minimum instruction set, the basis core.
[7]The OSPF programs implement the Djijstra search algorithm that is widely used in networking equipment such as routers.

frequent control transfer instructions and hence increase the likelihood of having branch instructions with delay slots that cannot be handled by the `O3` and `funroll-loops` options.

5.3 Potential Applications

This section describes a few applications where the static classification should be useful.

5.3.1 Superscalar reconfigurability

This example concerns superscalar processors. A recent study [6] has shown that the hardware ressources of such processors are not all used in the sense that the logic is not fully stressed: the maximum instruction issue rate is rarely reached. The proposed solution consists in using dynamically modified FPU ressources to make integer operations possible instead of systematically stalling the processor. This dynamic configurability of the processor is achieved thanks to a decision algorithm that scans the integer and FPU reservation stations to choose the optimal configuration at run-time: this task is very complex due to many dependencies. It appears that the decision could be enhanced through the use of compiler 'hints' inserted in the code, which highlights the fact that a hybrid method based on software information and hardware interpretation leads to more performance—hence the essence of our proposal.

5.3.2 Data dependency

This application does not deal with performance but tends to reduce power comsumption and alleviate both circuitry and hardware complexity. ARM and MIPS [1, 15] processors use different techniques to fix data dependencies. ARM implements a full dynamic approach with an interlock system within their scalar processors whereas MIPS is based on a static approach with the notion of delay slot. The former technique stalls the pipeline when a data dependency has been detected and the latter one introduces No OPeration instructions (`nop`) within the code thanks to the compiler/assembler. Both methods have advantages and drawbacks:

- Interlock minimizes the code size but increases the hardware complexity which must check for data dependency. Moreover, cycles are wasted each time the pipe is stalled.
- Delay slots minimize the hardware size and complexity but instruction memory require a larger space. An interesting feature is that delay slots may be filled in by independent instructions, thus saving execution time.

What would be interesting in this case is to combine these two mechanisms into one, thus taking their respective advantages and limiting their drawbacks. To do so, in case of small data memory—less than 32 Kbytes for instance—we can let the compiler tool chain make the dependency checking job instead of the hardware and then introduce that information through a free bit of the memory instructions. Now, `nop` instructions after memory instructions corresponding to unfilled delay slot can be removed from the assembly code. Consequently, the processor only has to decode this bit to know the data dependencies of memory instructions and their context instead of comparing register numbers. Thus, the processor will stall the pipeline according to the value of the mentioned bit and no hardware data dependency checking is necessary.

5.3.3 Branch predictor selection

For architectures that present several branch predictors, static classification could be used to select the most suitable branch predictor according to the branch type as mentioned in [18] but with the difference that our proposal keeps the same instruction format.

6. CONCLUSIONS

Providing more information to the processor is planned, among other things, to increase its performance, reduce the circuitry, save silicon, reduce the switching activity and hence the energy consumption, reduce the microarchitecture complexity and make testing and debugging easier.

The proposed new interaction between software and hardware through the instrumentation of the assembly code is a way for microarchitects to reach as much as possible the previous goals.

This work shows—through a real example—how to instrument the assembly code while keeping the instruction format unchanged. The proposal is built on indirect information shared at the runtime between the ISA and the processor, as well as the designer's smartness who has to find out what may alleviate the processor.

Even if the proposed static classification is well suited for scalar processors dedicated for embedded systems, it could be applied to other processor architectures like superscalar. Although the presented application was more to validate our flow and to illustrate the potential of our approach than to emphazes the application itself, it highlights a gain of up to 7% for a networking program running on a MIPS architecture.

Now, potential users of the presented flow can imagine how to exploit the assembler instrumentation in order to use static classification for their own application.

7. ACKNOWLEDGMENTS

We would like to thank the reviewers for their helpful and detailed comments, Markus Levy for using the EEMBC's benchmarks and the Swiss Commission for Technology and Innovation (CTI) for their support.

8. REFERENCES

[1] Arm processors and architecture, http://www.arm.com/products/cpus/index.html.

[2] The embedded microprocessor benchmark consortium, http://www.eembc.hotdesk.com/home.jhtml.

[3] Gcc home page - cross compiler tool chain, http://gcc.gnu.org.

[4] Tensilica, inc., http://www.tensilica.com.

[5] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. Patt. Branch classification: A new mechanism for improving branch predictor performance. *International Journal of Parallel Programming*, 24(2):133–158, June 1996.

[6] M. Epalza, P. Ienne, and D. Mlynek. Adding limited reconfigurability to superscalar processors. In *Proceedings of the 13th Int'l Conference on (PACT'04) Parallel Architecture and Compilation Techniques*, October 2003.

[7] M. Evers and T. Y. Yeh. Understanding branches and designing branch predictors for high-performance microprocessors. *Proceedings of the IEEE*, 89(11):1613–1620, November 2001.

[8] B. Gaeke. The vmips project, http://www.dgate.org/vmips. Version 1.1.3.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture : a Quantitative Approach*. Morgan Kaufmann Publishers, third edition, 2003.

[10] D. A. Jimenez. Reconsidering complex branch predictors. In *Proceedings of the 9th Int'l Symposium on High Performance Computer Architecture*, 2003.

[11] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 1988.

[12] J. Levine, T. Mason, and D. Brown. *Lex and Yacc*. O'Reilly and Associates, 2nd edition, 1992.

[13] S. Mahlke and B. Natarajan. Compiler synthesized dynamic branch prediction. In *Procceedings of the 29th annual ACM/IEEE Int'l Symposium on Microarchitecture*, 1996.

[14] R. Stallman. Using and porting gcc. Technical report, Technical Report Documentation GNU, 1995.

[15] D. Sweetman. *See MIPS run*. Morgan Kaufmann Publishers, 1999.

[16] D. Sweetman and N. Stephens. *IDT R30xx Family Software Reference Manual*, 1994.

[17] N. Torkington and T. Christiansen. *Perl Cookbook*. O'Reilly and Associates, 2nd edition, 2003.

[18] Q. Zaho and D. J. Lilja. Static classification of value predictability using compiler hints. *IEEE Transactions on Computer*, 53(8):929–944, August 2004.