Acta Cybernetica **21** (2013) 5–20.

Low Level Conditional Move Optimization*

Artyom Antyipin†, Attila Góbi†, and Tamás Kozsik†

Abstract

The high level optimizations are becoming more and more sophisticated, the importance of low level optimizations should not be underestimated. Due to the changes in the inner architecture of modern processors, some optimization techniques may become more or less effective. Existing techniques need, from time to time, to be reconsidered, and new techniques, targeting these modern architectures, may emerge.

Due to the growing instruction pipeline of modern processors, recovering after branch mis-predictions is becoming more expensive, and so avoiding that is becoming more critical. In this paper we introduce a novel approach to branch elimination using conditional move operations, namely the `CMOVcc` instruction group. The inappropriate use of these instructions may result in sensible performance regression, but in many cases they outperform the sequence of a conditional jump and an unconditional move instruction.

Our goal is to analyze the usage of `CMOVcc` in different contexts on modern processors, and based on these results, propose a technique to automatically decide whether the conditional move or the sequence of a conditional jump and an unconditional move should be performed in a given situation.

Keywords: assembly, low level optimization, compilers

1 Introduction

Low level optimization has always been an important part of code generation. Sensible performance improvements can be achieved simply by reordering instructions or using an alternative, but equivalent, instruction sequence. Modern compilers support numerous optimization techniques applied to the generated code. Upcoming microprocessors are usually designed to run existing code faster without any adaptation. To achieve this, instruction processing is split into several stages, forming the so-called instruction pipeline. Each stage of the pipeline depends on the output of its predecessor, hence the processor starts to process the instruction several clock cycles prior to the actual execution. In order to keep the processor

*Supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003).

†Dept. Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary, E-mail: {artyom,gobi,kto}@elte.hu

DOI: 10.14232/actacyb.21.1.2013.2

running, it is essential to keep the pipeline full. However, if the code being processed contains conditional branches, the processor has to choose one execution path. If there is a mis-prediction, the processor abandons the fetched instructions, which leads to several lost cycles, while the first instruction of the mis-predicted branch reaches the execution stage. During these cycles the executing engine is likely to be idle which, beside wasting time, also increases power leakage of the processor. The power gating technique has been proposed to address this issue but has not yet been adopted by any modern microprocessor [11].

Although modern processors use sophisticated branch prediction algorithms, prediction is practically impossible when the branch condition depends on random data. This makes *Worst-Case Execution Time* (WCET) estimation of the code containing such branches very hard, as the exact value of mis-prediction penalty does not depend solely on the pipeline length [10]. Therefore, a decrease in the number of conditional branches in the code may results in improvements in WCET estimations, and in making better use of the instruction pipeline. These ideas motivated us to look for possible approaches to branch elimination.

The paper is organized as follows. The next section gives an overview of the examined processor architectures and the instructions related to our approach. In Section 3, a first optimization attempt is detailed. The idea is to replace two possibly mis-predicted conditional jumps with a single, but unpredictable indirect jump. This method and its impact on the execution time is detailed there. Section 4 introduces the better approach of ours – total branch elimination in code generated for `if`/`else` constructions by manipulating operations performed within branches. Section 5 discusses related work. Finally, Section 6 concludes with pointing out future directions of work.

2 Preliminaries

The rest of the paper assumes that the reader has working knowledge on how processors work. Hence, in this section a short introduction is presented to the examined architectures (Section 2.1), the relevant (i.e. conditional) instructions (Section 2.2), with the conditional move detailed (Section 2.3). Section 2.4 demonstrates a trivial optimization, which can also be found in a recent version of the *GNU Compiler Collection* and the *clang* compiler.

2.1 Processor architecture overview

The microprocessor architecture overview provided in this section is rather simplified. Its intention is to provide enough information to understand motivation behind our attempts, while keeping information not related to this paper uncovered. Complete technical documentation is available publicly at the websites of the corresponding vendors [12, 3].

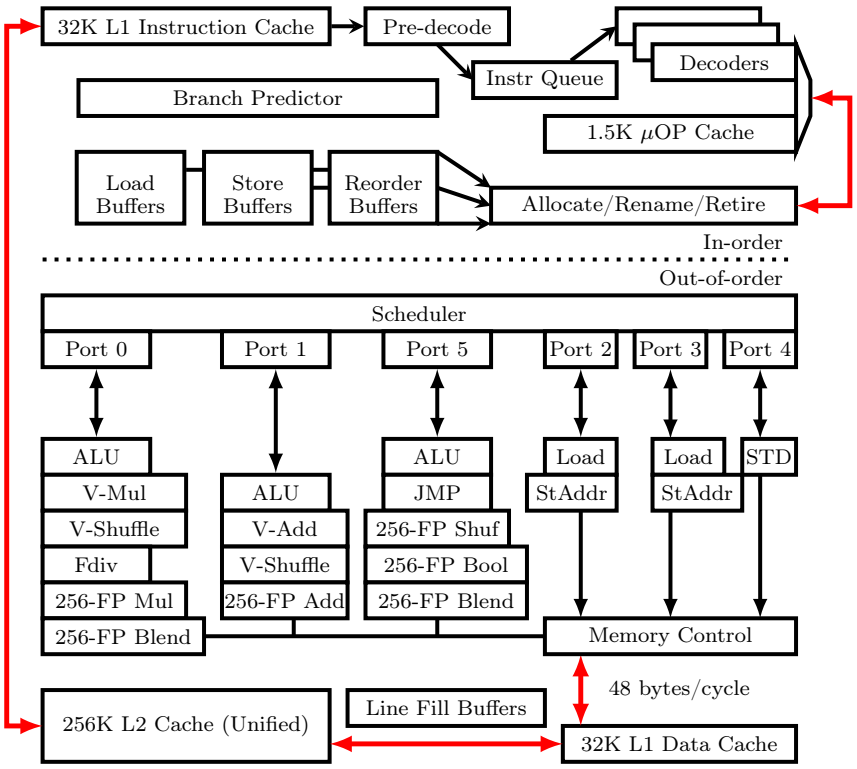

Figure 1: Intel microarchitecture with code name Sandy Bridge: Pipeline Functionality from [12]

2.1.1 Intel Sandy Bridge

Figure 1 depicts the pipeline and the major components of a processor core that is based on the Intel microarchitecture with code name Sandy Bridge. The pipeline consists of the following parts:

- In-order issue front-end, which includes
  - the branch prediction unit,
  - the instruction cache (L1i or ICache),
  - the instruction pre-decoder (4 units capable of micro and macro fusion),
  - the decoded ICache and
  - the micro-op queue, which decouples the front end and the out-of-order engine.
- Out-of-order execution engine which comprises of

  - the renamer,
  - the scheduler and
  - the execution core.

Branch mis-predictions affect both the front-end (directly) and the execution engine (indirectly). According to the technical manual [12] "mis-predicted branches can disrupt streams of μops, or cause the execution engine to waste execution resources on executing streams of μops in the non-architected code path", i.e. the micro-op queue of the front-end is emptied, and either instructions from the mis-predicted execution path are decoded, or, if these instructions were already decoded and cached within the decoded ICache, the queue is re-filled using the cached micro-ops. In both cases the execution engine is suspended until the first micro-op is queued.

2.1.2 AMD K10 and K12

The structure of the AMD Family 10h and 12h (also called K10 and K12 respectively) based microprocessors is similar in many ways to that of Sandy Bridge described above. Instruction processing is split into several phases:

- The *Branch Prediction Unit* decides which instructions are to be fetched from the L1 instruction cache.
- Instructions are fetched and decoded into macro-ops by the *Fetch-Decode Unit*.
- The macro-ops then are passed to the ICU (i.e. *Instruction Control Unit*) which is responsible for
  - macro-op dispatch,
  - macro-op retirement,
  - register and flag dependency resolution and renaming,
  - execution resource management,
  - interrupts and exceptions and
  - branch mis-prediction handling.
- Macro-ops are dispatched either to *Integer Unit* or *Floating-Point Unit*. Both of them consist of a scheduler and an execution unit. The execution unit in both cases contains three execution pipes capable of executing instructions of the appropriate type.

No mechanism of branch mis-prediction handling is described by the documentations [3], but the mis-prediction penalty is said to be at least 10 cycles.

The functionality of the AMD Family 10h and 12h microprocessors seems to be less complex than that of the microprocessors based on the Intel architecture with

code name Sandy Bridge. As a consequence, the use of the code generation methods introduced in this paper produces less sensible, but still measurable, impact on the execution time on AMD Family 10h and 12h microprocessors.

2.2 Conditional instructions overview

Before introducing conditional instructions, the corresponding functionality of microprocessors based on the x86 architecture must be clarified. Among other registers, the x86 architecture includes the probably most frequently used special-purpose register – the so-called `FLAGS` register. (The name `FLAGS` refers to the 16-bit register of the basic x86 architecture. The 32-bit and 64-bit extensions of the architecture also affect this register. The 32-bit and 64-bit extensions of the `FLAGS` register are called `EFLAGS` and `RFLAGS`, respectively). `FLAGS` represents the state of the processor. Its bits are called flags, and each of them has a different purpose. Generally, these flags can be split into two separate groups – the ones representing the state of the processor after executing a particular instruction (called *status flags*), and the ones that can be modified in order to change the state of the microprocessor. Whether the operation described by a conditional instruction is performed, depends on the state of the *status flags*, as explained below.

In assembly language, conditional instructions are usually written in the form `OPCODEcc`, where `OPCODE` is a conditional instruction itself, and `cc` (called *condition code*) is one of the predefined conditions over the state of the status flags. If the actual state of the status flags satisfies this predefined condition, the operation described by the conditional instruction is performed, otherwise no action is taken. As a consequence, in order to take advantage of using a particular conditional instruction, the status flags should be adjusted prior the execution of the instruction. Modification of the status flags is possible in the following ways.

- Some of the flags (`CF`,`DF`,`IF`) can be adjusted explicitly with an appropriate instruction.

- The value of the lower byte of `FLAGS` can be transferred into `AH`, modified, and transferred back to `FLAGS`.

- The whole value of `FLAGS`, `EFLAGS` or `RFLAGS` (depending on the current processor mode) can be transferred into stack, adjusted, and then transferred back.

- Status flags are also adjusted implicitly, when a particular instruction is executed. Generally, most of the arithmetic, logic and bit shifting instructions implicitly adjust these flags. Furthermore, the x86 architecture provides two special instructions – `TEST` and `CMP` – which perform the same operation as `AND` and `SUB`, respectively, but their result is not stored, but only flags are adjusted. This latter facility is used to explicitly compare values.

2.3 Conditional move instruction

The `CMOVcc` instruction was introduced in the P6 processor family (Intel Pentium II) and usually described using the syntax below. It should be noted that in this paper the AT&T assembly syntax is used. See [9] for details about differences between the AT&T and the Intel syntax.

`CMOVcc` *source*, *destination*

Here, *source* can be either a general-purpose register or an in-memory variable; *destination* is a general-purpose register, and *cc* is the condition code (see Section 2.2). The operation performed by `CMOVcc` is detailed below.

temp ← *source*
IF *condition* TRUE
  THEN
    destination ← *temp*;
FI;

The operation can be split into three sub-operations – namely loading the value of the *source* operand, evaluation of the condition, and storing the loaded value into the *destination* operand. Note that the load sub-operation is performed unconditionally, i.e. even if the condition is not satisfied. As a consequence, if an in-memory variable is used as a source operand, it is loaded to cache – which is likely to be unnecessary if the condition is not satisfied and the variable is not used by other instructions. Furthermore, is this case the address of the variable must be valid (i.e. point to memory accessible by the program) or else processor exception will be raised, even if no move operation is to be performed. So `CMOVcc` with an in-memory variable would rather be used only when the variable is also used by other unconditional instructions. This restriction makes `CMOVcc` useless for optimization in several cases, as loading the variable into a register and using that register instead always results in better performance. Despite this, in our research we investigated ways to achieve better performance by using `CMOVcc` instructions with both registers and in-memory variables as the first operand.

2.4 A trivial case

Consider the C code fragment 1. It contains a single conditional branch that depends on a single condition, and has a single assignment operation within its body. Without any optimization, this code may be compiled to the assembly code shown in code fragment 2. Two variables are compared using the `CMP` instruction (2), which adjusts the status flags as if y was subtracted from x. If x was less than y, i.e. arithmetic borrow has been generated out of the most significant bit position, then the `CF` flag was set, otherwise the `CF` flag was reset. If `CF` was not set, the conditional should be skipped (3), i.e. the conditional jump to the end of the body of the branch (5) should be performed. If `CF` was set (i.e. x was less than y),

Code fragment 1 Trivial case (C/C++)

```
1  unsigned int x, y;
2  if (x < y)
3  {
4    x = y;
5  }
```

Code fragment 2 Trivial case (conditional jump + unconditional move)

```
1  # assume x = %rcx, y = %rdx
2    cmpq $rdx, %rcx
3    jnc 1f
4    movq %rdx, %rcx
5  1:
```

the jump operation is not performed, and line (4), namely the body of the branch, is executed.

The code, produced by a compiler without optimization, provides the expected functionality, but the conditional jump has a good chance of causing branch mis-prediction, and of wasting 14 cycles[1] each time the code is executed. Due to macro-fusion and out-of-order execution, the net execution time of the instructions in the code above is either 1, 2 or 3 cycles, depending on the position of the code in memory and the preceding instructions. However, together with the branch mis-prediction penalty, the execution of the code is expected to take 15-18 cycles.

Note that with random input the prediction is likely to fail. Fortunately, the code fragment 1 can be easily optimized using an *if-conversion* [17]. With a minimal effort, the code generator notices that a `CMOVcc` instruction can be used, as the expected functionality matches perfectly the definition of `CMOVcc`, as described in section 2.3. In this case the code generator can generate the code shown in code fragment 3: the comparison operation is kept unchanged, and the sequence of the conditional jump and the unconditional move is replaced with a single conditional instruction. This code has exactly the same functionality, and has no branches – i.e. no branch mis-prediction can ever happen. As a consequence, the execution of this code will take constantly 2 cycles. Using this single optimization in algorithms with constructions similar to the one shown in code fragment 1 can dramatically increase performance of the generated code. A good example is the *maximum* algorithm: improvements of using this optimization are shown in figure 2.

This case is trivial to optimize, because the used high-level construct perfectly

[1] There is no official information about the mis-prediction penalty, but different Internet sources [2, 4] agree on the same value of at least 14 cycles on microprocessors with Sandy Bridge architecture. On processors with AMD K10 and K12 architecture this penalty is defined to be at least 10 cycles [3].

Code fragment 3 Trivial case (conditional move)

```
1  # assume x = %rcx, y = %rdx
2    cmpq $rdx, %rcx
3    cmovb %rdx, %rcx
```

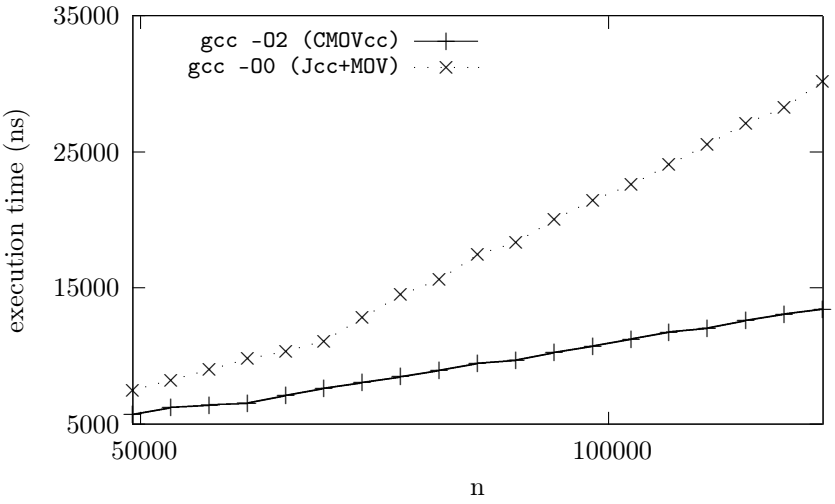

Figure 2: Maximum algorithm

fits the definition of `CMOVcc`. Popular compilers, like *gcc* [18] and *clang* [16], already support this optimization. It is worth mentioning that, probably because of problems discussed in Section 2.3, all optimizations involving the use of `CMOVcc` were disabled by default in older version of *gcc*. Newer versions (such as those above 4.5) of *gcc* have this optimization enabled – it is hard to tell exactly which versions, since no official announcement about this have ever been made.

In all tests included in this paper, the performance of our solutions was compared to the performance of the code generated by *gcc* with optimization enabled (`-O2`). Furthermore, we experienced no significant differences between code generated by *gcc* and *clang* for our test cases, and thus we assumed that the code generated by *clang* performs similarly to the one generated by *gcc*

3 Our first attempt

Consider the C++ function in code fragment 4. This function is given a pointer to some data, the length of the data and some threshold number x. It returns a tuple, containing the sum of the data items which are less than, greater than or equal to x, respectively. When the function is called, the body of the loop is executed

length times. When generating code for the body of the loop, popular compilers do recognize that a single compare operation is sufficient in this case, thus assembly code similar to code fragment 5 is generated. This code contains two conditional jump instructions, and if neither is taken, the third branch is executed. The main problem here is that the result of the comparison depends on potentially random data, and thus branch prediction is hardly possible in this case. As a consequence, this code contains two possibly mis-predicted jumps, which can be very costly, especially when executed within the loop.

Code fragment 4 Conditional sum (C++)

```
1  std::tuple<int, int, int> sum(int *ptr, int length, int x)
2  {
3    int eq = 0, lt = 0, gt = 0;
4    while (int i = 0; i < length; ++i)
5    {
6      if (ptr[i] < x)
7        lt += ptr[i];
8      else if (ptr[i] > x)
9        gt += ptr[i];
10     else
11       eq += ptr[i];
12   }
13   return std::make_tuple(lt, eq, gt);
14 }
```

Code fragment 5 Three-way branch, generated code

```
1    # assume %rdi = i,  %rsi = ptr, %ecx  = x
2    #        %r8d = eq, %r9d = lt,  %r10d = gt
3    movl (%rsi,%rdi,4), %edx
4    cmpl %ecx, %edx
5    jg do_gt
6    je do_eq
7  do_lt:
8    addl %edx, %r9d
9    jmp done
10 do_gt:
11   addl %edx, %r10d
12   jmp done
13 do_eq:
14   addl %edx, %r8d
15 done:
```

The main problem of the code generated for the body of the loop is the presence of two possibly mis-predicted conditional jumps. So our first intention was to decrease the number of the conditional jumps. Our main idea can be described as follows. Instead of performing a conditional jump, the pointer to the branch that should be taken is calculated using conditional move operations, and then an unconditional jump to this pointer is taken. This gives us the code shown in code fragment 6.

Code fragment 6 Three-way branch, single jump

```
 1    # assume %rdi = i,  %rsi = ptr, %ecx  = x
 2    #        %r8d = eq, %r9d = lt,  %r10d = gt
 3    movl (%rsi,%rdi,4), %edx
 4    leaq do_gt(%rip), %r11
 5    leaq do_eq(%rip), %r12
 6    leaq do_lt(%rip), %r13
 7    cmpl %ecx, %edx
 8    cmovg %r11, %r13
 9    cmove %r12, %r13
10    jmp *%r13
11  do_lt:
12    addl %edx, %r9d
13    jmp done
14  do_gt:
15    addl %edx, %r10d
16    jmp done
17  do_eq:
18    addl %edx, %r8d
19  done:
```

Unfortunately, the branch prediction unit of the examined processors cannot predict the single jump (in line 10), and hence this code constantly suffers from a single branch mis-prediction penalty. As a consequence, execution time of this code, opposed to the code shown in code fragment 5, depends neither on the input data nor on the inner state of the branch prediction unit.

After performing a series of tests, we came to the conclusion that the execution time of the code created using this method either equals to, or differs insignificantly from, the execution time of the code generated by *gcc*. As our attempt to minimize the number of branches did not lead to significant performance improvement, our goal changed to complete branch elimination, which led us to the approach we are to introduce.

4 Our approach

As we mentioned before, the main problem with the code generated by *gcc* for the function shown in code fragment 4 is the presence of two conditional jumps, and our goal is to completely eliminate branching, so that branch mis-prediction can never happen. Our idea is to rearrange the code in the following way. All the branches are executed unconditionally, and all the parameters used in the branches are assigned conditionally, i.e. depending on the condition, either set to the original value or to some neutral value determined by the operation (see code fragment 7). The new code provides the same functionality, and does not contain a single branch. It is worth to note that although this code performs poorly in the cases where branches can be predicted (e.g. if the function discussed in this section is used on sorted data), the execution time is halved in the general case.

Code fragment 7 Three-way branch, our approach

```
 1  # assume %rdi = i,  %rsi = ptr, %ecx  = x
 2  #        %r8d = eq, %r9d = lt,  %r10d = gt
 3  xorl %r11d, %r11d
 4  xorl %r12d, %r12d
 5  xorl %r13d, %r13d
 6  movl (%rsi,%rdi,4), %edx
 7  cmpl %ecx, %edx
 8  cmove %edx, %r11d
 9  cmovl %edx, %r12d
10  cmovg %edx, %r13d
11  addl %r11d, %r8d
12  addl %r12d, %r9d
13  addl %r13d, %r10d
```

In general, any `if/else if/else` construction that satisfies the restrictions listed below will perform better, if optimised according to our approach.

- All the conditions must use the result of a single assembly comparison operation, or at least two conditions must use the result of an assembly comparison operation.
- Overall cycles needed to execute all the operations of all the branches must be less than the overall possible branch mis-prediction penalty.
- All the operations of all the branches must have neutral values.

Note that our approach sets no restrictions on the exact number of parameters of the operations within the branches. Although the number of general-purpose registers is restricted, in-memory variables can also be used, as the penalty of the memory access is insignificant when compared to the penalty of a single mis-prediction.

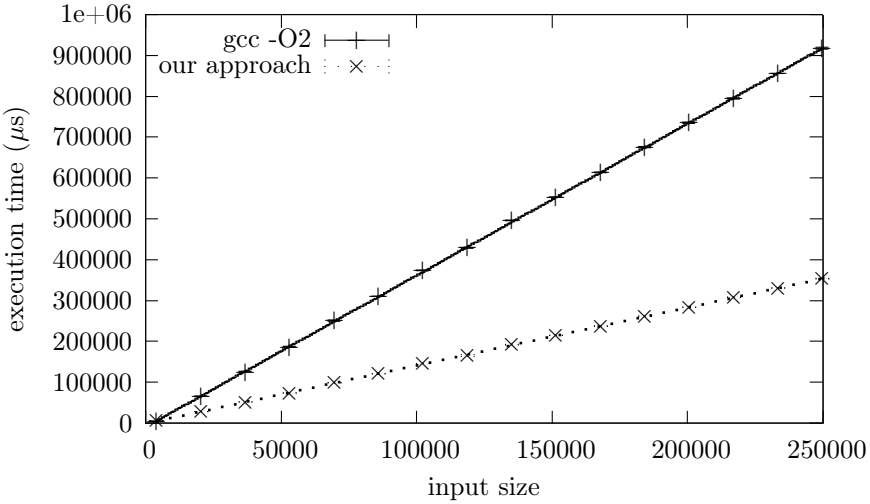

Figure 3: Execution time of the function specified in code fragment 4

Figure 3 shows results on the execution time of 1000 iterations of the code optimized by our approach and the one generated by *gcc*. The results are measured on an Intel Core i7-2620M processor, and a vector filled with pseudo-random data was used as an input. Table 1 contains further results on measuring the execution time of the function, including also the sorted input case, and the results taken on an AMD K10 processor as well.

For the measurements sorted and random input vectors of different sizes were used. Input sizes range through the rows of Table 1, while measurements on sorted and random input are depicted on the left and on the right, respectively. For every input size, experiments were carried out with *gcc* and with our hand-optimized code ("cmovcc"). The columns tagged "ratio" displays the execution time of our optimized code divided by the execution time of the one generated by *gcc*.

For each case the execution time was measured 12 times. In each experiment the first measurement was systematically larger than the others (probably because of cashing effects), therefore it was dropped. The remaining 11 measurements were averaged, and the variance was calculated. The variances were usually less than 1%, and never exceeded 5%, and thus they can hardly be observed on Figure 3. Assuming that the results are linear to the size of the input, we fitted a line on the measured data using the least squares method. In the case of input containing random data, the ratio in the slope of the lines are 2.619 ± 0.008. This allows us to conclude that our optimization yields 2.6 asymptotic speedup for the general case.

Our measurements are in accordance with the analysis presented in Section 2.4 on page 10. One can easily see that the code generated by gcc – i.e. the code using conditional jumps – performs very well when executed on sorted data (due to successful branch prediction) but shows dramatic performance decrease when

Table 1: Execution time of 1000 iterations (µs)

i7-2620M (based on microarchitecture code name Sandy Brigde)						
Input size	Sorted input			Random input		
	gcc	cmovcc	ratio	gcc	cmovcc	ratio
20480	12557	28969	230.70%	65543	28509	56.50%
69632	43238	99127	229.26%	250939	98447	60.77%
118784	72571	169463	233.51%	428812	164987	61.52%
167936	115746	239143	206.61%	614162	237441	61.34%
217088	160287	309680	193.20%	795009	307895	61.27%
249856	193510	356918	184.44%	918140	353916	61.45%
Phenom II X4 945 (AMD K10)						
Input size	Sorted input			Random input		
	gcc	cmovcc	ratio	gcc	cmovcc	ratio
20480	27420	32067	116.95%	79913	32073	59.87%
69632	93890	109755	116.90%	270162	109155	59.60%
118784	167020	193562	115.89%	461258	187265	59.40%
167936	226755	275454	121.48%	660903	273295	58.65%
217088	282321	357646	126.68%	853350	358102	58.04%
249856	323537	414665	128.17%	984064	414703	57.86%

random data is supplied. In contrast, the code created with our approach using CMOVcc shows no significant difference between the cases with sorted input and random input.

On random input the code optimized with CMOVcc was more than twice as fast as the one generated by *gcc*, both on the Intel and the AMD machines. On sorted input, *gcc* code performed twice as good as ours on Intel, and about 20% better on AMD.

5 Related work

To our best knowledge, all researches related to the usage of CMOVcc target only microprocessors with architecture different from x86, namely IA32, IA64 and Alpha [6, 19, 14, 17] and thus the technical documentations [3, 12, 13] provided by the vendors of the particular microprocessors remain the main source for the optimization techniques.

A study has been made in [5] on well-known optimization techniques including the one discussed in Section 2.4, but only the techniques already implemented and used by the particular compilers were considered. Furthermore, CMOVcc was used to optimize a HMMer search algorithm [15], and to mitigate timing-based side-

channel attacks by eliminating control flow dependencies [8]. In both papers the instruction was used within very specific cases, and no optimization technique has been proposed.

Another source of possible optimization techniques is the documentation of the popular compilers, but they are usually based on the mentioned technical documentations provided by the vendors of the microprocessors. Furthermore, in many cases implementation differs from the documentation as it contains modifications based on the feedbacks and proposals of the end-users.

6 Conclusions and Future work

In this paper we have studied branch elimination techniques based on replacing conditional jumps with conditional move operations. We have discussed the methods of branch number reduction and total branch elimination in code generated for the higher-level `if/else` constructions. The former one has proved to achieve only insignificant impact on the execution time of the produced code. The code created by using the latter method never suffers branch mis-prediction penalty, and hence outperforms the code generated by the popular compilers in the general case. Still, because of the increased complexity of the code, it performs poorly in some special cases, i.e. when no branch mis-prediction is caused by the compiler-generated code.

Although performance is not improved in the case of sorted input, the execution time of the code no longer depends on branch predictions. This has positive impact on WCET analysis, and makes its estimation more straightforward. This property can be extremely important in real-time systems [7].

In the future we will define the exact set of cases when our method could be used. Afterwards we plan to integrate our method into the popular compilers (e.g. *gcc* and *clang/llvm*) by providing appropriate plug-ins. This can serve as a convenient test-bed, and can speed up further research in this area.

References

[1] Зубков, С.В. *Ассемблер для DOS, Windows и UNIX*. Для программистов. ДМК Пресс, 2004.

[2] 7-Zip LZMA Benchmark, Intel Sandy Brigde. `http://www.7-cpu.com/cpu/SandyBridge.html`.

[3] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD Family 10h and 12h Processors*, February 2011. Publication Number: 40546.

[4] Anandtech - the bulldozer aftermath: Delving even deeper. `http://www.anandtech.com/show/5057/the-bulldozer-aftermath-delving-even-deeper/2`.

[5] Bik, A.J.C., Kreitzer, D.L., and Tian, X. A case study on compiler optimizations for the Intel® Core TM 2 Duo Processor. *International Journal of Parallel Programming*, 36(6):571–591, 2008.

[6] Chuang, W. and Calder, B. Predicate prediction for efficient out-of-order execution. In *Proceedings of the 17th Annual International Conference on Supercomputing*, pages 183–192. ACM, 2003.

[7] Colin, A. and Puaut, I. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2):249–274, 2000.

[8] Coppens, B., Verbauwhede, I., De Bosschere, K., and De Sutter, B. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE Symposium on Security and Privacy*, pages 45–60. IEEE, 2009.

[9] Dean Elsner, Jay Fenlason & friends. Using the GNU Assembler for the family. `http://www.cs.utah.edu/dept/old/texinfo/as/as.html#SEC150`, March 1993.

[10] Eyerman, S., Smith, J.E., and Eeckhout, L. Characterizing the branch misprediction penalty. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 48–58. IEEE, 2006.

[11] Hu, Z., Buyuktosunoglu, A., Srinivasan, V., Zyuban, V., Jacobson, H., and Bose, P. Microarchitectural techniques for power gating of execution units. In *Proceedings of the 2004 international symposium on Low power electronics and design*, pages 32–37. ACM, 2004.

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, June 2011. Order Number: 248966-025.

[13] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, March 2012. Order Number: 325462-042US.

[14] Klauser, A., Austin, T., Grunwald, D., and Calder, B. Dynamic hammock predication for non-predicated instruction set architectures. In *International Conference on Parallel Architectures and Compilation Techniques, Proceedings*, pages 278–285. IEEE, 1998.

[15] Landman, J., Ray, J., and Walters, JP. Accelerating HMMer searches on Opteron processors with minimally invasive recoding. In *20th International Conference on Advanced Information Networking and Applications*, volume 2. IEEE, 2006.

[16] Lattner, C. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, 2008.

[17] Mahlke, S.A., Hank, R.E., McCormick, J.E., August, D.I., and Hwu, W.M.W. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of 22nd Annual International Symposium on Computer Architecture*, pages 138–149. IEEE, 1995.

[18] Mitchell, Mark and Samuel, Alexander. Gcc 3.0 state of the source. In *4th Annual Linux Showcase and Conference*, 2000.

[19] Wang, P.H., Wang, H., Kling, R.M., Ramakrishnan, K., and Shen, J.P. Register renaming and scheduling for dynamic execution of predicated code. In *The Seventh International Symposium on High-Performance Computer Architecture*, pages 15–25. IEEE, 2001.