

Optimizing a 3D Image Reconstruction Algorithm: Investigating the Interaction between the High-Level Implementation, the Compiler and the Architecture

Tom Vander Aa[†] Lieven Eeckhout[‡] Bart Goeman[‡]
Hans Vandierendonck[‡] Tanja Van Achteren[†]
Rudy Lauwereins[⊙] Koen De Bosschere[‡]

[†]{tom.vanderaa,tanja.vanachteren}@esat.kuleuven.ac.be
ESAT, KULeuven, Kasteelpark Arenberg 10, B-3001 Leuven, Belgium

[‡]{leeckhou,bgoeman,hvdieren,kdb}@elis.rug.ac.be
ELIS, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium

[⊙]rudy.lauwereins@imec.be
IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

Abstract

Digital signal processing and multimedia workloads will be a dominant workload for computer based systems in the near future. In this paper, we evaluate the performance of an important media application, namely a relatively new 3D image reconstruction algorithm, on two platforms: a DSP processor (Texas Instruments TMS320C6701) and a high-performance general-purpose microprocessor (Alpha 21164). Prior to evaluating the performance of both architectural paradigms—very long instruction word (VLIW) versus an in-order superscalar organization—we optimized the algorithm by applying algorithmic optimizations as well as implementation-dependent optimizations. For the VLIW architecture, we obtained a 12X speedup for a 465×320 image; on the Alpha 21164, a 4X speedup was obtained. Thanks to this high speedup, this 3D image reconstruction algorithm becomes useful for real-time use. Next to evaluating the various optimizations, we also discuss the implications of these optimizations on the performance of various architectural structures, such as the branch predictor and the memory hierarchy.

Keywords: 3D image reconstruction, program optimization, VLIW, superscalar

1 Introduction

Digital signal processing and multimedia applications are becoming increasingly important for computer based systems. We can expect that multimedia applications will be a dominant workload for general-purpose microprocessors in the near future (if this is not yet the case today). As a result, microprocessor designers and researchers are having more and more interest in digital signal processing applications and multimedia workloads. However, it is not yet clear which architectural paradigm is best suited for this type of workload. Several propositions have been made: special-purpose multimedia processors that are based on ASIC design, versus general-purpose processors with multimedia extensions added to the instruction-

set architecture (ISA) to speed up digital signal and media processing. Examples of the latter approach are MMX, SSE and SSE-2 proposed by Intel, VIS by SUN, 3DNow! by AMD, AltiVec by Motorola, etc. In addition, it is not yet clear on what superscalar paradigm these processors should be based: very long instruction word (VLIW), in-order, out-of-order, explicitly parallel instruction computing (EPIC), with or without single instruction multiple data (SIMD).

Several studies have been reported in recent literature that compare the performance of digital signal processing and multimedia applications on different architectures. The study most related to the one presented here was done by Talla *et al.* [Talla et al., 2000] in which DSP and multimedia applications were evaluated on VLIW, SIMD and superscalar processors. Several kernels—such as a dot product and a finite impulse response filter (FIR)—as well as several applications—such as an ADPCM speech compression algorithm and a G.711 speech coding algorithm—were evaluated on the Texas Instruments TMS320C62xx processor and on the Pentium II processor (with MMX). In that study, the authors made use of assembly libraries and compiler intrinsics to optimize their applications.

In this paper, we take another approach. Instead of evaluating several widespread applications, we evaluate one single application that is relatively new, namely a 3D image reconstruction algorithm [Proesmans et al., 1996]. In addition, the algorithm is optimized into a form that achieves optimal performance. Indeed, during the optimization process, we not only make use of implementation-dependent optimizations as is done in most studies in this area. We also perform algorithmic modifications to the program to obtain a highly optimized algorithm for the platforms under consideration. Note that we only apply source to source transformations. We evaluate this highly optimized algorithm (and its various versions during this optimization process) on the eight-issue Texas Instruments TMS320C6701 VLIW microprocessor and on the four-issue Alpha 21164 in-order superscalar processor. This highly optimized algorithm achieves a 12X speedup over the original version on the VLIW 'C67 architecture for a 465 × 320 image, which makes the program suitable for real-time use. On the Alpha 21164, a speedup of 4X is achieved. In addition, we also discuss what the implications are of the vari-

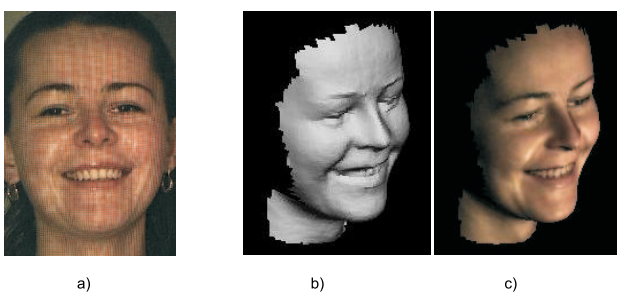


Figure 1: A one shot 3D image acquisition algorithm; (a) image, with projected grid; (b) reconstructed third dimension; (c) reconstructed scene, with textures.

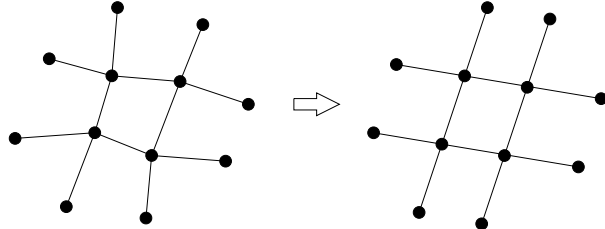


Figure 2: The SNAKE moves the grid for better reconstruction.

ous optimization steps on the performance of various architectural structures, such as the branch predictor and the memory hierarchy.

This paper is organized as follows. In section 2, the 3D image reconstruction algorithm is detailed. Section 3 discusses the architectures evaluated, namely the TMS320C6701 and the Alpha 21164. In section 4, the optimization steps performed on the algorithm are highlighted which are evaluated in section 5. Finally, we conclude in section 6.

2 SNAKE: 3D image reconstruction algorithm

The algorithm used is a one-shot 3D image acquisition system, which generates 3D shape descriptions from a single image, taken of a scene on which a simple grid is projected [Proesmans et al., 1996]. Viewed from a different angle, the grid appears deformed in the image, from which the three-dimensional shape can be extracted, see Figure 1.

The algorithm can be divided in roughly three steps. First, the cross points of the grid in the image are detected with pixel precision. A second step improves the accuracy of the grid with sub-pixel precision by applying an iterative energy minimizing snake-like process. The last step is the actual 3D shape extraction from the detected cross point coordinates. A description of the different steps in the algorithm, and how the cross point coordinates yield 3D shape can be found in [Proesmans et al., 1996].

Snake algorithm The snake algorithm, essential for a good 3D reconstruction, is the most time consuming part of the algorithm. It changes the grid to let it more closely follow the pattern lines, minimizing the energy function

$$E = \sum_{p=1}^{N_E} I(C_p, E_p) + \sum_{p=1}^{N_S} I(C_p, S_p) + \text{smoothing terms}$$

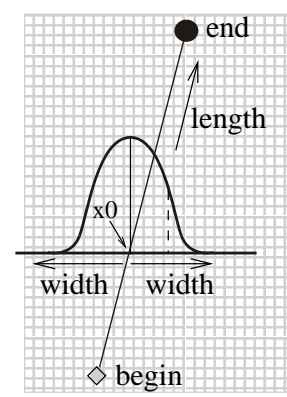


Figure 3: The inner loops of SNAKE.

```

1 compute_slope_of_line_between_cross_points ();
2 for (length = begin; length <= end; length++)
3 {
4   x0 = compute_x0_as_floating_point_number ();
5   width = width_left;
6   while( (w = gauss(x0-width)) > 0.0001 ) {
7     sum += w*image(width, length);
8     norm += w;
9     width--;
10  }
11
12  width = width_right;
13  while( (w = gauss(width-x0)) > 0.0001 ) {
14    sum += w*image(width, length);
15    norm += w;
16    width++;
17  }
18
19  integral = sum / norm;
20 }

```

Figure 4: Simplified pseudo code of the inner loops of SNAKE.

with N_E and N_S the numbers of cross points with (E)ast and (S)outh neighbours, respectively, and where $I()$ denotes the average intensity along the line segment between two neighboring cross points (see also Figure 2). Minimization moves the grid towards the darker positions in the image, i.e., the line centers.

Inner loops 98% of the total runtime in SNAKE is spent in its two most inner loops which calculate the integral $I()$ in the above formula. Therefore all optimizations focus on these inner loops. We will explain this code briefly.

As depicted in Figures 3 and 4, the second most inner loop iterates over the connection line between two cross-points (length loop, statement 2 in Figure 4), while the most inner loops iterate perpendicular to this connection line (width loop, statements 6 and 13 in Figure 4). Both iteration spaces are data dependent: the outer length loop depends on the position of the cross points at the pixel level; the inner width loops depend on the position of the connection line at the sub-pixel level. Note that x_0 , which is the intersection of the interconnection line between the two cross points and the horizontal ‘pixel’ line, see Figure 3, is calculated as a floating-point value. This results in sub-pixel level calculations. In these two inner loops the integral, corresponding to the $I()$ value in the above formula, is calculated.

3 Architectures

In this section, we briefly describe the architectures of the Texas Instruments TMS320C67 and the Alpha 21164 on which the optimizations for the 3D image reconstruction algorithm were done.

3.1 Texas Instruments TMS320C6701

The Texas Instruments TMS320C6701 [SPRU189D, 1999] ('C67) is a 32-bit floating-point VLIW DSP processor which is capable of executing at most eight 32-bit instructions per cycle at a clock rate of 167 MHz. Six of these eight instruction slots can be filled with floating-point operations. The processor has 32 general-purpose registers and is organized around two identical banks. Each bank has 16 registers and four ALUs: the D unit for load/store operations, the M unit for multiplications, the L unit and the S unit for arithmetic, logical, shift and branch operations. Each bank also has one single data bus connected to the register file in the other bank.

The pipeline is organized in three stages: fetch, decode and execute. Each fetch packet contains eight instructions. In addition, the fetch packet is split up into execute packets. An execute packet contains one instruction or two to eight parallel instructions. It is important to note that the execution of a new fetch packet is not started until the last execute packet, of the previous fetch packet is executed.

The execution stage is subdivided in ten (fully pipelined) stages (for floating-point operations) or five stages (for fixed-point operations) and different types of operations require different number of stages to complete their execution. This gives rise to different delay slots. For example, for a load operation it takes four cycles before the result being written can be read by another operation. A branch operation has five delay slots, which means that it takes five cycles between the branch execution and the execution of an instruction along the branch target. If the compiler is incapable of filling these delay slots with useful instructions, expensive execution cycles may be lost. Another important aspect of the 'C67 ISA is that each operation is conditional. I.e., each operation has a predicate associated that specifies whether the instruction needs to be executed. Predication is a useful static technique for hiding the penalty of branch mispredictions in pipelined architectures [Mahlke et al., 1995].

Since the 'C67 is a VLIW architecture, the performance of this processor heavily relies on the ability of the compiler to fill the eight instruction slots with useful instructions. Note also that the 'C67 has no branch predictor in contrast to most contemporary superscalar microprocessors.

The TMS320C6701 has separate data and instruction memories. The internal program memory can be mapped into the CPU address space or operated as a program cache. The measurements presented in this paper are for a configuration in which the program memory is organized as a RAM, not as a cache. A 256-bit wide path is provided to allow a continuous stream of eight 32-bit instructions from the 64 KB of program RAM to the CPU. The internal data RAM is 64KB in size and is divided into four banks of 16 KB. The CPU provides two data paths to the data memory system.

3.2 Alpha 21164

The Alpha 21164 [Alpha 21164, 1997] is a 64-bit RISC architecture which is capable of executing four instruc-

tions per cycle at a clock rate of 500 MHz. The architecture of the Alpha 21164 is in-order and can initiate the execution of two integer and two floating-point operations per cycle. Speculative execution is made possible in the 21164 to hide the penalty of branches. This means that the target address of a branch is predicted dynamically and that instructions are speculatively executed past speculated branches. On a correct branch prediction, the latency of the branch is effectively hidden. In case of a branch misprediction on the other hand, the speculated instructions will be squashed and the instructions from the correct path need to be fetched. The (conditional) branch predictor of the Alpha 21164 has 4192 two-bit saturating counters and is accessed by using the least significant bits of the branch address. The Alpha 21164 has an 8 KB direct-mapped L1 I-cache and an 8 KB direct-mapped dual-ported L1 D-cache. The L2 cache is unified, is 96 KB in size and is three-way set-associative.

4 Optimizations

Most of the optimizations presented in this section were proposed to improve the algorithm for the 'C67. However, these optimizations also improved the performance of the algorithm on the Alpha 21164 as will be shown in section 5.

The algorithm we originally started from, was developed and written in C by an image processing group [Proesmans et al., 1996]. Several algorithmic modifications were done in order to improve the performance of the original algorithm. For each of these optimizations, we have evaluated the accuracy of the modified algorithm and we did not notice any precision loss. Next to algorithmic optimizations, numerous implementational optimizations were done. In other words, the source code modifications done by a programmer who has knowledge of the algorithm, the architecture and the compiler used, reveal important optimization opportunities to the compiler that are unreachable through naive programming. Note also that some of these optimizations could be done by the compiler as well (in theory). In practice however, we had to help the compiler to do some of these optimizations. Other optimizations on the other hand, require a good understanding of the algorithm and the underlying architecture.

The optimizations done to the original code are the following:

1. **Efficient rounding.** The TMS320C6x C compiler user's guide [SPRU187E, 1998] recommends the use of *intrinsics*. One of the intrinsics suggested is for efficiently converting a floating-point value to an integer value, namely `_spint`. This function call will be translated by the compiler to a special assembler instruction (SPINT) for converting a floating-point value to an integer value. A type cast on the other hand, calls an expensive rounding function for calculating the conversion.
2. **Adding a black border.** In order to be able to eliminate checking for image borders in the inner loop of the algorithm, we have added a black border around the image. This algorithmic optimization did not affect the accuracy of the algorithm but did improve the performance of the algorithm.
3. **Enabling software pipelining.** The inner loop of the algorithm has bounds that are data-dependent. In other words, the number of iterations is unknown prior to execution. This

Normal execution					
Iteration	Order within iteration →				
1	A1	B1	C1	D1	
2	A2	B2	C2	D2	
3	A3	B3	C3	D3	
4	A4	B4	C4	D4	
5	A5	B5	C5	D5	
Software pipelining					
Iteration	Order within iteration →				
prologue 1	A1				
prologue 2	B1	A2			
prologue 3	C1	B2	A3		
iteration 1	D1	C2	B3	A4	
iteration 2		D2	C3	B4	A5
iteration 3			D3	C4	B5
epilogue 1				D4	C5
epilogue 2					D5
epilogue 3					D6

Figure 5: The principle of software pipelining. Although in the optimized loop steps of different iterations are being executed per iteration cycle, it is clear these steps are being executed in the same global order. It is necessary to provide a prologue and epilogue in order to get the loop into it’s steady state.

makes it impossible for the compiler to, for example software-pipeline this loop (as documented in [SPRU198C, 1998]). Since software pipelining is the most important technique the TI compiler uses to parallelize for-loops, the instruction-level parallelism without it, is very low. However, we experimentally observed that the number of iterations could be fixed without a notable loss of precision. Note that the Gauss distribution curve quickly falls to very small values, see Figure 3. We fixed the number of iterations of the inner width loops to 4, compare Figures 4 and 6 ‘before loop fusion’. The fixed number of iterations gives opportunities to the compiler to easily perform loop optimizations which result in significant performance speedups. For example, software pipelining requires a minimum number of iterations before the technique results in speedup because of the relatively long prologue and epilogue, see Figure 5.

- Loop fusion.** The kernel of the algorithm contains two inner loops. By fusing these two inner loops, see Figure 6, a larger number of iterations are exposed in one loop making software pipelining more valuable since prologue and epilogue are relatively shorter.
- Address calculation optimization using an induction variable.** In the algorithm, each pixel needs to be multiplied with a value from the Gauss distribution function. This Gauss distribution function is implemented as a lookup table (LUT). In this optimization step, we have optimized the address calculation for addressing this lookup table by noting that the distance between two accesses to the lookup table in the same loop was always 200 indices. We used this information to create an induction variable to keep track of the index in the LUT, see Figure 7.
- Eliminated norm.** As can be seen in Figure 4, the calculation of the integral involves a division by a norm in the outer length loop. This is necessary because the weight function we use, i.e. some

```

1 /* before loop fusion */
2 for(width = center-1; width<=center-4;width--) {
3     w = gauss(x0-width);
4     sum += w*image(width, length);
5     norm += w;
6 }
7
8 for(width = center; width<center+4;width++) {
9     w = gauss(x0-width);
10    sum += w*image(width, length);
11    norm += w;
12 }
13
14 /* after loop fusion */
15 for(width = center-4; width<center+4;width++) {
16     w = gauss(x0-width);
17     sum += w*image(width, length);
18     norm += w;
19 }

```

Figure 6: Loop fusion.

```

1 /* before address calculation optimization
2    -- gauss macro expanded */
3 for(width = center-4; width<center+4;width++) {
4     w = LUT [_spint ((200*(x0-width)) + c0)];
5     sum += w*image(width, length);
6     norm += w;
7 }
8
9 /* after address calculation optimization */
10 width = center - 4;
11 w = _spint ((200*(x0-width)) + c0);
12 for (; width<center+4;width++) {
13     v = LUT [w];
14     sum += v*image(width, length);
15     norm += v;
16     w += 200;
17 }

```

Figure 7: Address calculation optimization.

discrete points from the gauss curve, does not add up to one.

However, since we fixed the number of iterations in optimization step 3, the norm that was calculated now always has nearly the same value. By replacing the norm by an actual constant, we can eliminate the addition (`norm += w;`) in the inner width loop. In addition, we can move the division (`integral = sum/norm;`) outside the length loop.

- Reduction in strength.** In this optimization step we have traded complex and expensive operations, such as floating-point multiplications, for less expensive operations, such as additions. This optimization technique is well known as reduction in strength [Aho et al., 1986] and was applied to speed up the calculation of x_0 .
- Efficient division.** The ‘C67 C compiler user’s guide [SPRU187E, 1998] recommends the use of the intrinsic `_rcp` for calculating a division. So, instead of writing a/b , the following is recommended $a * _rcp(b)$, i.e., a multiplication with the reciprocal. The intrinsic `_rcp` was used to speed up the calculation of the slope of the interconnection line between the two cross points [Foley et al., 1995].
- Using fixed-point calculation and avoiding NOPs.** On the ‘C67, floating-point operations are more expensive than integer operations, because the

```

1  /* before transformation */
2  for (i = 0; i < n; i++) {
3      X;
4      for(j = 0; j < m; j++) {
5          Y;
6      }
7  }
8
9
10 /* after transformation */
11 X1;
12 for (i = 0; i < n; i++) {
13     for(j = 0; j < m; j++) {
14         Y;
15     }
16     X2;
17 }

```

Figure 8: Filling up branch delay slots.

pipelines of the floating-point functional units are much deeper (up to 10 execution stages versus 5 stages for integer operations [SPRU189D, 1999]). Fixed-point variables have the same or better precision than their floating-point counterparts with the same number of bits, but their dynamic range is limited. In cases where the actual range of the variable was limited, we could replace the use of floating-point variables by integer variables. Floating-point to fixed-point conversion eliminates floating-point calculations and in addition, expensive floating-point to integer conversions are no longer required. This conversion was applied in three cases:

- (a) The `length` loop in Figure 4 iterates over the pixels between two cross points. This is done by setting up a line between the cross points in floating-point calculus and determining the pixels nearest to this line. But since pixel coordinates are within a limited range—i.e., the dimensions of the image—fixed-point calculus can be used efficiently for calculating x_0 . The numbers before the (imaginary) binary point represent the pixel coordinate and the numbers after the binary point represent the sub-pixel precision.
- (b) Every integral is a weighted sum, where the weights w are stored in a lookup table. The lookup table has a granularity of 200 elements per unit and a range between $-5.0F$ and $5.0F$. So in total, the lookup table contains 2,000 elements. This lookup table is indexed in the inner loops of the algorithm by $x_0 - \text{width}$ which is a floating-point value. Because of the limited range (only 2,000 elements), this index can be converted to a fixed-point value while achieving the same precision.
- (c) The values in the lookup table are also floating-point values and are limited from -1.0 to 1.0 . These values can thus also be converted to fixed-point.

Another optimization that was done in this version, was avoiding NOPs. The 'C67 has five delay slots for branch instructions. Or in other words, as much as 40 instruction slots are to be filled up with useful instructions after a branch. If the compiler is not capable of filling up these delay slots with useful instructions, NOPs need to be inserted and expensive execution cycles will be lost. To avoid the insertion of NOPs in case of

a branch, we have moved as much code as possible past branches. In some cases, we had to apply code duplication in order to move code past branches, see Figure 8. In this figure, X1 and X2 are slightly modified versions of statement X to guarantee the correctness of the algorithm.

Also floating-point operations have delay cycles. When a floating-point operation is started in cycle x , the result will only be available in cycle $x + d$, with $d > 1$. If an instruction needs to be scheduled that is dependent on this instruction, the former instruction can only be scheduled in cycle $x + d$, which might imply that NOPs need to be inserted in cycles $x + 1$ to $x + d - 1$. In order to hide the latency of these long-latency operations, it is beneficial to merge various computation flows.

10. **Loop unrolling.** Because the body of the inner loop did not have enough instructions to keep the 8 functional units working, we unrolled the `length` loop, see Figure 4, once and merged the two iterations.

5 Results

5.1 Methodology

The results presented in this section are obtained using the following tools. For the 'C67, we used the Code Composer Studio v1.00 [Code Composer, 1999]. It comes with compiler version 3.01. Later on, we used a newer version of the Code Composer Studio, namely version 1.20 which comes with compiler version 4.00. Compiler options used are: `-mv6700` to generate 'C67-specific code, full optimizations (`-o3`) and all the options to get compiler feedback (`-kss -alsx -mw -os -on2`).

On the 'C67 we used the built-in hardware counters to measure clock cycles, NOP and non-NOP instructions. We also analyzed the executables and counted the functional unit usage for the different versions of the program. The execution count of each loop was calculated using the basic block profiling tool `bprof`. This way we could calculate the instructions per cycles (IPC) for the 'C67.

For the Alpha 21164, we used two compilers, namely the GNU C compiler `gcc` version 2.95 with the highest optimization `-O3` and the Alpha C compiler `cc` version V6.1-011 with the highest optimization `-O4`, the flag for inter-file optimization `-ifo` and the flag for architecture-specific optimizations, mainly instruction scheduling `-arch ev56`.

The results for the Alpha 21164 were obtained on a DEC 500ua station using the UNIX time utility. We also made use of ATOM [Srivastava et al., 1994], a binary instrumentation tool. ATOM is capable of instrumenting functions, basic blocks and individual instructions. As such, profiling tools, e.g., for measuring branch or cache behaviour, can be easily built.

The numbers reported are for a big image containing 465×320 pixels.

5.2 Evaluation

Figure 9 shows the execution time in seconds for the various versions as they were obtained by performing the optimization steps detailed in section 4. As can be seen from this figure and as expected, performance generally improves while additional optimizations are done. This is really pronounced for the 'C67, especially for optimization steps 3 (software pipelining) and 6 (eliminated norm). For the Alpha 21164, optimization steps 5 (optimized address calculation) and

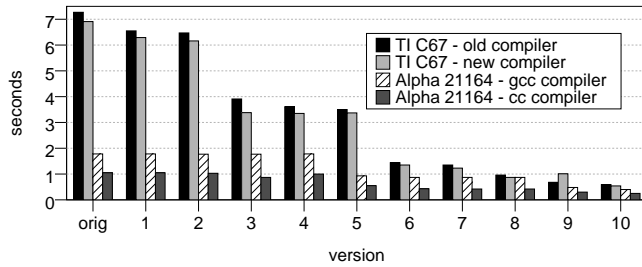


Figure 9: Execution time in seconds for the various versions.

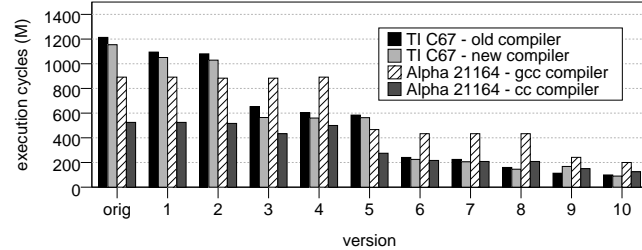


Figure 10: Execution cycles (in millions) for the various versions.

9 (fixed-point calculation and avoiding NOPs) result in significant speedups.

The algorithmic and implementational optimizations have more effect on the VLIW architecture than on the superscalar RISC architecture. I.e., the total performance increase (relatively speaking) is larger for the 'C67 than for the 21164: 12X versus 4X, respectively. Obviously, this is due to the fact that the algorithm was specifically optimized for the VLIW architecture, see section 4. This high speedup of 12X for the 'C67 shows that the performance on a VLIW architecture is highly sensitive to the source coding and the compiler, which is a general property of VLIW architectures. However, the superscalar Alpha 21164 also significantly benefits from these optimizations. Obviously, this is due to the fact that several optimizations are less implementation-dependent and will thus be beneficial for a broad range of architectures, for example the address calculation optimization (step 5) and the eliminated norm computation (step 6).

Figure 9 presents the execution time in seconds for the various (optimized) versions of snake. Figure 10 on the other hand, presents the number of execution cycles in millions. This graph could be easily obtained by dividing the data of Figure 9 with the respective clock frequencies, 167 MHz for the 'C67 and 500 MHz for the Alpha 21164. Note that the fact that the execution time of the highest optimized version (version 10) on the Alpha 21164 is smaller than on the 'C67 in Figure 9, is solely due the faster clock frequency of the 21164. Indeed, in Figure 10, the number of clock cycles is larger for the Alpha 21164 than for the 'C67 for the highest optimized version of snake (version 10).

It is also interesting to make a note on the power consumption of both microprocessors. The Alpha 21164 consumes 41W max power [Alpha 21164, 1997]. The 'C67 on the other hand, consumes at least a factor 10 less power under a worst case scenario [SPRA486B, 1999]. Since the execution times are comparable on both microprocessors, we can expect that executing the algorithm on the 'C67 will be far more energy-efficient than executing the algorithm on the Alpha 21164.

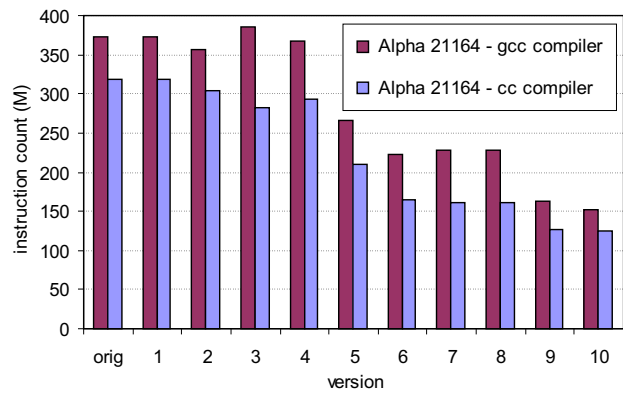


Figure 11: Dynamic instruction count for the Alpha ISA.

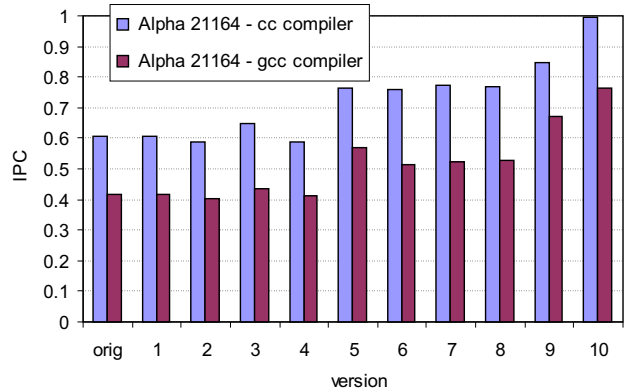


Figure 12: IPC for the Alpha 21164.

5.3 Discussion

An interesting aspect to be investigated in more detail is why performance gets better with additional optimizations. Or in other words, how is the behaviour of the optimized program experienced by the architecture? How is the functional unit utilization affected? What about the memory behaviour of the optimized program? To answer these questions we have set up several experiments on the Alpha 21164 using ATOM [Srivastava et al., 1994], a binary instrumentation tool. As mentioned before, ATOM can be used to build simple simulators efficiently.

Instruction count. In a first experiment, we have measured the dynamic instruction count, or the number of instructions executed to complete the program. As is shown in Figure 11, the dynamic instruction count generally goes down with an increasing number of optimizations applied. This can be explained by the fact that 'useless' instructions are removed by performing these optimizations. If we divide the data in this graph with the data in Figure 10, we obtain the number of instructions executed per cycle (IPC) which is presented in Figure 12. Although the dynamic instruction count decreases, IPC increases with an increasing optimization level. Note also that although the Alpha 21164 is a four-wide superscalar machine, IPC only reaches 1 for the highly optimized version of snake compiled with the cc compiler.

Instruction mix. Next, we looked at the instruction mix, see Figure 13. For both compilers, a clear

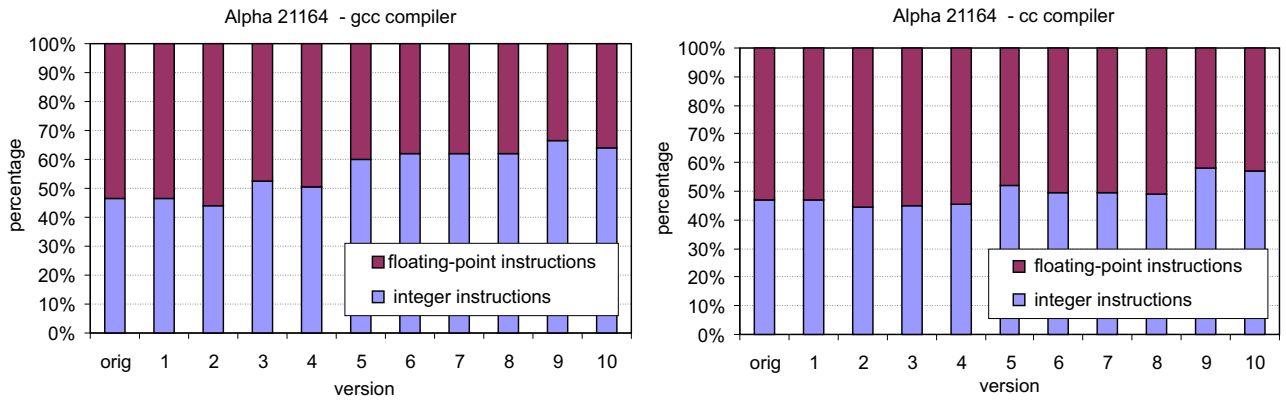


Figure 13: The instruction mix for the Alpha gcc and cc compiler version on the left and the right, respectively.

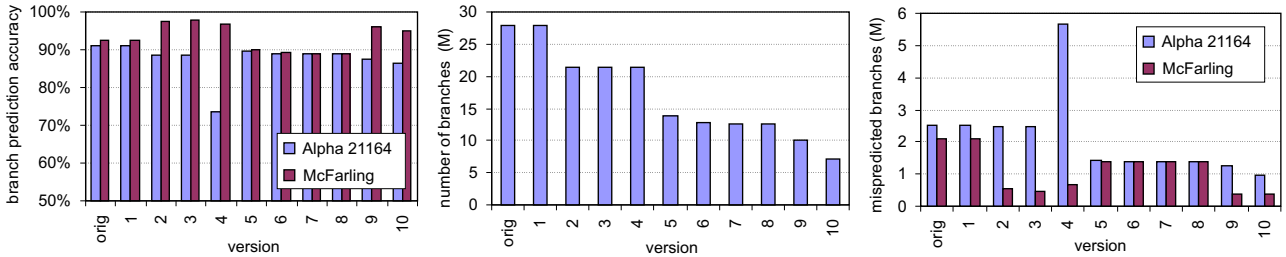


Figure 14: Branch predictability for the Alpha gcc compiler version; numbers are reported for both the branch predictor available in the Alpha 21164 and a simulated hybrid McFarling branch predictor. Left graph: branch prediction accuracy; middle graph: number of branches (in millions); right graph: number of mispredicted branches (in millions).

reduction in the number of floating-point instructions can be observed in step 5—the expensive LUT address calculation was implemented using an induction variable—and step 9—using fixed-point instead of floating-point calculation for accessing the lookup table. In all the other cases, see the cc version in Figure 13, the (relative) number of floating-point instructions increases with additional optimizations. This is due to the fact that most optimizations lead to more efficient code with a reduction in the number of integer instructions as a result. The (absolute) number of floating-point instructions remains nearly the same (to perform the computations in the algorithm), which results in an increase in the relative number of floating-point instructions.

Branch prediction. Next, we measured the branch prediction behaviour on the Alpha 21164. The Alpha 21164 implements a simple branch prediction scheme that uses a PC-indexed table of 2-bit saturating counters [Alpha 21164, 1997]. We also investigated what would happen with a more sophisticated branch prediction scheme that uses the McFarling method [McFarling, 1993]. The McFarling branch predictor is a hybrid branch predictor that uses a meta predictor to choose between a local and a global branch predictor. Figure 14 (on the left) shows the branch prediction accuracy, or the relative amount of branches that is predicted correctly, for both schemes. The general trend is that the accuracy goes down with increasing optimization steps. This can be explained by the fact that the number of branches decreases with the various optimization steps, see the middle graph of Figure 14. However, almost no mispredicted branches are removed. Indeed, the total mispredicted branches stays almost the same, see Figure 14 on the right.

Optimization step 4 (loop fusion) seems to be a special case: the branch prediction accuracy is significantly lower than in the other optimized versions. This can be explained as follows: due to loop fusion, the accesses to the lookup table (through the index $x0 - width$) alternately have a positive and a negative index since now we calculate from left to right over the horizontal line in one loop. This caused the if-test to detect positive or negative values to alternate as well, which totally confused the branch predictor. In the original situation (before loop fusion), positive and negative indices reside in separate loops leading to separate static branches for positive and negative indices. These two static branches are mapped to different entries in the branch predictor making it possible to the branch predictor to distinguish both branches. Note that the McFarling branch predictor could easily handle this case.

Data cache behaviour. Finally, we also measured data cache behaviour, see Figure 15. These results are comparable to the branch prediction results: the number of load and store operations decreases and the miss rate increases because the loads and stores that are removed, generally do not cause cache misses. As a result, the absolute number of cache misses decreases, see Figure 15 on the right.

6 Conclusion

In this paper, we have presented the optimization of a 3D image reconstruction algorithm. Prior to this research, this algorithm was not useful for real-time purposes. But thanks to the 12X speedup that was obtained on the Texas Instruments TMS320C6701,

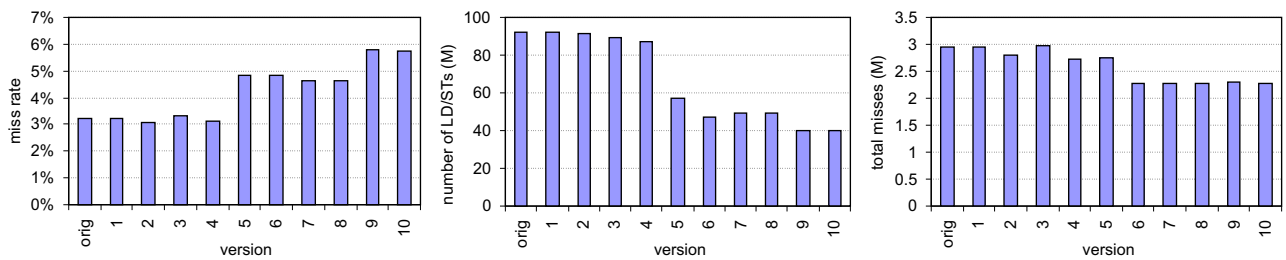


Figure 15: Cache miss behaviour for loads and stores for the Alpha gcc compiler version. Left graph: cache miss rate; middle graph: number of load/stores (in millions); right graph: number of load/store cache misses (in millions).

real-time applications of this algorithm become viable. The following optimization steps resulted in the biggest performance gains on the TMS320C6701: software pipelining, eliminating the norm calculation from the inner loops, the optimized address calculation for accessing the lookup table, converting floating-point into fixed-point calculus and the avoidance of NOPs by filling up branch delay slots. On the Alpha 21164, a 4X speedup was obtained. For the Alpha 21164, we have also evaluated the influence of the various optimizations on the performance of different architectural structures, such as the branch predictor and the memory hierarchy. From this analysis, we can conclude that one should be careful with performing optimizations since these optimizations may have an unexpected influence on the performance of various architectural structures.

Acknowledgments

Tom Vander Aa is supported by the FWO-project G.0036.99 and the TI Elite project. Lieven Eeckhout and Hans Vandierendonck are supported by a grant from the Flemish Institute for the Promotion of the Scientific-Technological Research in the Industry (IWT). Bart Goeman is supported by the FWO-project G.0036.99. Tanja Van Achteren is supported by the FWO-project G.0036.99 and the IUAP 4/24 project.

References

- [Proesmans et al., 1996] Proesmans, M., Van Gool, L., and Oosterlinck, A. (1996). One shot active 3D shape reconstruction. In *Proceedings 13th ICPR International Conference on Pattern Recognition: Applications & Robotic Systems*, volume II C, pages 336–340.
- [SPRU187E, 1998] Texas Instruments. TMS320C6x Optimizing C Compiler User’s Guide. Manual number SPRU187E.
- [Aho et al., 1986] Aho, A., Sethi, R., and Ullman, J. (1986). *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company.
- [Mahlke et al., 1995] Mahlke, S., Hank, R., McCormick, J., August, D., and Hwu, W. (1995). A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 138–149.
- [SPRU189D, 1999] Texas Instruments. TMS320C6000 CPU and instruction set reference guide. Manual number SPRU189D.
- [Alpha 21164, 1997] Digital Equipment Corporation. Digital Semiconductor 21164 Alpha Microprocessor Hardware Reference Manual.
- [Srivastava et al., 1994] Srivastava, A., and Eustace, A. (1994). ATOM: a system for building customized program analysis tools. Western Research Lab Technical Report WRL 94/2.
- [SPRA486B, 1999] Texas Instruments. TMS320C6000 power consumption summary. Manual number SPRA486B.
- [McFarling, 1993] McFarling, S. (1993). Combining branch predictors. Western Research Lab Technical Report WRL TN-36.
- [Talla et al., 2000] Talla, D., John, L., Lapinskii, V. and Evans, B. (2000). Evaluating signal processing and multimedia applications on SIMD, VLIW and superscalar architectures. In *Proceedings of the IEEE International Conference on Computer Design ICCD-2000*, pages 171–182.
- [Foley et al., 1995] Foley, J., van Dam, A., Feiner, S., and Hughes, J. (1995). *Computer graphics, principles and practice*. Addison Wesley, second edition.
- [Code Composer, 1999] Texas Instruments (199). Code Composer Studio White Paper.
- [SPRU198C, 1998] Texas Instruments. TMS320C62x/C67x programmer’s guide. Manual number SPRU198C.