On the Use of Subword Parallelism in Medical Image Processing

Bjorn De Sutter, Mark Christiaens, Koen De Bosschere, Jan Van Campenhout

Department of Electronics and Information Systems, University of Gent, B-9000 Gent, Belgium

Abstract

Parallel implementations of algorithms for medical image processing mostly focus on the use of multiprocessor parallelism. Modern processor architectures however, provide several additional forms of parallelism at the processor level: subword parallelism, speculative execution, superscalar pipelining, very long instruction word, etc. In this article, we show that well-known parallelization techniques for multiprocessor systems can be used to exploit subword parallelism. Loop unrolling, loop fusion and `if`-hoisting prove to be valuable to achieve this goal. To illustrate this, we transformed the inner loops of a positron emission tomography image reconstruction algorithm. We achieved a speed-up of 45% on Sun's UltraSPARC processor.

Key words: subword parallelism, loop transformations, positron emission tomography

1 Introduction

Parallel and distributed processing has been applied to almost every compute-intensive task to speed up its execution. To that end, libraries (DSM [27,28], PVM [4], MPI [2,3], etc.) and programming environments [9] (FPT [13], SUIF [15], Polaris [10], Parafrase-2 [25], etc.) have been developed. The long-term goal of all this work is eventually to come to the fully automatic parallelization of code and to make optimal use of the available computing resources.

With the advent of sophisticated superscalar processor architectures, multiprocessor parallelism is however not the only technique available to speed up an application: cache behavior, branch prediction, instruction scheduling, and the ability to make use of the more powerful instructions have a nontrivial impact on the global performance of an application. They all aim at executing more instructions per cycle (IPC), by making better use of the available hardware. A speed-up of a factor 2-3 is not unusual when combining all these techniques [30].

Recently, several general purpose processors have been enriched with so-called multimedia extensions (Intel MMX [1], Hewlett Packard's MAX-II [17] extensions, Sun's VISual Instruction Set [8], MIPS's MDMX [32]). These extensions are characterized by two features:

(1) besides modulo arithmetic, they also support saturation arithmetic which is more convenient when dealing with overflow in signal and image processing,
(2) they allow to simultaneously operate on several small data items by first packing them into one machine word and by applying specialized instructions to them (e.g. four simultaneous 16-bit additions instead of one 64-bit addition).

This latter feature is called *subword parallelism*, which allows to make better use of the 64-bit machine words of modern architectures for applications that typically deal with small data items (pixel, samples, etc.). Most transformational instructions can easily be generalized to take the subwords into account.

Besides the evolutions in the general purpose processors, there is also an increased interest in VLIW-architectures [31] where one instruction consists of several (independent) elementary operations that are all executed in parallel within certain restrictions. Here too, optimally scheduling the operations over the instruction may yield important speed-ups. A technique that proves very useful is predication which allows to conditionally execute predicated operations.

State of the art compilers already do a good job at optimizing the instruction scheduling and at improving the branch prediction. Some also take the cache behavior into account. With respect to the exploitation of the multimedia extensions such as subword parallelism, there is yet a long way to go. At the time of this writing, most compilers do not even generate the novel instructions for saturation or subword arithmetic. Most applications that use the multimedia extensions are hand-coded assembly routines for fairly simple multimedia processing algorithms. The main cause why compilers do not exploit the power of these new instructions is that it is far from trivial to detect opportunities in the source program where e.g., subword parallelism could be exploited.

In this paper, we show that traditional program transformations such as loop unfolding, code hoisting, and loop merging can be used to create opportunities for exploiting subword parallelism. We claim that the suite of transformations we propose is not limited to simple and regular multimedia processing algorithms but that it is also applicable in more complex medical image processing algorithms, provided that they have at least one loop with independent iterations. That loop is then used as starting point in the search for calculations that can be executed in subword parallel. We believe that subword parallelism is a viable new form of parallel processing that can be detected automatically by the compiler, and that certainly deserves more attention in the future.

In section 2 we present the individual code transformations that are used. Herein we deal with loops consisting of one basic block, loops containing `if`-tests, and loops containing other loops, so that we can basically process any well-structured program. In section 3, we manually apply the suite of code transformations to a nontrivial loop that calculates the radiological path in a medical image reconstruction application. Section 4 reveals some implementation aspects of our work, and section 5 presents the speed-up that results from the subword parallelism.

2 Loop Transformations for Uncovering Subword Parallelism

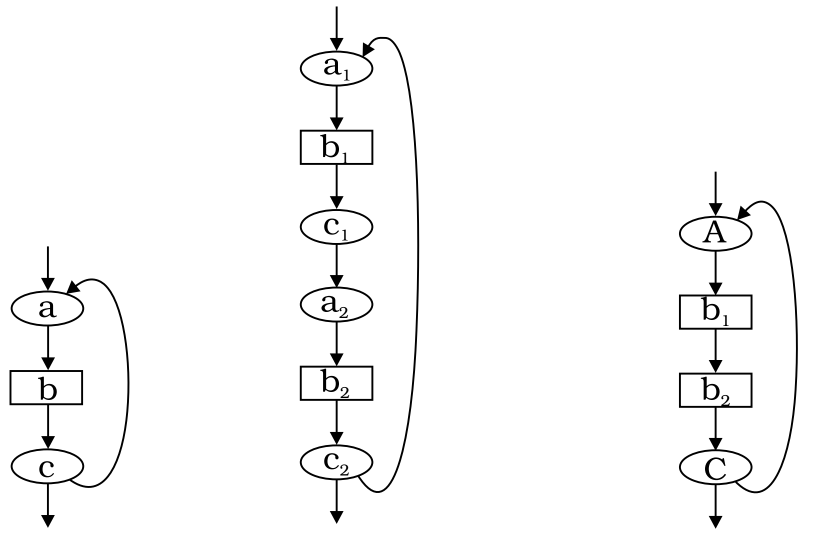
With subword parallelism, several identical operations on different data are grouped into one instruction. It is a specific form of SIMD (Single Instruction Multiple Data). In order to fully exploit this technique, basic blocks need to contain enough identical operations. In what follows we will propose how combinations of several known techniques can transform general loops in loops suited for the exploitation of subword parallelism.

An important reason to work on loops is that, although some of the transformations have a wider applicability, the conditions that must hold to apply the transformations are more easily verified in loops. This makes the proposed techniques possible candidates for integration in future compilers.

2.1 Loop Unrolling and Software Pipelining

For very simple loops, loop unrolling and software pipelining are techniques suitable for the creation of loop bodies with identical operations on different data. If the original loop consists of independent iterations, then the instructions from the original iterations are independent in the resulting loop body as well.

For more complex loops, loop unrolling and software pipelining are also the



(a) initial loop (b) after loop unrolling (c) after software pipelining

Fig. 1. Control flow graphs after consecutive simple code transformations

first transformations one needs to apply in order to eventually create the desired basic blocks.

Starting from an initial loop, as depicted in Figure 1a, loop unrolling can yield a flow graph as shown in Figure 1b. Basic block a is the entry block of the loop, while c is the exit block. The convention in the control flow graphs (CFG) that we use, is that rectangles denote not yet identified constructs or regions (loops, `if-then-else`-constructs, etc.), while ellipses stand for (possibly empty) basic blocks. Note that we have unrolled the loop once, for the clarity of the figures, but more may be desirable.

If the iterations in the original loop were independent, the basic blocks a_2, b_2 and c_2 operate on data independent from the data operated on in basic blocks a_1, b_1 and c_1. Throughout this article, as a convention, regions or basic blocks with different indices operate on independent data. Basic blocks with the same name, but different indices, perform the same operations on independent data.

Because of the mentioned independence in the graph in Figure 1b, software pipelining the loop body is possible. The result of this transformation is shown in Figure 1c. Basic block A is the result of merging basic blocks a_1 and a_2, and C is the result of merging c_1 and c_2.

For the rest of this article, it is important to note that, while the original loop was assumed to have independent iterations, this assumption is no longer needed in the unrolled version or for any loop introduced later in this section.

Now, for the simplest case, where b is a simple basic block, the instructions in b_1 and b_2 can be grouped and possibly replaced by subword parallel instructions

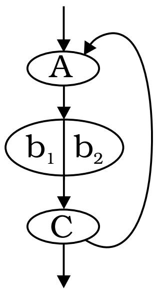
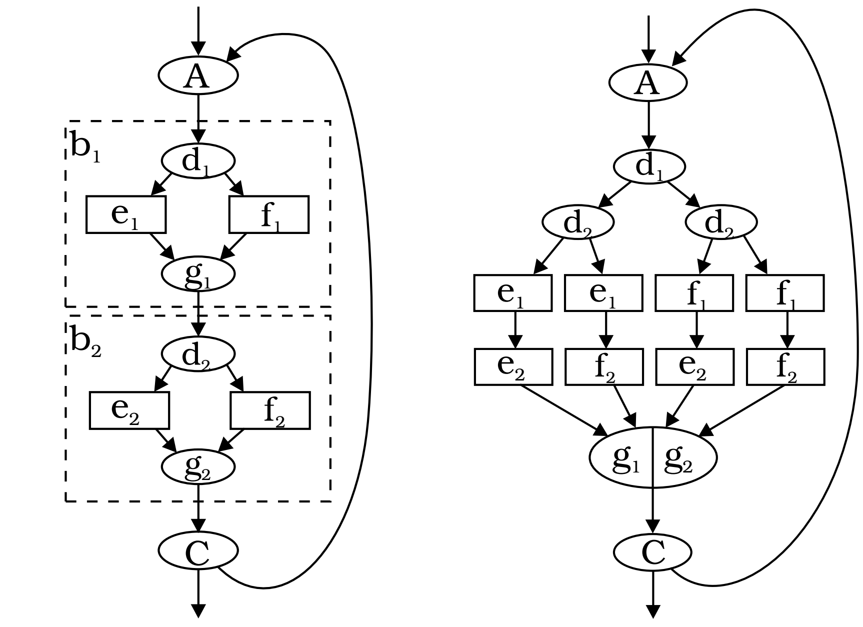


Fig. 2. The final loop after merging the inner basic blocks



(a) loop body with `if`-structure (b) after `if`-hoisting

Fig. 3. Loops with `if-then-else`-structures in the loop body

as in Figure 2.

2.2 If-hoisting

In more complex loops, with e.g. `if-then-else`-constructs in the loop body b, loop unrolling and software pipelining yields a CFG as shown in Figure 3a. No basic blocks containing identical operations exist if they were not already present in the original loop body.

But if we now hoist the lower `if`-tests, together with e_2 and f_2 (which results

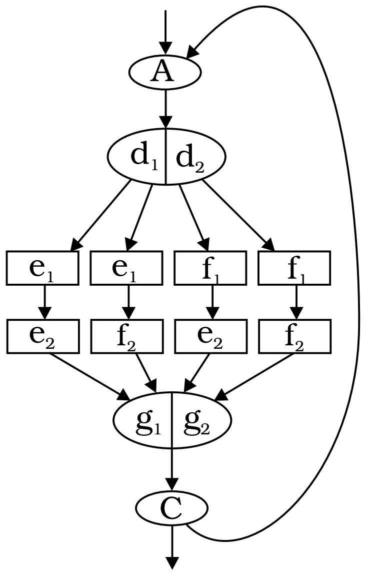
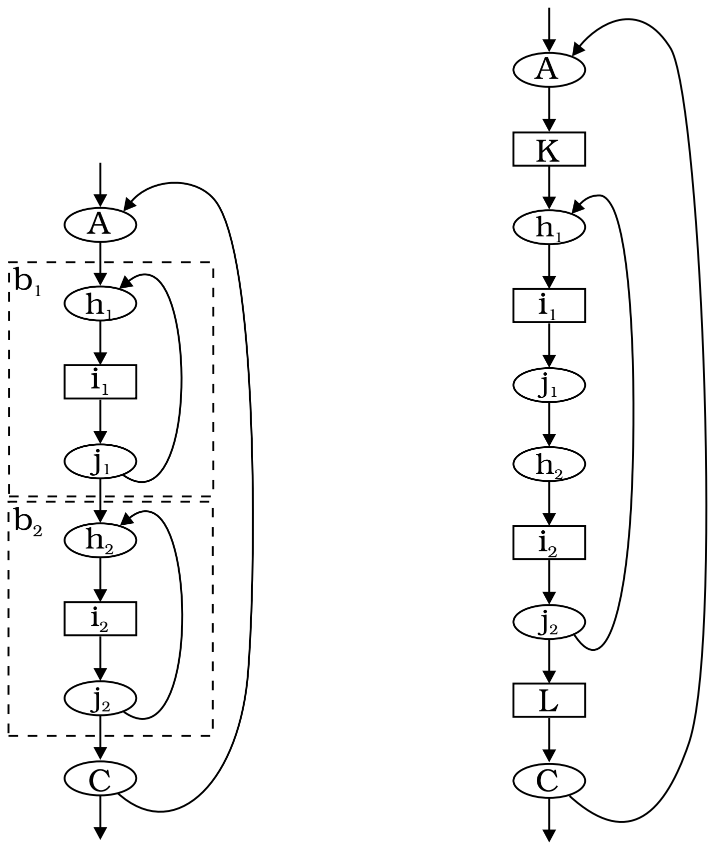


Fig. 4. A loop using a parallel comparison

in the graph of Figure 3b) then the regions e_1 and f_1 immediately precede e_2 and f_2. Then g_1 and g_2 are candidates to be executed in subword parallel. Furthermore, if on the one hand e and f are basic blocks, e_1 and e_2, and f_1 and f_2 can be executed in subword parallel. In addition, if f and e have common operations, even e_1 and f_2, and f_1 and e_2 can be executed in subword parallel.

If on the other hand, e and f are not basic blocks but do have identical constructs and operations, a combination of all the transformations in this section can be applied on them and might still uncover subword parallelism. (The two consecutive nodes on every of the four paths are then in the same situation as b_1 and b_2 in Figure 1c and the same transformations can be applied here as well.)

Another way to speed up the execution of a loop as shown in Figure 3b using subword parallelism is by optimizing the `if`-tests tree. Most multimedia-extended architectures provide parallel subword comparisons. These comparisons always generate a bit pattern. This pattern can easily be transformed into a bit pattern fit to index a jump-table. Replacing the consecutive tests on every path by one parallel test and a look-up in the jump table results in the flow graph shown in Figure 4. As can be expected, this optimization gives a boost in performance.



(a) nested loop (b) after loop fusion

Fig. 5. Nested loops

2.3 Loop Fusion

Another possible structure for the region b in Figure 1a is a loop. It does not matter for our purpose whether this inner loop has independent iterations or not. The more general graph of Figure 1c can now be refined to the one depicted in Figure 5a.

All operations in b_1 are independent of those in b_2 since the original outer loop had independent iterations. Thus, loop fusion of the two inner loops can be applied. The resulting graph is shown in Figure 5b. Because the number of iterations of the loops b_1 and b_2 may differ, it might be necessary that some iterations are executed separately. These can be put in regions K and L. The inner loop of this graph is now identical to the loop of Figure 1b and all illustrated transformations can again be applied to this inner loop.

Note that loop fusion can be advantageous even when subword parallelism is not used. In some algorithms the cache-behavior is greatly improved if the two fused loops load data in each others vicinity. An example is given in section 3.

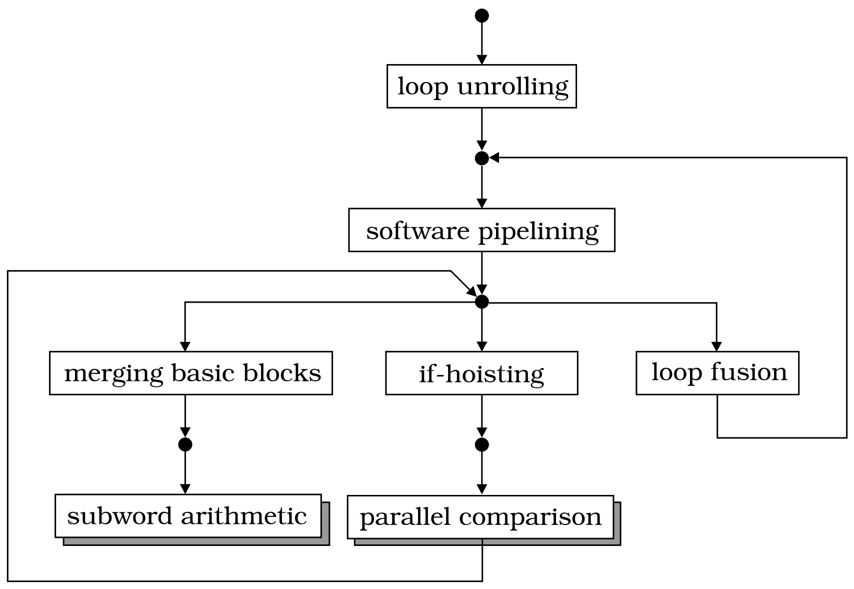


Fig. 6. A flowchart of the possible loop transformations that might result in exploiting subword parallelism

2.4 Combining Transformations

Figure 6 shows how combinations of the proposed transformations can eventually lead to basic blocks containing the desired instructions to use subword parallelism.

The transformations we have shown are more generally applicable. Consider for example a loop body consisting of an `if-then-else` followed by a nested loop. After loop unrolling, the two or more `if-then-else`-structures in the new loop body can be shifted upwards, ready for `if`-hoisting. The remaining loops are grouped in the lower part of the body and are candidates for loop fusion.

To conclude the section about general code transformations, notice that the goal of these transformations, namely the creation of basic blocks with independent but identical operations, is not only useful for exploiting subword parallelism. These transformations can also lead to better scheduling opportunities for the compiler in general.

3 A Case Study: The Calculation of Radiological Paths

We have shown that loops with independent iterations show much potential for the exploitation of subword parallelism. The possible examples are numerous: matrix operations, signal filtering, image reconstruction, etc. It is precisely for this kind of applications that multimedia extensions were added to general-purpose architectures: to achieve a speed-up in the execution of the rather simple inner loops of multimedia applications.

In this paper however, we want to prove that applications with inner loops of a more complex structure will also benefit from multimedia extensions. To fully illustrate the possibilities to exploit subword parallelism for more complex loops, we shall delve a little deeper into a particular medical imaging algorithm: the calculation of radiological paths and its implementation on an UltraSPARC II [7]. This processor is an implementation of the SPARCv9 [5] architecture, extended with a set of multimedia-oriented instructions: the VISual Instruction Set or VIS [8].

3.1 The Calculation of Radiological Paths

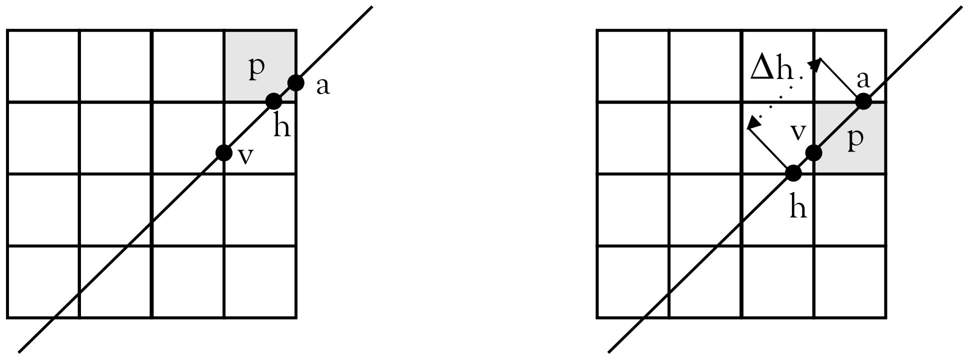
A radiological path (RP) of a line through an image is defined as a weighted sum of pixel values. The weighting factors are the distances the line traverses through the pixels.

By itself, this calculation is not very time consuming since most images have a maximum resolution of 512×512. For positron emission tomography (PET) image reconstruction however, the RP is calculated for sets of hundreds of parallel lines under varying angles. This results in routines that are repeated millions of times during one image reconstruction, making them a suitable candidate for optimization.

The fastest method known to us to calculate a RP is the incremental algorithm. It is inspired by Siddon's algorithm [20] that works on a parametrized representation of the line. For the sake of brevity, we will explain in this article only the bare minimum necessary to understand the applicability of subword parallelism to this problem. For an in-depth description see [12].

The line is represented by a parameter varying linearly (see Figure 7a). First, we determine a, the parameter value of the entry point of the line into the image and the corresponding entry pixel p. Next, the first intersections of the line with horizontal and vertical pixel boundaries inside the image are calculated (with resp. parameter values h and v). This provides us with a basis to start calculating the RP.

Suppose, that the pixel boundary through which we leave p is a horizontal one. This can easily be detected by noticing that $h \leq v$. The measure for the length of the line segment in p is $l = h - a$. We obtain the contribution of pixel p to the RP by multiplying l with the density of p.



(a) entering the image (b) one vertical step

Fig. 7. The parameter values during traversal of the image.

To advance to the next pixel, we adjust the values of a, h, v and p (see Figure 7b). The entry point into the next pixel is easily determined: it is the intersection with the horizontal line of which we already know the parameter value h. So we give a the value of h. The value of v need not change; the next intersection with a vertical line is still the same. We do, however, need to change the value of h. The new value of h becomes $h + \Delta h$, where Δh is the constant distance between two consecutive intersections with horizontal boundaries. Finally, after a simple integer addition, the new p points to the pixel under the previous one.

The calculations, as described in the previous paragraphs, are repeated until the line leaves the image. The total sum of all the contributions of the pixels forms the RP of the line.

All this makes for a fairly simple iteration scheme. It has a few shortcomings however. First, for every pixel, we need to determine whether the line crossed a horizontal or vertical boundary. This results in a loop with an `if-then-else`-construct inside the inner loop producing code with small basic blocks. Second, every iteration is dependent on the previous one, making parallel execution nontrivial.

In what follows, we will apply our general transformations to this code and show it effectively eliminates the shortcomings in this code.

3.2 The Transformations Applied

Our algorithm for the calculation of the RP in pseudo-code is presented in Figure 8. On the right side, we have annotated the code according to the labels used in the transformations in section 2.

Since during the calculation of the RP, all data is written to separate memory

```
for (each angle $\phi$)  // loop 1
    for (each line $l$ under angle $\phi$)  // loop 2
        initialize line $l$;                                        a
        for (each pixel in line $l$)  // loop 3                     }
            if ($h < v$)                                  d  }      }
                $RP \mathrel{+}= [h - a] * image[p]$;  }     }      }
                $a = h$;                               } e   }      }
                $h \mathrel{+}= \Delta h$;             }     }      }
                $p \mathrel{+}= \Delta y$;             }     } i    } b
            else                                             }      }
                $RP \mathrel{+}= [v - a] * image[p]$;  }     }      }
                $a = v$;                               } f   }      }
                $v \mathrel{+}= \Delta v$;             }     }      }
                $p \mathrel{+}= \Delta x$;             }     }      }
        store RP;                                                   c
```

Fig. 8. Pseudo-code for part of an ML-EM image reconstruction

```
for (each angle $\phi$)  // loop 1
    for (each lines $l_1$, $l_2$, $l_3$, $l_4$ under angle $\phi$)  // loop 2
```

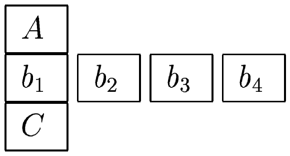


Fig. 9. Pseudo-code after loop unrolling and software pipelining

locations, there is no dependency between the iterations of loop 2. This loop is therefore the starting point of our consecutive transformations.

The first transformations we always apply are loop unrolling and software pipelining. Since we are trying to implement this algorithm on an UltraSPARC which is able to perform 4-way subword parallel operations, we will unroll the loop 4 times. This is possible because the 16 bit subwords of a 64 bit register have a sufficiently large range and precision. The result of these transformations can be seen in Figure 9, which corresponds to Figure 1c. All initializations have been absorbed by code region A and all stores by region C.

Next, we notice that b consists of a loop. Therefore, we can perform our third transformation, loop fusion. The result is shown in Figure 10. This figure corresponds to Figure 5b where $h_{\{1,2\}}$ and $j_{\{1,2\}}$ are empty. The number of iterations (m) in the fused loop is that of the shortest original loop. It is calculated in K. Region L consists of code for the residual part of each of the four loops.

```
for (each angle $\phi$)   // loop 1
    for (each lines $l_1$, $l_2$, $l_3$, $l_4$ under angle $\phi$)   // loop 2
        $\boxed{A}$
        $\boxed{K}$
        for (for the first $m$ pixels of $l_1$, $l_2$, $l_3$, $l_4$ )   // loop 3 fused
            $\boxed{b'_1 = i_1}$ $\boxed{b'_2 = i_2}$ $\boxed{b'_3 = i_3}$ $\boxed{b'_4 = i_4}$
        $\boxed{L}$
        $\boxed{C}$
```

Fig. 10. Pseudo-code after loop fusion

```
for (the first $m$ pixels of $l_1$, $l_2$, $l_3$, $l_4$ )   // loop 3 fused
    if $\boxed{d_1}$ then if $\boxed{d_2}$ then if $\boxed{d_3}$ then if $\boxed{d_4}$ then
                                                      $\boxed{e_1}$ $\boxed{e_2}$ $\boxed{e_3}$ $\boxed{e_4}$
                                                  else
                                                      $\boxed{e_1}$ $\boxed{e_2}$ $\boxed{e_3}$ $\boxed{f_4}$
                                   else if $\boxed{d_4}$ then
                                                      $\boxed{e_1}$ $\boxed{e_2}$ $\boxed{f_3}$ $\boxed{e_4}$
                                                  else
                                                      $\boxed{e_1}$ $\boxed{e_2}$ $\boxed{f_3}$ $\boxed{f_4}$
                                                  ⋮
```

Fig. 11. Pseudo-code after `if`-hoisting

The innermost loop, thus created, corresponds to the situation depicted in Figure 1c. So we can start the whole transformation process over again using this loop, where i is renamed to b'.

From this point forward, we will show only the transformations on the innermost loop. The code regions b'_1 through b'_4 consist of identical, independent `if-then-else`-structures of which the `then` and `else`-part perform exactly the same operations on different data. After performing `if`-hoisting, the result is shown in Figure 11.

Since the tests are simple comparisons, they can be performed in parallel. The result on the UltraSPARC architecture is a 4 bit bitfield (one bit for each comparison), which can be used as an index in the jump table. This way, after 1 table look-up, we can jump directly to the correct combination of `then` and `else` parts. The result is shown in Figure 12.

It is clear that, since the e_i and f_i basic blocks perform identical operations (multiplication, addition, subtraction and assignment) on independent data, they can be parallelized using subword parallelism. We will go in a little more

```
for (the first $m$ pixels of $l_1$, $l_2$, $l_3$, $l_4$ )   // loop 3 fused
  bitfield = par_comp ( $\boxed{d_1 \mid d_2 \mid d_3 \mid d_4}$ )
  goto jump_table[bitfield]
  jump_table[0]:   $\boxed{e_1}\ \boxed{e_2}\ \boxed{e_3}\ \boxed{e_4}$
  jump_table[1]:   $\boxed{e_1}\ \boxed{e_2}\ \boxed{e_3}\ \boxed{f_4}$
  ⋮
  jump_table[15]:  $\boxed{f_1}\ \boxed{f_2}\ \boxed{f_3}\ \boxed{f_4}$
```

Fig. 12. Pseudo-code with parallel subword comparison

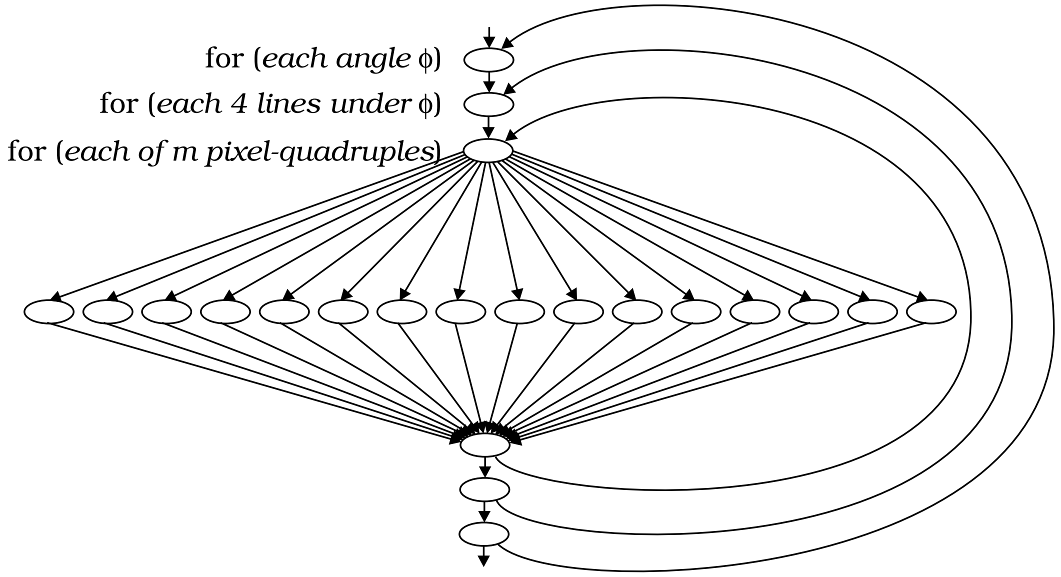
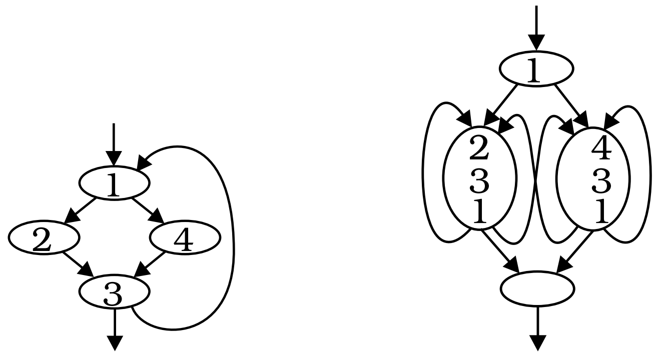


Fig. 13. The flow graph of the incremental algorithm after code transformations

detail on this in the next paragraph. The final flow graph, after all transformations have been applied, is depicted in Figure 13.

This is the simplest visualization of the CFG. It is clear that the edges converging in the exit block of the inner loop and the back edge can be avoided by appending the entry and the exit blocks to each of the 16 calculation blocks. From the end of each block, we then jump directly to the appropriate block for the next iteration using the jump table (see Figure 14 for a CFG with 2 paths). The jump table is extended to contain an extra address, the target address of the exit jump. A conditional move can set the index in the table to point to this extra address in order to exit the loop.



(a) original loop body with 2 paths (b) transformed loop

Fig. 14. Reducing the number of branches in the inner loop body

3.3 Exploiting Subword Parallelism

In this paragraph we will show how to use subword parallel instructions for the calculations in the e and f blocks.

The conversion to subword parallel operations implies more than the replacement of several identical instructions by a parallel one. The operands of the original instructions have to be packed in words and become subwords before one can operate on them in parallel. Three steps are necessary in general. For the sake of clarity, we apply them to our example algorithm.

Packing data in registers In our algorithm, a-values, h-values and v-values for each of the 4 lines are grouped in $\langle a\rangle$, $\langle h\rangle$ and $\langle v\rangle$. This grouping is fixed for the whole execution of the loop.

Loading data elements In each iteration, 4 new pixel values are loaded. These are loaded separately, each in a different register. We need to pack them into one register in each iteration. This is the main difference between our type of algorithm and typical imaging or multimedia applications. In those applications, operations on data streams run through the data in a very regular form in which adjacent pixels can be loaded in parallel and can be operated upon immediately. We need more preparatory work to pack the data elements in registers.

Performing operations In every iteration some of the elements of $\langle h\rangle$ are updated (`then`-part in the original program), as well as other elements in $\langle v\rangle$ (`else`-part in the original program). Operations on some elements in $\langle h\rangle$ can be performed using masks.

For example, an addition to the first and third subword of $\langle h\rangle$ is done by adding $\langle \Delta h\rangle$ to it, whereas the second and fourth element of $\langle \Delta h\rangle$ are set to zero by a mask. The use of masks is straightforward, but to be efficient, enough registers have to be available. In our program, 16 different masks are used, which are best placed in registers permanently.

Summarizing, we can say that variables used in the loop have to be packed, masks have to be used to perform the correct operations and pixels have to be loaded and merged in registers.

It is important to note that the calculation of pixel indices is not done using subword parallelism. These calculations are done using integer arithmetic in the first place, and thus are executed in parallel with the subword parallel operations on superscalar architectures. Besides that, for large images, the range needed for the indices would be too large to fit in subwords anyway.

3.4 Possible Further Optimizations

Each of the 16 paths in the inner loop requires the application of masks to data. If we take a closer look, we can wonder whether this is really necessary.

If the next iteration passes along the same path as the previous one, some masked operands are still available from that previous iteration. This is the case for $\langle \Delta h \rangle$ and $\langle \Delta v \rangle$.

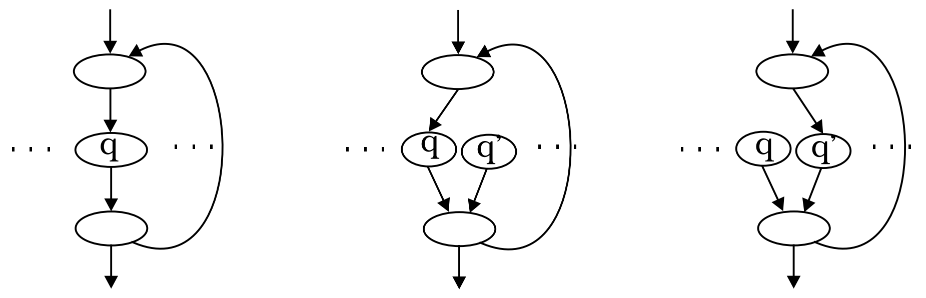
This kind of partial redundant calculations will almost certainly occur when `if`-hoisting is applied. In some cases, this can be optimized as depicted in Figure 15. The left graph shows one of the 16 different paths in the inner loop of our algorithm.

First we duplicate the calculation block q. The result is shown in figure 15b. From the copy q', we delete the unnecessary instructions for the case when the previous iteration passed along this same path. Next, we insert two instructions in the original basic block q that set the jump-table pointer to a new table, in which the address of q is replaced by that of q'. These two instructions present no overhead, since they are executed in parallel with the subword parallel instructions.

This way, as long as the same path is chosen, the loop will iterate using the faster basic blocks q' (the control flow graph as depicted in Figure 15c) and only when a new path is chosen will the initializing instructions for that path be executed (the control flow graph for that path looks like Figure 15b).

Throughout the program, all addresses in the jump table but one point to the original calculation blocks. Only the address of the block we have just executed is replaced with that of the corresponding faster block.

This optimization will definitely not hold for some algorithms. In our case however, one third of the iterations passes along the same path as the previous iteration. This makes the effort of optimizing the code as illustrated



(a) one original path (b) first pass through path (c) repeated pass through path

Fig. 15. Inner loop paths after `if`-hoisting with duplicated basic blocks

viable. In general, we note that after `if`-hoisting, a multitude of similar calculation blocks is created. It is our opinion that it will often be useful to try to recuperate some intermediate data from one block to reuse it in the following blocks. This requires tracing the program to estimate the dynamic probability that certain branches are taken.

4 Programming Subword Parallelism

We have shown that to fully exploit subword parallelism, nontrivial code transformations are necessary. The programmer can use all the help he/she can get to simplify this process. During our research on the UltraSPARC, we tried several tools to automate as much as possible the programming task.

4.1 Automatic Compilation

In an ideal world, one would provide a specification of a problem without any regard for the underlying architecture. Then the compiler would do extensive transformations on the specification and eventually produce excellent code. Sadly, at the time of writing, this is not the case; the compiler cannot even produce the new VIS instructions that implement subword parallelism on the UltraSPARC. Still, it is interesting to wonder what the minimal adjustments to a compiler/programming language should be to make the use of subword parallelism possible.

First of all, the compiler must be capable of generating the whole of the instruction set, including the instructions implementing subword parallelism. This implies that it has knowledge of the latencies of these instructions, their path through the processor pipeline, etc. This is well within the reach of modern compilers.

But how could a compiler decide to *use* such new instructions? It might try to extract by itself the necessary information from the source code. In a language like C, it would probably find little opportunity for optimizations, since almost none of the data types fit into subwords (only the `char` type). The compiler would have to decide whether the ranges of other variables with data types like `float`, `integer`, ... would fit into 8 or 16 bits. This implies an understanding of the semantics of programs beyond the capability of current compiler technology.

We must therefore explore other options. A first one is the addition of compiler directives. These would allow the programmer to express the fact that, although a variable has a larger range, its values would fit into 8 or 16 bits. This opens the way for more compiler optimizations. Yet another step further is the alteration of the language itself by adding new data types like an array of four 16-bit values or eight 8-bit values. Then operations like addition, comparison, etc. on these new data types can be defined. This way, the programmer is able to directly express his algorithms exploiting the new hardware to the fullest. Both these options have a major drawback: they endanger portability, albeit possible to make compiler directives transparent.

The loop transformations that we have put forward are used in vectorization compilers. These are mostly Fortran compilers. It is not trivial to port these compilers to languages like C, where pointers and other constructs complicate data analysis.

4.2 Templates

The compiler, not being able to exploit the VISual Instruction Set of the UltraSPARC, did present another option: inline assembler templates. This is a technique whereby pieces of assembly code are wrapped into a C-function. This C-function can then be called from a program, allowing the inclusion of specific assembly routines. The idea is then to inline these C-routines in the rest of the code, effectively removing the function call and only leaving the assembly body.

Sun provides a library containing one wrapper function for every VIS-assembly instruction. Although it seems an efficient technique to exploit subword parallelism, it totally confused the data analysis phase of the compiler. This resulted in extremely bad code in which tremendous amounts of time were spent superfluously loading and storing the content of registers. Rice's report [30] on using VIS has shown that for simpler algorithms, inline templates are a viable option. It is clear that using these templates complicates portability.

4.3 Libraries

A third option to exploit subword parallelism are preprogrammed libraries. Although this is a very interesting method for most programmers, since it presents them with highly optimized, out-of-the-box code, it does require the existence of library routines that can be used. Our problem was far too specific for any useful libraries to exist. At the moment of writing, only libraries for typical multimedia applications exist.

4.4 Hand Coded Assembly Language

Eventually, we were forced to hand code our algorithms in assembly. This is of course a very time-consuming, tedious and error-prone task. Its main advantage is full control of all the features present in a processor. Its disadvantage is its lack of portability between different processor families and even between processor generations. When a new version of a processor is released, its instruction latencies and pipeline behavior in general may have changed, warranting the manual rescheduling of the assembly code. Again a very time consuming task, best performed aided by a simulator of the CPU in question.

5 Results of Using Subword Parallelism

Having described how and by what means the incremental algorithm is implemented to exploit subword parallelism, we can now give results of these optimizations. In Figure 16 the execution times for a PET image reconstruction are plotted for various image sizes.

We embedded our routines in an existing 2D PET reconstruction program, *emvox2d*. This evaluation program is developed by the Medical Image Processing Group (MIPG), Department of Radiology, University of Pennsylvania, Philadelphia, USA. More advanced statistical PET reconstruction algorithms, such as RAM-LA, are faster, but use the same core routines for the calculation of the RP. Emvox2d uses the Maximum Likelihood-Expectation Maximization algorithm (ML-EM) [19], in which RP calculations take about 90% of the time. Our algorithm is linear in the sum of the dimensions of the image. Any nonlinearities in the graph result from cache-behavior.

The speed-up achieved by exploiting subword parallelism is 45% compared to our best C-code optimized using the SCO 4.0 compiler. This is not only due to the use of subword parallelism, but also to the better cache behavior: the

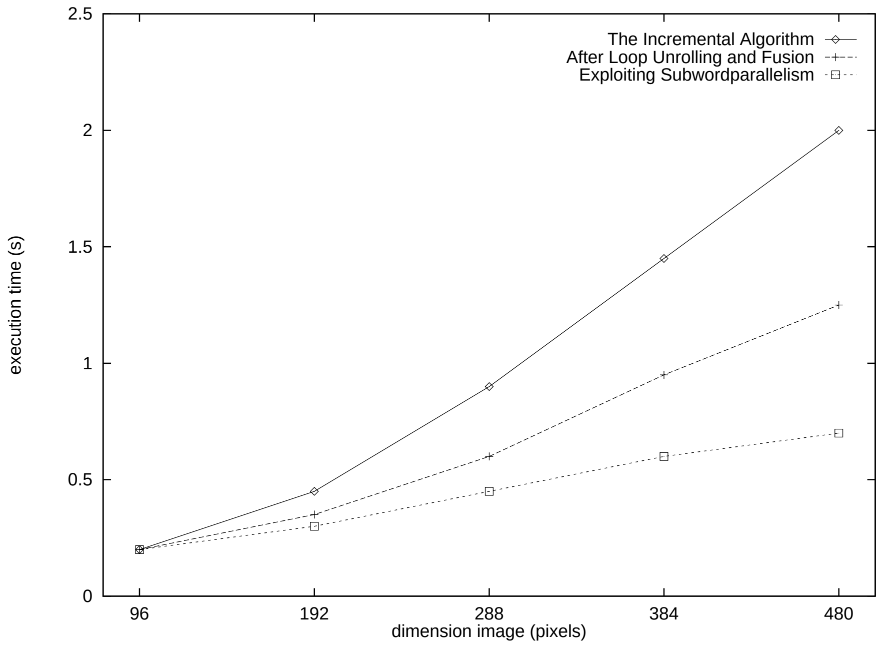


Fig. 16. Execution time of an ML-EM PET image reconstruction.

pixels are represented in a smaller data format and thus more pixels fit in the cache.

Finally, it is noteworthy to say that our optimizations are applicable to 3D PET image reconstruction as well.

6 Related Work

Rice has described in [30] how typical imaging and multimedia applications can benefit from multimedia extensions. These applications include image addition with clamping, blending using mask images, convolutions, resizing using bicubic interpolation, rescaling, conversion from YUV to RGB and Gouraud shading and texture mapping. These are applications with simple inner loop bodies.

In his report, he extensively shows how inline templates can be used and how VIS can be enhanced.

A new generation of true multimedia processors [14] (such as the Philips TriMedia [6], the Mpact Media Engine [33], the MicroUnity MediaProcessor [16] and NVidia NV-1 [34]) offer, in addition to subword parallelism, some new or reborn types of parallelism:

- very long instruction word (VLIW),
- hardware micro-threading, in which instructions from different threads are interleaved,
- special functional units that perform typical multimedia tasks, in parallel with the core processor activity,
- interfaces that prepare incoming data streams for efficient processing on the core processor,
- input and output buffers that replace DMA-channels (Direct Memory Access) to minimize the necessary memory bandwidth.

These processors are very efficient for processing multimedia. We have done some experiments on the TriMedia, which have learned us that its VLIW core processor is highly suited for the calculation of radiological paths. Because on the TriMedia basic blocks have to be large as well, since the delay slot contains 15 operations, we used the same code transformations, except for `if`-hoisting. Instead, we used guarding (all instructions can be executed conditionally) to merge the `then`-part and the `else`-part in the inner loop body into one basic block. The result of the `if`-test is then used as a guard or predicate.

Recently, much research is being done into the use of guarding and predicates in general to increase the number of instruction that can be executed simultaneously [24,11,23]. The transformations shown should increase the potential for exploiting these techniques.

It should be noted that the SPARC processor that we considered, includes a fairly simple branch prediction scheme (2-bit prediction scheme) and that 2 level branch history based prediction schemes [21,29] could possibly also speed up the execution of the program by better predicting the outcome of the original branches. We proposed a static technique that transforms a sequence of control flow instructions into one parallel branch so that only one branch misprediction can take place and as such only one branch penalty will be observed.

We also made use of the fact that if certain control paths are taken, some data values do not have to be recalculated. This topic too is actively being investigated by several groups around the world [22,26]. However they look at dynamic techniques in what they call data-prediction techniques.

7 Conclusions

In this article, we have shown that subword parallelism, as recently introduced in multimedia-oriented architectures, is beneficial for more general algorithms. More specifically, the calculation of the radiological path, a recurring routine in image reconstruction algorithms, executes 45% faster on an UltraSPARC when using the VISual Instruction Set.

The loop transformations we applied are standard techniques in multiprocessor parallelization and prove very effective for exploiting subword parallelism. Performing these transformations is a task that should be automated as has been done in parallelizing compilers. Further research in this direction should include the necessary data types and data analysis.

Acknowledgement

We would like to thank Ignace Lemahieu for his advice and support in our research. The colleagues at our laboratory proved a great support in creating the environment for our research. We also thank Henk Neefs for his comments on preliminary versions of this article.

References

[1] Intel Corporation. *Intel Architecture MMX Technology, Programmer's Reference Manual*, March 1996.

[2] University of Tennessee, Knoxville, Tennessee. *MPI: A Message-Passing Interface Standard*, June 1995.

[3] University of Tennessee, Knoxville, Tennessee. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderman. *PVM 3 user's guide and reference manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 1994.

[5] Inc. SPARC International. *The SPARC Architecture Manual, version 9.* PTR Prentice Hall, 113 Sylvan Avenue, Englewood Cliffs, New Jersey 07632, USA, 1994.

[6] TriMedia division Systems software group. *TriMedia databook, assembly language reference manual and programmer's reference manual.* Philips Semiconductors, 1.0 edition, November 1996.

[7] Sun Microsystems, Inc. Business. *UltraSPARC Programmer Reference Manual*, 1.1 edition, September 1995.

[8] Sun Microelectronics. *Visual Instruction Set (VIS) User's Guide*, 1.0 edition, 1996 April.

[9] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[10] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–81, December 1996.

[11] P.P. Chang, N.J. Warter, S.A. Mahlke, W.Y. Chen, and W.W. Hwu. Three architectural models for compiler-controlled speculative execution. *IEEE Transactions on Computers*, 44(4):481–494, apr 1995.

[12] M. Christiaens, B. De Sutter, K. De Bosschere, J. Van Campenhout, and I. Lemahieu. A fast, cache-aware algorithm for the calculation of radiological paths exploiting subword parallelism. *Journal of Systems Architecture, Special Issue on Parallel Image Processing*, 1998. Accepted for publication.

[13] E.H. D'Hollander, F. Zhang, and Q. Wang. The fortran parallel transformer and its programming environment. *Information Science*, 1998. To appear.

[14] T.R. Halfhill and J. Montgomery. Chip fashion. *Byte*, pages 171–178, November 1995.

[15] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.-W. Liao, E. Bugnion, and M.S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Micro*, 29(12):84–89, December 1996.

[16] Greg Hansen. MicroUnity's MediaProcessor architecture. *IEEE Micro*, 16(4):34–41, August 1996.

[17] R.B. Lee. Subword parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, August 1996.

[18] H. Neefs, K. De Bosschere, and J. Van Campenhout. Exploitable levels of ILP in future processors. *Journal of Systems Architecture*, 1998. Accepted for publication.

[19] L.A. Shepp and Y. Vardi. Maximum likelihood reconstruction for emission tomography. *IEEE Transactions on Medical Imaging*, 1(2):113–122, October 1982.

[20] R.L. Siddon. Fast calculation of the exact radiological path for a three-dimensional CT array. *Medical Physics*, 12(2):252–255, March 1985.

[21] P. Chang, E. Hao, T. Yeh, and Y. Patt. Branch classification: A new mechanism for improving branch predictor performance. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 22–31. ACM SIGMICRO and IEEE Computer Society TC-MICRO, December 1994.

[22] H.M. Lipasti and J.P Shen. Exceeding the dataflow limit via value predicton. In *Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture*, volume 29, pages 226–237. ACM SIGMICRO and IEEE Computer Society TC-MICRO, 1996.

[23] S.A. Mahlke, R.E. Hank, R.A. Bringmann, J.C. Gyllenhaal, D.M. Gallagher, and W.W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 217–227. ACM SIGMICRO and IEEE Computer Society TC-MICRO, December 1994.

[24] H. Neefs, F. Habils, L. Eeckhout, K. De Bosschere, and J. Van Campenhout. An analysis of predication in the context of a fixed-length block structured instruction set architecture. Submitted to the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, 1998.

[25] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. The structure of Parafrase-2: An advanced parallelizing compiler for C and Fortran. In *Languages and Compilers for Parallel Computing*, pages 423–453, Cambridge, MA, USA, 1990. MIT Press.

[26] Y. Sazeides, Vassiliadis S., and J.E. Smith. The performance potential of data dependence speculation and collapsing. In *Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture*, volume 29, pages 238–247. ACM SIGMICRO and IEEE Computer Society TC-MICRO, 1996.

[27] B. Van Assche. DSM under Mach: A quantitative approach. In *Parallel Computing: State-of-the-Art and Perspectives*, pages 463–470, Gent, September 1996. Elsevier.

[28] B. Van Assche and E.H. D'Hollander. Message passing versus distributed shared memory. In *Symposium on Knowledge and Information Technology*, pages 61–68, Gent, September 23–24, 1996.

[29] C. Young, N. Gloy, and M.D. Smith. A comparative analysis of schemes for correlated branch prediction. In *The 22nd Annual International Symposium on Computer Architecture*, pages 276–286. ACM SIGARCH and IEEE Computer Society TCCA, ACM press, June 1995.

[30] D.S. Rice. High-performance image processing using special-purpose CPU instructions: The UltraSPARC visual instruction set. Technical report, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, March 1996. `http://cs-tr.cs.berkeley.edu`.

[31] An introduction to Very-Long-Instruction-Word (VLIW) computer architecture. Philips Semiconductors, 1996.

[32] MIPS extensions for digital media with 3D, intruction set architecture specification, March 1997. `http://www.mips.com`.

[33] MPact: A new level of concurrent performance, September 1996. `http://www.mpact.com`.

[34] NVidia product description, April 1997. `http://www.nvidia.com`.

About The Authors

Bjorn De Sutter was born in Gent, Flanders, in 1974. He graduated in computer science and engineering at the University of Gent in 1997. He is currently, as a Ph.D. student, researching whole-program optimization at link-time at the Electronics and Information Systems Department of the Faculty of Applied Sciences at the University of Gent.

Mark Christiaens was born in Tielt, Belgium, in 1974. He graduated in computer science and engineering at the University of Gent, Belgium, in 1997. As a Ph.D. student, he is currently researching the debugging of distributed programs at the Electronics and Information Systems Department of the Faculty of Applied Sciences at the University of Gent.

Koen De Bosschere was born in Oudenaarde, Belgium, in 1963. In 1986 and 1987 respectively, he graduated in electronic engineering and computer science at the University of Gent. In 1992, he obtained his doctoral degree at the same university. He is now research associate with the Fund for Scientific Research - Flanders, and lecturer at the ELIS Department of the University of Gent.

Jan Van Campenhout was born in Vilvoorde, Belgium, on August 9, 1949. He received a Degree in Electro-Mechanical Engineering from the University of Gent, in 1972; and the MSEE and Ph.D. Degrees from Stanford University, in 1975 and 1978, respectively. Prof. Van Campenhout teaches courses at the Faculty of Applied Sciences of the University of Gent and is head of the ELIS department.