Efficient data driven run-time code generation for Multimedia applications

K. Brifault, H-P. Charles

PR*i*SM Laboratory, University of Versailles
{kbrifa,hpc}@prism.uvsq.fr

Abstract. Knowledge of data values and invariants at run-time allows to generate better code in terms of efficiency, size and power consumption. These techniques are particularly effective in the case of intensively reused functions, such as graphic applications.
This article introduces a low-level compiler technique using parametric embedded sections to generate binary code at run-time. A minimalistic generator which creates code and allocates registers using the data input, allows to generate only the needed instructions. We have called this generator a "compilet". Graphic experiments done on Itanium 2, PowerPC and UltraSPARC platforms highlight the interest of our technique despite an unavoidable code generation overhead.
Our measurements pinpoint the dependency between overhead redeeming and function reuses, allowing to determine where it can improve speed in image processing, such as in convolution filters. We see, for example, that the comparison between the dynamic generated code made by our compilet and its C counterpart has shown a speed improvement as high as of 43% on the Itanium 2 architecture, 25% on UltraSPARC and 41% on PowerPC.
This technique proves to be particularly useful in the multimedia field, where the advantages of dynamic compilation have not been fully taken into account yet.

1 Introduction

There are multiple intermediate stages between the traditional compilation process and the execution [1] such as parsing, intermediate code generation, flow analysis, code optimization and generation, libraries linking (compile-time), data loading (load-time), execution (run-time) in which procedures are called (at call-time). Code performance can be improved using static compilation owing to different heuristics and other techniques such as strength reduction or loop unrolling [18]. However classical static compilation operating only at "compile time", and does not have a knowledge of data values and invariants.
New methods have been introduced with the appearance of virtual machines. Java [11], for instance, has split compiling into a two-step process, consisting of a translation phase and an execution phase. The first stage generates platform-independent bytecode, and the second one, at run-time, generates target code on demand. This is called the Just In Time (JIT) compiler principle [3]. Target-dependent code is generated only at run-time, using a complex piece of code linked to the Java virtual machine (JVM) [20]. Hence, it takes time to compile a method, especially when we want to apply any kind

of optimization, and when this compilation has to be done each time the application is run. Moreover, a good JIT is complex and takes up a considerable amount of space.

Another optimization method uses techniques to inline assembly code inside a C program. The gcc `asm` extension is an example of such. It allows to inline assembly instructions inside a source function. Another example is the `Altivec` extension for gcc which allows the use of multimedia instructions which many C compilers can not generate. But, in this case, developers must have an extended knowledge of every platform on which they program, and of their specific instructions, in order to optimize the code, as assembly is platform-dependent. This technique is frequently used in image processing as compilers do not usually have the capability to use graphical instructions without a link to a specific multimedia library.

In this article, our work is multimedia-orientated and we generate code at run-time via ccg, a toolkit which allows to take into account assembly codes and Instruction Level Parallelism[1]. As most information is only known at execution, classical compilers do not have the knowledge of data values or invariants and must use heuristics to optimize programs [16]. Our approach consists in using dynamic compilation which delays the choice of optimizations by generating code at run-time, where the knowledge of values and invariants can be exploited, yielding code that is often superior to statically optimized code. Generated code is often better as the knowledge of values, invariants, or data structure allows the reduction of the number of operations to be executed. However, dynamic compilation only complements static compilation, with traditional optimizations being essential for many reasons [15,12].

Dynamic compilation is not a new concept, but our approach consists in applying it efficiently to the multimedia field. Multimedia has become one of the main uses of personal computing systems [5]: it represents a challenging design target for the industry, because of complex processing, requiring increasingly efficient computers, while maintaining affordable cost to meet consumer demand [14]. We are working towards this goal by achieving more with a given computation power in spite of the unavoidable overhead of dynamic code generation. In graphic processing, many computations are intensively reused, hence the overhead might be redeemed. Our technique allows a speed improvement: this determines the point at which reuses will off-set the overhead. Moreover, it generates only instructions that will actually be used. Another advantage is that we use all the statements, even the graphical instructions, without resorting to any multimedia libraries. Today, many people talk about applying dynamic compilation to image processing, but implementations are scarce and seem uncontrolled. In our approach, some methodology is introduced, as programmers have to specify the code to be optimized at run-time via some sort of “template” of machine code [2]. That has an impact upon the program implementation, as programs using dynamic code generation must be targeted for each machine. This embedded code can be parametric or not, *e.g.* it is possible to put C expressions or C variables into assembly instructions. For instance, the embedded section can contain an assembly instruction in which the constant is a C constant or expression of the program.

[1] ILP: is a family of processor and compiler design techniques that speed up execution by causing individual machine operations, such as memory loads and stores, integer additions and floating point multiplications, to execute in parallel.

Two difficulties become apparent. The cost of this code generation at run-time can be high [9,8], and it increases with the complexity. One of the main challenges of dynamic compilation is then to achieve high quality dynamically generated code at low run-time cost, as the time to perform run-time compilation and optimization must be recovered before any benefit from dynamic compilation can be obtained. The other difficulty is that while static compilers make optimization decisions by themselves, it is an ideal which seems untangible when referring to dynamic compilation, even for today's state of the art compilers.
In section 2, we describe our experimental setup and the methodology used to create our "compilets"[2]. Next, in section 3, we present results pertaining to multimedia applications, on convolution filters and geometric transformations (detailed for three different target architectures: Itanium 2, PowerPC, and UltraSPARC). Then we show how those results bring our contribution to light and we conclude with several directions that will be explored in further research.

2 Experimental setup and Methodology

2.1 Hardware environment

Our experiments are made on UltraSPARC, PowerPC and Itanium 2 architectures. The UltraSPARC is a 1.2 GHz processor machine (with a sparcv9 floating point processor and 2GB of memory) which cache memory hierarchy is organized in three levels: the D_Cache of 64 KB, the I_Cache of 32 KB and the E_Cache of 8 MB. The PowerPC is a 800 MHz processor machine with 256 MB of memory. The cache memory hierarchy is organized in two levels: the L1 cache of 32 KB and L2 cache of 256KB. The Itanium 2 platform is a uniprocessor operating at 1.3 GHz with 2 GB of memory. Itanium 2 is an "in order" processor, that offers a wide degree of parallelism which the cache memory hierarchy is organized in three levels, the L1D level of 16 KB, the L2 level unified of 256 KB and the L3 level of 1.5 MB.

2.2 Software environment

One of the interests of our approach resides in that all of our experimentations are written in C language in which several segments of dynamic code are embedded. Easily written, our codes both use classical optimizations of static compilation and allow to incorporate some dynamic codes, taking into account and exploiting data values, thus reducing the number of operations to be executed.
The dynamic code sections are sequences of assembly language statements which can not be directly used by compilers. They must be first translated by ccg, a combination of preprocessor and run-time assembler for C and C++ programs created by Ian Piumarta [21,6] in 1999. Our technique operates in two phases. At compile time, ccg (Figure 1) receives source programs which contain both embedded dynamic code and C language sections but processes only dynamic parts. The preprocessor output is its C counterpart

[2] compilet: it is a minimal dynamic code generator which creates code and allocates registers using the data input allowing to generate only the needed instructions.

in which the dynamic code sections have been replaced with ANSI C, via platform specific C macros. These macros can be taken as input by a classical compiler to produce an executable program. At run-time, each time a function containing dynamic code sections is called, the corresponding binary code gets generated in accordance with specific C macros and data values.

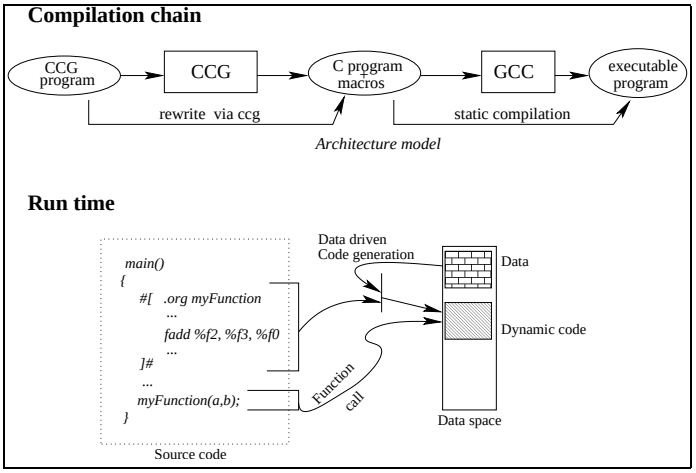

Fig. 1. Both sides of ccg: compile and run-time

Our experiments are made on PowerPC, UltraSPARC v9 and Itanium 2 platforms. For all of them, we designed macros, particularly in graphical or floating-point instruction set, which got implemented in ccg by its author, Ian Piumarta. Ccg is not platform-independent. A given program file generates a code that runs on one architecture only, as it includes assembly language statements. Portability of dynamic code generation is not the goal of ccg. It was developed to be an optimizing, dynamic code generator, supporting the lowest (platform-dependent) layer. We have chosen ccg mainly for its ease of implementation and modification. Moreover, ccg leaves to the client program total freedom in specializing all aspects of the generated code: literals, constants, registers and so on. This setup allowed us to verify the feasibility of our approach, providing us with a way to test various design choices and to bring to the fore the optimal setting of ccg to use in image processing.

The test machines were respectively running Linux IA-64 Debian version 2.6.3 based on the #1 SMP kernel for Itanium 2, MacOS 10.2.3 (based on Darwin Kernel version 6.8) on PowerPC and SunOS version 5.8 on UltraSPARC architecture. The compiler for all platforms is the 3.3 version of gcc. We gave ourselves some constraints in the choice of compilation options (set to *CFLAGS=-O7 -fno-inline -ffast-math* for all platforms. Other more specific information such as *-mcpu=970 -mtune=970* for PowerPC and *-mcpu=v9 -mtune=v9* for UltraSPARC are given; These options can not be activated on the Itanium 2 processor), because we have to compare equivalent codes be it static or dynamic, in order for our results to have a meaning, even if better dynamic performance could have been achieved with less specific options. For this same reason, the choice of instructions has been restricted to the same instructions than those proposed by a traditional static compiler (in the case of vector-matrix multiply).

2.3 Target codes

Convolution filter Here, we present a type of multimedia application, the convolution filter, implemented with our technique. Convolution is a simple mathematical operation which consists in applying a matrix to each image pixel, in order to create another image where pixels are a linear combination of their neighbors (Figure 2). For every pixel of an image, the convolution matrix, which has a small size, is not modified and can be easily stored in registers. Moreover, the instructions linked to the values of the convolution matrix remain the same during the whole processing, as these values do not change. In our algorithm, we break usual compilation rules by eliminating either the multiply operation if the filter value is equal to one, or both the multiply and the adding operation if the filter value is equal to zero. In the specific case of image processing, those deletions can be made. Indeed, all the components of pixels are defined in the 0 to 255 range. Hence, all operations are made between known constants (the values of the convolution matrix) and variables which are absolutely delimited. We can then consider that the removal of a multiply by zero is of no consequence.

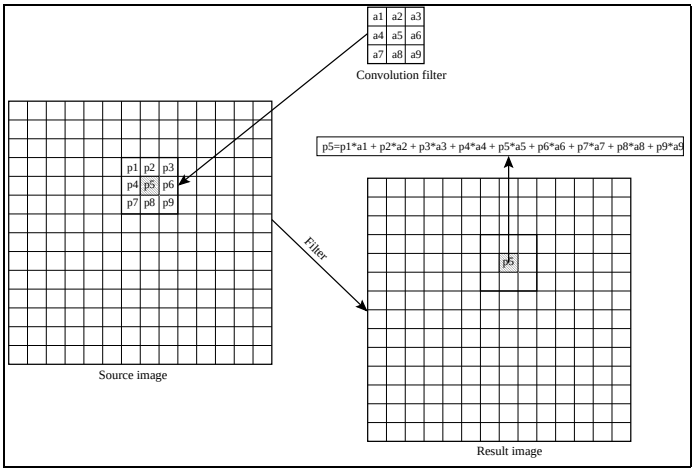

Fig. 2. Example of a small image and convolution filter

Our experiments are done with several convolution filters:

- Mean filter: A simple, intuitive and easy way to implement a method of smoothing images, *e.g.* reducing the amount of intensity variation between one pixel and the next. A method often used to reduce noise in images.
- Gaussian filter: A 2-D convolution operator used to remove detail and noise. In this sense, it is similar to the mean filter, but it uses a Gaussian distribution for matrix values.

Vector-Matrix multiply In the second experiment, we took an implementation from OpenGL Mesa-6.0, which contains unrolling and linearization:

```
d[0] = v[0] * m[ 0] + v[1] * m[ 1] + v[2] * m[ 2] + v[3] * m[ 3];
d[1] = v[0] * m[ 4] + v[1] * m[ 5] + v[2] * m[ 6] + v[3] * m[ 7];
d[2] = v[0] * m[ 8] + v[1] * m[ 9] + v[2] * m[10] + v[3] * m[11];
d[3] = v[0] * m[12] + v[1] * m[13] + v[2] * m[14] + v[3] * m[15];

where v represents the source vector, m the unidimensional array
which contains the matrix values and d, the destination vector.
```

To validate our approach, we have chosen to present four well-known matrices:

1. a full matrix ;
2. a geometric transformation matrix: $M_{t3D} = \begin{vmatrix} dx & \theta & x & 0 \\ \theta & dy & x & 0 \\ x & x & dz & 0 \\ tx & ty & tz & 1 \end{vmatrix}$
3. a scaling/translation matrix: $M_{scal/tl\theta} = \begin{vmatrix} dx & 0 & 0 & 0 \\ 0 & dy & 0 & 0 \\ 0 & 0 & dz & 0 \\ tx & ty & tz & 1 \end{vmatrix}$
4. a scaling matrix: $M_{scaling} = \begin{vmatrix} dx & 0 & 0 & 0 \\ 0 & dy & 0 & 0 \\ 0 & 0 & dz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$

In those experimentations, values of each matrix are variable in accordance with the selected structure. For instance, in a full matrix, neither zero nor one will be accepted. By structure, a lot of different matrices are tested. Results are given in CPU cycles and are an average of thousands of experimental measures.

2.4 Implementation of our compilets

Convolution Filter For the convolution filter, we have used the assembly graphical instructions which allow to process data values as vector or pixel composed by three components and not as an integer or a float using saturated arithmetics. The use of this multimedia instruction allows to reduce the number of assembly instructions and the size of the generated code.

We also took interest in the size of convolution filter. On the different platforms, the number of registers available is variable. For instance, on the IPF architecture, 96 integer registers are at the service of our convolution filter. But, on UltraSPARC, the number of registers is only a third of what it is on IPF. Taking into account this condition, when the size of convolution filter is not too large, the matrix can be stored in register thus allowing to earn back the data loading time. Hence, we have two possibilities during the creation of our compilet: the storing in register or the loading of matrix values. This choice depends on the size of the convolution filter and on the number of registers in the target architecture.

Last but not least, the unlawful removal of some computations such as multiply by zero can be considered as of no importance as the possible resulting loss is extremely scarce. All the more if the source image is absolutely correct and if graphical instructions, which apply to the saturated arithmetics, have been used. These arithmetics delimit the values of each pixel component in a interval ranging between 0 and 255. A pixel loss, *e. g.* a black or white satured pixel, can only occur if the register size is too small (in number of bits) to contain the result of the combination with the convolution filter. This overflow is highly unlikely and can be kept under control.

Vector-Matrix multiply In the case of the 3D transformation, we have chosen to remain close to the Mesa-6.0 implementation. Hence, our optimization techniques which take into account the ILP, are only applied line by line and not to all lines at the same time. We have coded two variants:

- The standard form is an equivalent of the static version with some little improvements such as a better use of ILP (particularly on Itanium 2) as:
  $d[0] = (((v[0] * m[0]) + (v[1] * m[1])) + ((v[2] * m[2]) + (v[3] * m[3])))$
  It is to be noted that this code version is adapted in accordance with the above criteria. Further in this article, we give another, more efficient version.

- The optimized form (see below) is not really that optimal, but is simple and easy to rapidly set up. It consists in reducing loadings and the number of arithmetic instructions. In fact, this segment of ccg code serves only to choose adequate instructions and to generate correct register allocation for a given matrix.

```
for(i=0 ;  i<4 ;  ++i)
|  if(0.0 == mat[row][i])
|  |  if(0 == i)         #[ fadd f(10+row)=f0,f(32+i+4*row);;     ]#
|  else
|  |  if(1.0 == mat[row][i])
|  |  |  if(0 == i)      #[ fadd f(10+row)=f0,f(6+i);;            ]#
|  |  |  else            #[ fadd f(10+row)=f(6+i),f(10+row);;     ]#
|  |  else
|  |  |  if(-1.0 == mat[row][i])
|  |  |  |  if(0 == i)   #[ fneg f(10+row)=f(6+i);;               ]#
|  |  |  |  else         #[ fsub f(10+row)=f(10+row),f(6+i);;     ]#
|  |  |  else
|  |  |  |  if(0 == i)   #[ fmpy f(10+row)=f(6+i),f(32+i+4*row);; ]#
|  |  |  |  else         #[ fma f(10+row)=f(6+i),f(32+i+4*row), f(10+row);; ]#
```

The optimization of the dynamic generated code is done only at the level of 3D transformations and not at the level of data retrieval. However, there is a waste of time and space with the loading of zeroes and ones. Hence, another "parametric embedded code" of machine code has been added:

```
tmp = m[i][0];
if ((0 == opt) || ((1.0 != tmp) && (-1.0 != tmp) && (0.0 != tmp)))
|     #[ lfs r(2+4*i),((i*4+0)*FLOATSIZE), r5 ]#
```

In this case, there is no data loading when matrix values are equal to zero or one. It is important to note that the resulting code is written for a given matrix. So, in order to redeem the overhead of code generation, several iterations with different vectors have to be made. Such is often the case in image processing where a 3D scene can be completely modified in accordance with a unique matrix.

2.5 Measurements

Here (Figure 3), we focused exclusively on the cost of running a function dynamically generated using both modes, standard and optimized, compared to one in a standard

static C version. First, we notice the amount of time saved. In the case of the IPF architecture, in any experiment, we get, at best here, a 30% speed improvement with our dynamic version compared to the static version generated by the gcc v3.3 compiler. From that, we verify that gcc compilers do not take yet advantage of the specifics of IPF architecture (such as microSIMD), as it does for the Ultra-SPARC v9 or PowerPC architectures. In the last two experiments, the performance of our dynamic versions, and the opportunity to use them, depends strictly on the number of zeroes in the transformation matrix. The main reason is that static compilers have schedulers able to interleave data loadings and arithmetic operations. Our optimized version, which has been made at little cost, has those instructions kept apart.

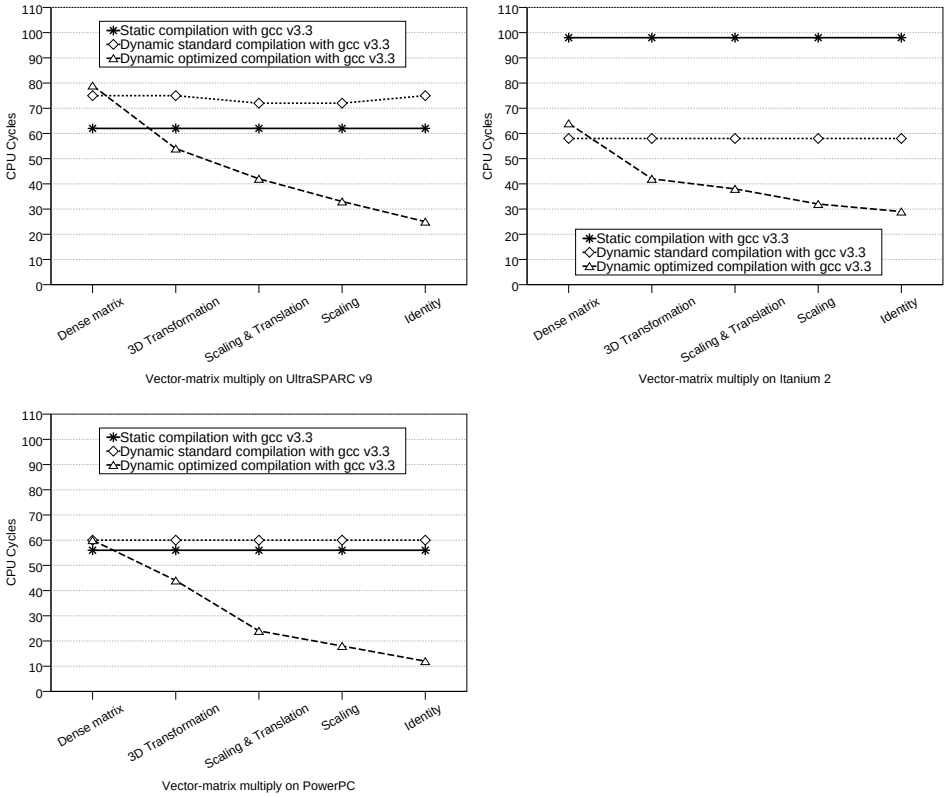

Fig. 3. Comparison between the different dynamic versions and their C counterpart by architecture

Another remark can be made about the optimized code generation graph which crosses the standard one in the case of the dense matrix experiments (in Figure 3). On the Itanium 2 platform, two parameters exercise an influence on each other. First, the standard dynamic version has been lightly modified to take into account instruction parallelism (ILP) and to reduce data dependencies. Then, the optimized dynamic code has been written without zeroes or ones contained in the specific matrix, regardless of other considerations. Hence, the algorithm for the optimized version has a line decomposition

model as:

$$d[0] = ((((v[0] * m[0]) + v[1] * m[1]) + v[2] * m[2]) + v[3] * m[3])$$

without unrolling, explains penalties and stalls. We can note that with a degree four unrolling, this kind of overhead disappears.
Our performance becomes more acceptable as soon as the number of zeroes gets equal to 3 or more. It is the case for a geometric transformation matrix, classic form in 3D. We then see that the result is similar, whichever is the chosen platform: our optimized dynamic version strongly outperforms the statically compiled version. Finally, the performance of our generated code increases proportionally to the number of zeroes in the transformation matrix.
In the previous section, we compared the dynamic and static versions of our different transformations only in their use. We voluntarily omitted the cost of the dynamic code generation here. However, the overhead of this generation can not be neglected and, to redeem it, we have to reuse the dynamic version several times. Our goal is to find a satisfying compromise from which dynamic compilation becomes profitable. In Table 1, the cost of the code generation use is detailed for both versions, the optimized one as well as the standard, more straightforward version. It is very high for all platforms, particularly on Itanium 2 architecture, where ccg has a scheduler allowing to choose template fields (a tag which declares explicitely the instruction type on IPF).
It may be remarked that results on Itanium 2 architecture seem opposite to those of other platforms. The reason is that the generated code on Itanium 2 for the standard version is even larger (in terms of size) than the optimized one. For each line (of Mesa implementation), seven floating-point instructions are made instead of four in optimized version. The purpose of this implementation was to reduce data dependency and latency. Nevertheless, for dense matrices, the obtained performance result is satisfying: the standard optimized version is faster (around 50%) than the static one.

Platforms	Std. code (cycles)	Opt. code (cycles)
Itanium 2	19580	16260
UltraSPARC v9	2710	3450
PowerPC	5780	6370

Table 1. Overhead in CPU cycles for dense matrices

3 Results

3.1 Filter experiment

For all experiments, we use the suited aggressive options for the gcc compiler. In the case of the mean filter (Figure 4), the speedup we get with the code generated by our compilets is of 4 at worst on the Itanium 2 architecture, and of 17 at best. In fact, we used the parallelism micro-SIMD instructions in our compilets, which are adapted to graphic applications, whereas the gcc compiler can only use a classical instruction set. On the UltraSPARC platform, our compilets are compared to their VIS counterpart, and we get a 1.2 speedup. On the PowerPC architecture, our generated code is compared to

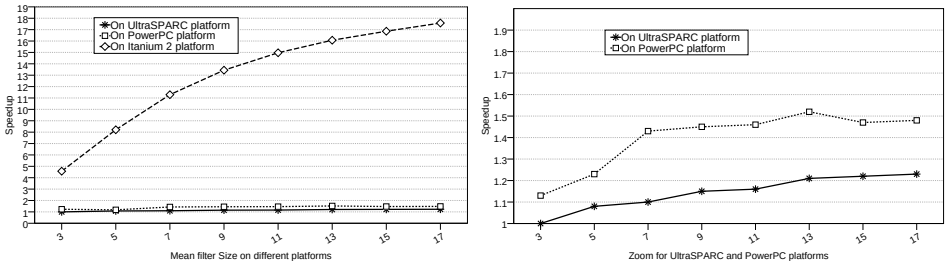

Fig. 4. Mean filters: Comparison between dynamic and C versions for a 512*512 image by architecture

the Altivec version, and it appears that we get a 1.4 speedup. Moreover, with a sparse filter our compilets obtain, on UltraSPARC and PowerPC architecture respectively, a speedup of 1.4 and 3.

Platforms	Static C code (in s.)	dynamic code (in s.)
UltraSPARC	0.56	0.38
Itanium 2	0.79	0.10
PowerPC	0.15	0.10

Table 2. Gaussain filter: Comparison between dynamic and C version for a 1024*678 image by architecture

Finally, we have tested a $5 * 5$ Gaussian filter (Table 2) and we have obtained on UltraSPARC, PowerPC and Itanium 2 platform, a speedup of 1.4, 1.5 and 7.6 respectively.

3.2 Vector-Matrix experiments

This simple experiment is an especially adequate candidate for run-time code generation as it is a geometric transformation (which is one of the basis of 3D operations). It is possible to rewrite it, either with assembly floating-point instructions, or with assembly packed FP arithmetic instructions. As we gave ourselves some constraints, the choice of instruction type has been restricted to floating-point.

On the Itanium platform (figure 5), the overhead due to the generation of dynamic code, standard or optimized, is redeemed for with 400 uses of the generated function. It is an acceptable number, as normal exploitation of a 3D transformation can use more than one million points. In the case of a non-full matrix, results are even more acceptable: the cost of the optimized generated code is covered at only 200 uses with a 16% speedup improvement for a geometric transformation, and a 28% speedup improvement for a scaling matrix. This gain, earned thanks to a small effort of conception and coding, largely justifies the use of dynamic compilation for image processing on the Itanium architecture, even more considering that no optimization has been yet made with specialized assembly instructions.

The highest speedup improvement of around 43% has been obtained for the processing of 50000 points with the dynamically generated function. This means that dynamic

compilation will reach an optimal efficiency with 3D scenery which size is usually around one million points or more. But it is to be noted that its efficiency appears at around 300 points.

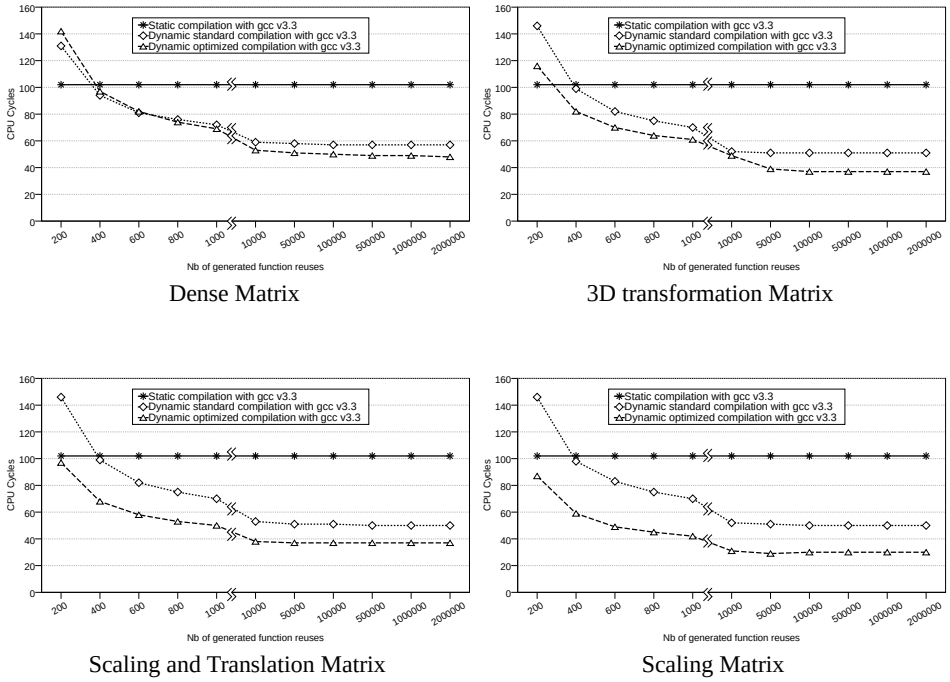

Fig. 5. Itanium 2 Architecture

On the UltraSPARC v9 platform (figure 6), we did not implement any optimization in data loading. For a full matrix, we then observe that the overhead of code generation is too significant to be easily redeemed. Hence, our dynamic versions can not challenge the static function (not without a higher cost in development). The use of our dynamic programs only becomes acceptable when the number of zeroes gets above four, or when the number of uses gets above 800, which remains quite pertinent for image processing. The highest speedup improvement of around 25% is reached here again when 50000 points are processed with the dynamically generated function.

On the PowerPC platform (figure 7), for a full matrix, the overhead of compilation is still significant to be redeemed in a small number of uses and, once more, our dynamic versions can not rival the static function. We have there to go above 1000 uses of our dynamic function for the dynamic compilation to be profitable. However, in the case of image processing, once again the result is acceptable. This observation gets reversed when the matrix contains three or more zeroes. We then get a clear advantage favoring the dynamic version over the static version, as we get a 8 to 25% speedup improvement compared to the latter.

The highest speedup improvement of around 41% is reached for a processing of around 10000 points with the dynamically generated function. As already noted, our dynamic

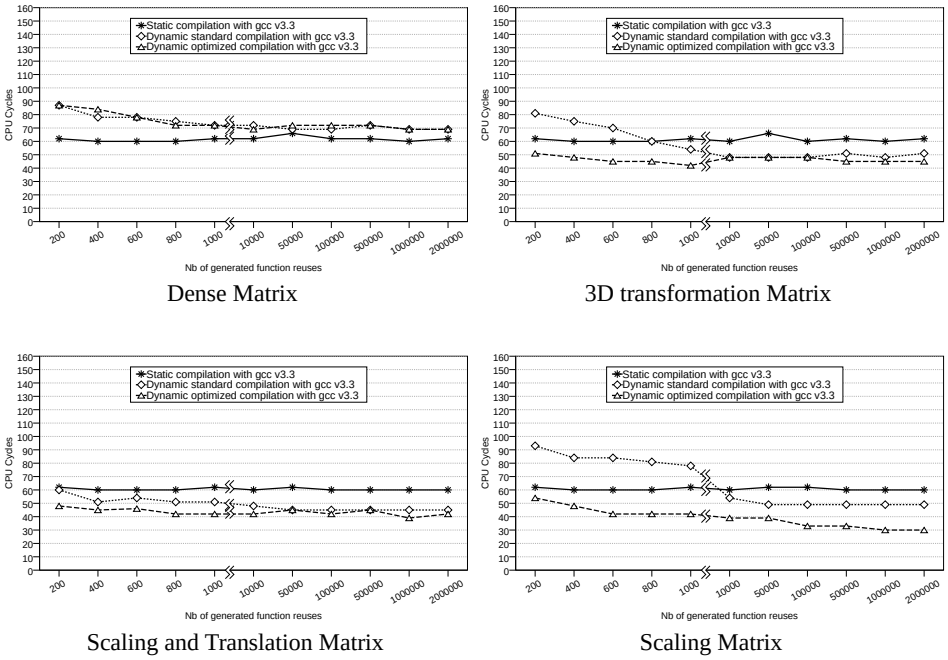

Fig. 6. UltraSPARC v9 Architecture

version is here again completely adapted to image processing.

Dynamically generated code size and binary compilet size. Our technique bestows another advantage. The binary code size is smaller compared to the C version generated by static compilers. For instance, in the Itanium 2 architecture, the generated static code (with gcc compiler version 3.3 and the previous defined options) has a size of 658 bytes in the case of a dense matrix. For a geometric transformation matrix, this size is of 466 bytes and of 370 bytes for a scaling matrix. It is to be noted that on UltraSPARC, for a dense matrix, the code size is around 224 bytes and around 168 bytes on PowerPC. So, the code size on Itanium architecture is twice as large as its counterparts on UltraSPARC and PowerPC. In fact, on the Itanium 2 platform, all instructions are inevitably stored per three in structures named “bundles”. Each bundle contains a tag named “template field” that determines which functional unit is able to evaluate the statement and what interlacing can exist between the functional units. Those template fields only give the possibility to store a unique floating-point unit per two bundles and, in order to fill those bundles, ccg adds “nop” instructions. This explains the code size on Itanium 2 in comparison with its counterparts of other platforms. However, the generated dynamic code remains smaller on Itanium 2 architecture than the static one, at worst by 22%, and can be reduced by 43% for real scaling matrices.

These results do not take into account the size of the compilet, which is necessary to dynamic code generation. On all architectures, the memory footprint is below 8 kbytes.

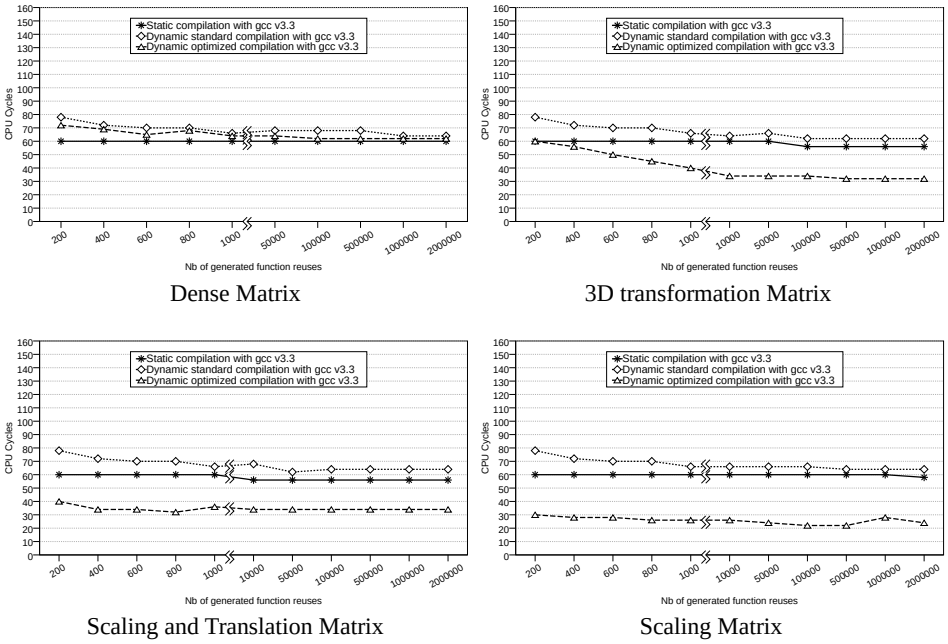

Fig. 7. PowerPC Architecture

Platforms	compilet code size (bytes)
UltraSPARC	3100
PowerPC	3260
Itanium 2	7107

It appears clearly, in this condition, that the size of the compilet added to the generated code makes it bigger than its static counterpart. But it is to be noted that this same compilet is intended for intensive use, as it can generate several different codes. It is, above all, a minimalistic code generator, of a size hardly reaching 8 kbytes, and able to elaborate a greatly optimized code.

4 Related work

While there is a large body of research on dynamic compilation built over the last ten years, only a small part of this research is close to our preoccupations here. For the majority of them, the dynamic code generator is often a large and complex program which allows the programmers to express algorithms in high-level languages such as Fabius [13]. Some other dynamic code generators such as Calpa [17] or Dynamo [15] respectively generates annotations automatically for the DyC dynamic compiler [9] in combining execution frequency and value profile information to choose runtime constants and other dynamic compilation strategies or tries to improve performance by identifying frequently taken paths through a program. It is to be noted that Dynamo's approach, which tries to speed up an application in executable format, is complementary

to Calpa's. On the contrary, our compilet, which uses low level compiler techniques, is a minimal dynamic code generator which creates only the needed code with a small cost in terms of size and development.
Some researches on staged dynamic compilers, which postpone a portion of compilation until runtime when code can be specialized based on runtime values [4,10], focus on spending as little time as possible in the dynamic compiler performing extensive offline pre-computations. This technique, which is also used in our compilets thanks to the ANSI C macros producing executable code segments, negates the need for any intermediate representation at runtime. However, we have chosen to return at low level compiler techniques to take into account the graphical instruction set for each platform, if it exists, and obtain better performances. We use some parametric codes which are written directly in low level language.
In our domain, a group which has conceived the FFTW [7] does a static optimized code generation with dynamic scheduling. Our compilet prefers using a code generator which creates the optimized adapted code at runtime.

5 Conclusion

In this article, we have confirmed that it is possible to use the data values input as parameters for an effective run-time code generator in multimedia applications. Results highlight that the produced binary code can be of satisfying quality, tailored for the data input, while needing only a slight programming effort.
It is important to note that a "compilet" is of very small size and is now platform-independent. For instance, the vector-matrix compilet for UltraSPARC is around a hundred lines of mixed C and ccg code. From this small programming effort, in our opinion, we got better performance than a static compiler consisting of thousands of code lines. This technique gives direct access to specific instructions of the target processor. This is an elegant way to deal with arithmetics such as saturated arithmetics which are seldom supported by compilers. The C dialtec Cg [19] allows to generate code for graphical processors but can not mix C and assembly code nor interact with general purpose processors. This is also a way to easily and effectively use vector units without having to deal with a vectorizer enabled compiler which generally requires to rewrite of the code.
In future works, we will implement those "compilets" in mainstream applications such as mathematical kernels and 3D visualization softwares, and we will include them in more complex run-time processes, able to detect when and where this technology will be useful (a form of JIT or a language interpreter).

References

1. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: principles, techniques, and tools*. Pearson Higher Education, January 1985.
2. AUSLANDER, J., PHILIPOSE, M., CHAMBERS, C., EGGERS, S. J., AND BERSHAD, B. N. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation* (May 1996), ACM Press, pp. 149–159.

3. AYCOCK, J. A brief history of just-in-time. *ACM Comput. Surv. Volume 35*, Issue 2 (2003), 97–113.
4. CONSEL, C., AND NOËL, F. A general approach for run-time specialization and its application to c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (January 1996), ACM Press, pp. 145–156.
5. DIEFENDORFF, K., AND DUBEY, P. R. How multimedia workloads will change processor design. *IEEE Computer Volume 30*, No. 9 (September 1997), 43–45.
6. FOLLIOT, B., PIUMARTA, I., SEINTURIER, L., BAILLARGUET, C., KHOURY, C., LEGER, A., AND A1, F. O. Beyond flexibility and reflection: The virtual virtual machine approach. In *International Workshop on Cluster Computing* (September 2001), vol. 2326, Springer-Verlag Heidelberg, p. 16.
7. FRIGO, M., AND JOHNSON, S. G. The fastest Fourier transform in the west. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, September 1997.
8. GRANT, B., MOCK, M., PHILIPOSE, M., CHAMBERS, C., AND EGGERS, S. J. The benefits and costs of dyc's run-time optimizations. *ACM Trans. Program. Lang. Syst. Volume 22*, Issue 5 (September 2000), 932–972.
9. GRANT, B., MOCK, M., PHILIPOSE, M., CHAMBERS, C., AND EGGERS, S. J. Dyc: An expressive annotation-directed dynamic compiler for c. *Theoretical Computer Science Volume 248*, Issue 1-2 (October 2000).
10. GRANT, B., PHILIPOSE, M., MOCK, M., CHAMBERS, C., AND EGGERS, S. J. An evaluation of staged run-time optimizations in dyc. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation* (May 1999), ACM Press, pp. 293–304.
11. JOY, B., STEELE, G., GOSLING, J., AND BRACHA, G. *Java(TM) Language Specification (First Edition)*. Addison-Wesley Pub Co, September 1996.
12. KENNEDY, K., AND ALLEN, R. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, October 2001.
13. LEE, P., AND LEONE, M. Optimizing ml with run-time code generation. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation* (1996), ACM Press, pp. 137–148.
14. LEE, R. B., AND SMITH, M. D. Media processing: a new design target. *IEEE Micro Volume 16* (January 1997), 43–45.
15. LEONE, M., AND DYBVIG, R. K. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report 490, Department of Computer Science, Indiana University, September 1997.
16. LEONE, M., AND LEE, P. A declarative approach to run-time code generation. In *Workshop Record of WCSSS 1996: The Inaugural Workshop on Compiler Support for System Software* (February 1996), ACM Press, pp. 8–17.
17. MOCK, M., CHAMBERS, C., AND EGGERS, S. J. Calpa: a tool for automating selective dynamic compilation. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture* (December 2000), ACM Press, pp. 291–302.
18. MUCHNICK, S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, August 1997.
19. NVIDIA CORPORATE OFFICE. *NVIDIA Cg Toolkit: Cg Language Specification*, nvidia corporation ed. 2701 San Tomas Expressway, Santa Clara, CA 95050, August 2002.
20. PALECZNY, M., VICK, C., AND CLICK, C. The java hotspot server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium* (April 2001), vol. JVM 2001, USENIX, the Advanced Computing Systems Association, pp. 1–13.
21. PIUMARTA, I. Ccg: a tool for writing dynamic code generators. In *Workshop on simplicity, performance and portability in virtual machine design held in conjunction with OOPSLA 1999 Conference* (November 1999).