Efficient data driven run-time code generation

Karine Brifault
PR*i*SM, Université de Versailles
Versailles, France
kbrifa@prism.uvsq.fr

Henri-Pierre Charles
PR*i*SM, Université de Versailles
Versailles, France
hpc@prism.uvsq.fr

ABSTRACT

Knowledge of data values at run-time allows us to generate better code in terms of efficiency, size and power consumption.

This paper introduces a low-level compiling technique based on a minimal code generator with parametric embedded sections to generate binary code at run-time. This generator called a "compilet" creates code and allocates registers using the data input. Then, it generates the needed instructions. Our measurements, performed on Itanium 2 and PowerPC platforms have shown a speed improvement of 43% on the Itanium 2 platform and 41% on the PowerPC one.

The proposed technique proves to be particularly useful in the case of intensively reused functions in graphic applications, where the advantages of dynamic compilation have not been fully taken into account yet.

1. INTRODUCTION

Many different techniques [17] improve code performance in terms of efficiency, power consumption or size. In classical static compilation, heuristics or other techniques such as loop unrolling or strength reduction are used. But, the information knowledge such as data values is missing at compile-time which would be very useful for statements generation when data values and invariants can be exploited [15]. The resulted code where less computations have to be executed, is often superior in speed to statically optimized code. As the data values can change at each run, the profile-based technique is not really convincing. The dynamic compilation is then the most suitable technique which complements static compilation taking advantage of data values and invariants for every run[11].

New methods have been introduced with the appearance of virtual machines. Java [10], for instance, has split compiling into a two-step process, consisting of two translation phases (source to bytecode and bytecode to native) 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 [1]. Target-dependent code is generated only at run-time, using a complex piece of code linked to the Java virtual machine (JVM). Hence, it takes time to compile a method, especially when we want 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 that 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.

This paper deals with applying dynamic compilation to multimedia applications on two different platforms using a toolkit called ccg [5][20]. Multimedia applications become one of the main used ones on personal computers[4]: because of the spreading use of complex processes geared towards them, mainstream users are requiring increasingly more efficient computers, at the lowest possible cost[13]. Hence, this topic represents a challenging target for us. We are working towards this goal by achieving more with a given computation power, in spite of an unavoidable overhead due to the dynamic code generation. The proposed technique determines the threshold at which reuses will off-set the overhead. In addition, it generates only the instructions that will be needed to process the data. Another advantage is that we use all the instruction set, even the graphical instructions, without relying on any multimedia libraries. However, we have to solve the problem of this code generation cost which at run-time can be high [8][7] and higher as the code complexity increases.

In section 2, we describe our experimental environment and the methodology used to create our "compilets". In section 3, we present and discuss our results on convolution filters

and geometric transformations. In section 4, we review some related work putting in prospect our contribution. Finally, in section 5, we conclude with several directions that will be explored in future research.

2. EXPERIMENTAL ENVIRONMENT

Dynamic compilation is not a new concept, but our intention is to apply it efficiently to multimedia, where some interesting restrictions exist and make interesting our algorithm of code generation. For this study, the execution of speed improvement have been monitored.

2.1 Methodology

To validate our approach, we experimented two multimedia applications, the convolution filter and a vector-matrix multiply. Convolution 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 1). We use the multimedia instructions which allow to process data values as vector or pixel and not as an integer or a float using saturated arithmetics to reduce the number of assembly statements and the size of the generated code.

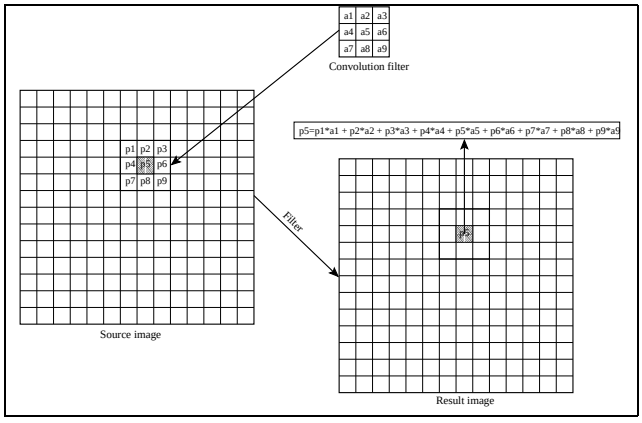


Figure 1: Example of a small image and convolution filter

In the vector-matrix multiply experiment, we consider an implementation from OpenGL Mesa-6.0.1, 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.
```

For the last experiment, we use two variants according to our algorithm: the standard form which is equivalent of the static version with some little improvements such as a better use of ILP, and the optimized form is not really that optimal, but is easy to set up. It consists in reducing loads and arithmetic instructions.

Besides, a specific algorithm has been created to take into account the particular arithmetics associated to pixels. In image processing, pixels, generally described by 4-byte integers, are controlled by the saturated arithmetics. For this reason we break the usual compilation rules in our algorithm by eliminating either the multiply operations by zero or by one. We can consider that this removal is of little consequence. Besides, it brings two benefits: first, the operations number is reduced as all the computations which are useless (multiplication by 1 or 0) disappear; Second, this reduction of instructions number decreases the execution time.

2.2 Overview of the compilets system

Our technique operates in two steps at runtime. The first one is to call a compilet, defined at compile-time in ccg language, which generates binary code according to the target architecture, the knowledge of data and our algorithm (Figure 2 and more detailed [2]).

```
for(i=0 ;  i<4 ;  ++i)
  if(0.0 == mat[row][i])
    if(0==i) genSub(regDst(lno),regDst(lno),regDst(lno));
  else
    if(1.0 == mat[row][i])
      if(0==i) genMove(regDst(lno),regSrc(i));
      else     genAdd(regDst(lno),regSrc(i),regDst(lno));
    else
      if(-1.0 == mat[row][i])
        if(0==i) genNeg(regDst(lno),regSrc(i));
        else     genSub(regDst(lno),regSrc(i),regDst(lno));
      else
        if(0==i) genMul(regDst(lno),regMat(lno,0),
                        regSrc(0));
        else     genMulAdd(regDst(lno),regMat(lno,i),
                           regSrc(i),regDst(lno));
```

Figure 2: Part of a compilet for a vector-matrix multiply

As the optimization of the dynamic generated code is done only at the level of 3D transformations and not at the level of data retrieval, 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)))
   genLoad(regMat(i,j),SIZEOFFLOAT*(4*i+j),regBase(0));
```

A compilet consists in reducing loads and arithmetic instructions such as computations by zero. It chooses adequate instructions and generates correct register allocation for a given matrix and a given architecture. The generated binary code is an executable function.

The second phase consists in calling this minimal generated function to execute the multimedia application.

One of the interests of our approach is the fact that all of our codes are written in C language in which several segments of dynamic code are embedded via an ISA (Instruction Set Architecture) which gives a portability to the compilets on different architectures. Hence, a developper can only use some defined compilets without any knowledge of the architecture.

2.3 Hardware and Software setup

Our experiments are made on Itanium 2 and PowerPC architectures. The Itanium processor is a uniprocessor operating

at 900 MHz with 2 GB of memory. 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. 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 32KB and L2 cache of 256 KB.

The test machines were respectively running Linux Red Hat 7.1 based on 2.4.18 SMP kernel for Itanium 2 and MacOS 10.2.3 Darwin Kernel version on PowerPC.

The compiler for all platforms is the 3.3 version of gcc. We observe some constraints in the choice of the 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 are added. Results are given in CPU cycles and are an average of 100000 of experimental measures.

3. RESULTS

3.1 Vector-Matrix experiments

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 PowerPC architecture. In the last experiment, 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.

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 is detailed for both versions: the optimized and the standard one. The cost is relatively important for all platforms, particularly on Itanium 2 platform, where ccg has a scheduler to choose the template fields (a tag which declares explicitly the instruction type on IPF).

Platforms	Std. code (cycles)	Opt. code (cycles)
Itanium 2	19580	16260
PowerPC	5780	6370

Table 1: Overhead in CPU cycles for dense matrices

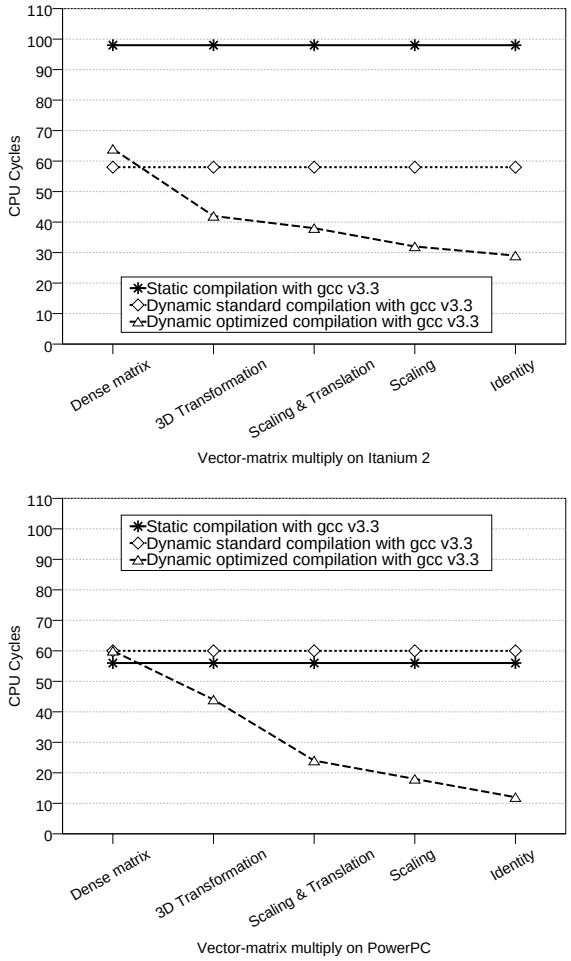
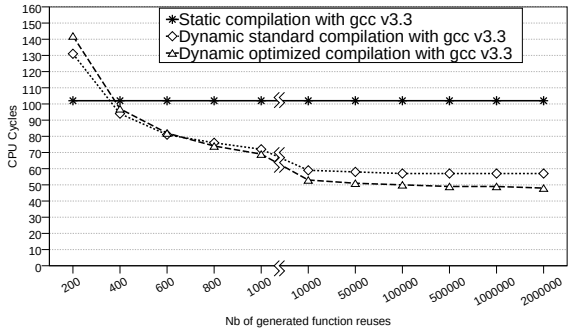


Figure 3: Comparison between the different dynamic versions and their C counterpart by architecture

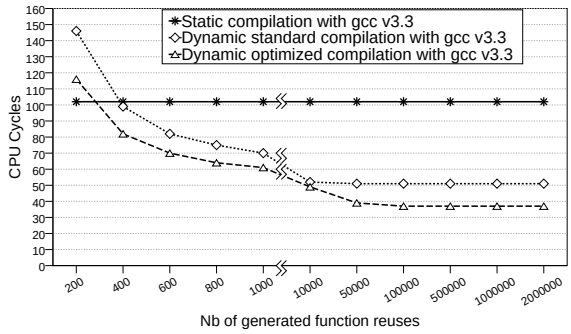
This simple experiment is suitable to show the dependency between overhead redeeming of code generation and function reuses.

On the Itanium platform (Figure 4), the overhead due to the generation of dynamic code, standard or optimized, is redeemed starting from 400 uses of the generated function. It is an acceptable number, as regular exploitation of a 3D transformation can use more than one million points. In the case of a 3D and scaling transformation respectively, the cost of the optimized generated code is covered at only 200 uses with a 16% and 28%speedup improvement.

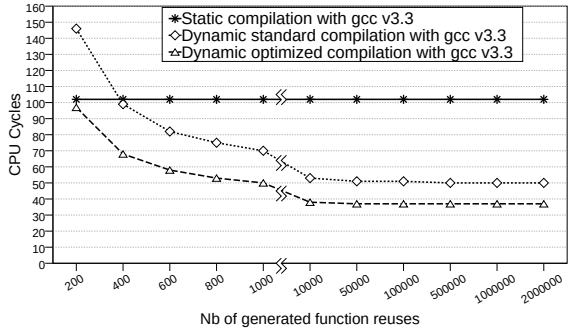
The highest speedup improvement of around 43% has been obtained for the processing of 50000 points with the dynamically generated function. This means that dynamic compilation will reach an optimal efficiency with 3D scenery which size is usually around one million points or more. However its efficiency appears since around 300 points. 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.



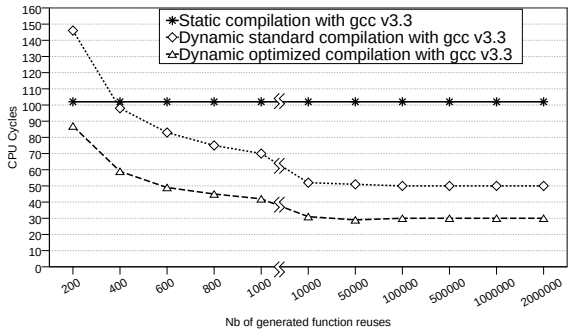
Dense Matrix



3D transformation Matrix



Scaling and Translation Matrix

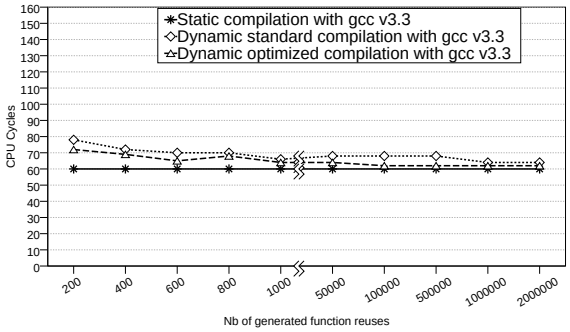


Scaling Matrix

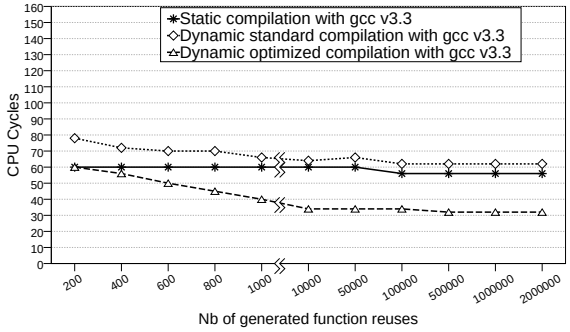
Figure 4: Itanium 2 Architecture

On the PowerPC platform (Figure 5), for a dense matrix, the overhead of compilation is still significant to be redeemed in a small number of uses and, once more, our dynamic versions are not enough efficient in comparison with 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 8 to 25% speedup improvement compared to the latter.

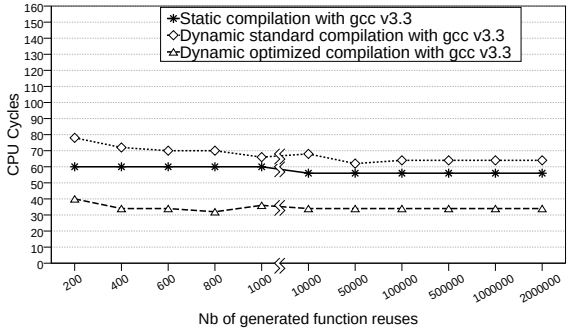
The highest speedup improvement of around 41% is reached for a 10000 points processing with the dynamically generated function. As already noticed, our dynamic version is here again completely adapted to image processing.



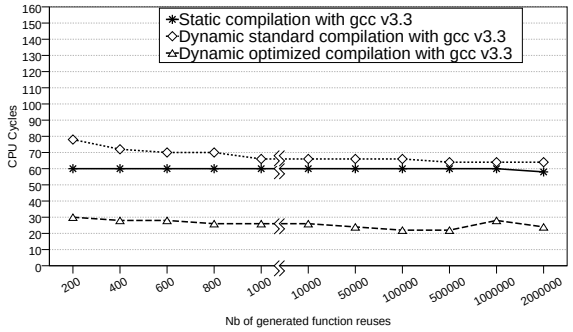
Dense Matrix



3D transformation Matrix



Scaling and Translation Matrix



Scaling Matrix

Figure 5: PowerPC Architecture

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. Indeed, with our algorithm, all the useless computations are eliminated. For instance, in the Itanium 2 architecture, the generated static code 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 PowerPC, for a dense matrix, the code size is around 168 bytes. So, the code size on Itanium architecture is twice as large as its counterparts on 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. On all architectures, the memory footprint is below 8 kbytes.

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

The size of the compilet added to the generated code makes it bigger than its static counterpart. But a same compilet, of a size hardly reaching 8 kbytes, is intended for intensive use, can generate several different codes and, above all, able to elaborate a greatly optimized code.

3.2 Filter experiment

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 PowerPC, the number of registers is only a half 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.

In the case of the mean filter (Figure 6), the speedup we obtain using our compilets for code generation is of 4 at worst on the Itanium 2 architecture, and of 17 at best. In fact, we use the micro-SIMD instructions parallelism in our compilets, which is adapted to graphic applications, whereas the gcc compiler can only use a classical instruction set. On the PowerPC platform, our generated code is compared to the Altivec version, and it notices a speedup of 1.4. Moreover, with a sparse filter, our compilets obtain, on PowerPC platform a speedup of 3. Finally, we have tested a $5*5$ Gaussian filter and we have obtained on both PowerPC and Itanium 2 platforms, a speedup of 1.5 and 7.6 respectively.

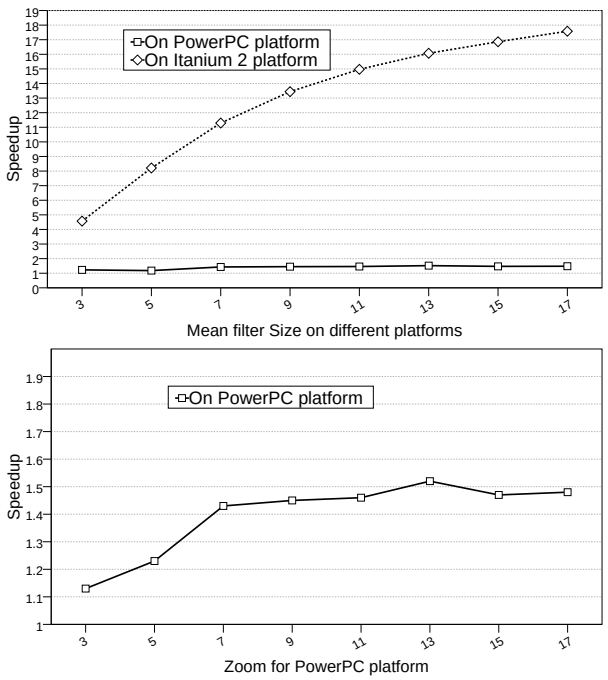


Figure 6: Mean filters: comparison between dynamic and C versions for a 512*512 image

4. RELATED WORK

Among all researchs on dynamic compilation built over the last ten years, only a small part is close to our concerns here. For the majority of them, the dynamic code generator is often a large and complex program which allows the pro-

grammers to express algorithms in high-level languages such as Fabius [12], Calpa [16], Dynamo [14] or HotSpot [19]. On the opposite, 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 coding time.

Some researchs on staged dynamic compilers, which postpone a portion of compilation until runtime when code can be specialized based on runtime values [3][9], 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 negates the need for any intermediate representation at runtime. However, we choose to return at low level compiler techniques to take into account the graphical instruction set for each platform, if it exists, and obtain better performance.

In the dynamic compilation for multimedia, a group which has conceived the FFTW [6] 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 paper, we 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 the "compilet" is of very small size and is platform-independent. With a small programming effort, we get 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 [18] allows to generate code for graphical processors but can not mix C and assembly code nor interact with general purpose processors.

In future works, we will implement those "compilets" in mainstream applications such as 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.

6. REFERENCES

[1] Aycock, J. A brief history of just-in-time. *ACM Comput. Surv. Volume 35*, Issue 2 (2003), 97–113.

[2] Brifault, K., and Charles, H.-P. Effective and economical run-time code generation driven by data for multimedia applications. Technical Report 2004/62, PRiSM Laboratory, University of Versailles, July 2004.

[3] Consel, C., and Nol, 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.

[4] Diefendorff, K., and Dubey, P. R. How multimedia workloads will change processor design. *IEEE Computer Volume 30*, No. 9 (September 1997), 43–45.

[5] 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.

[6] 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.

[7] 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.

[8] 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).

[9] 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.

[10] Joy, B., Steele, G., Gosling, J., and Bracha, G. *Java(TM) Language Specification (First Edition).* Addison-Wesley Pub Co, September 1996.

[11] Kennedy, K., and Allen, R. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, October 2001.

[12] 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.

[13] Lee, R. B., and Smith, M. D. Media processing: a new design target. *IEEE Micro Volume 16* (January 1997), 43–45.

[14] 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.

[15] 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.

[16] 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.

[17] Muchnick, S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, August 1997.

[18] NVIDIA Corporate Office. *NVIDIA Cg Toolkit: Cg Language Specification*, nvidia corporation ed. 2701 San Tomas Expressway, Santa Clara, CA 95050, August 2002.

[19] 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.

[20] 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).