GCDS: A Compiler Strategy for Trading Code Size Against Performance in Embedded Applications

François Bodin, Zbigniew Chamski, Christine Eisenbeis, Erven Rohou, André Seznec

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

GCDS: A Compiler Strategy for Trading Code Size Against Performance in Embedded Applications

François Bodin, Zbigniew Chamski,
Christine Eisenbeis, Erven Rohou,

André Seznec

N° 3346

Janvier 1998

THÈME 1

ISSN 0249-6399

GCDS: A Compiler Strategy for Trading Code Size Against Performance in Embedded Applications*

François Bodin, Zbigniew Chamski,
Christine Eisenbeis, Erven Rohou,
André Seznec

Thème 1 — Réseaux et systèmes
Projet CAPS

Rapport de recherche n3346 — Janvier 1998 — 21 pages

Abstract: In this paper we present GCDS, a new approach for code optimization within the context of embedded systems. GCDS applies several transformation sequences on each piece of code and chooses *a posteriori* the best trade-off between code size and performance, given the particular constraints. The proposed implementation relies on a modular framework that makes easy to handle different compilation strategies on the same piece of code.

Key-words: VLIW, compiler, software pipeline, trade-off, code size, execution time, embedded systems

(Résumé : tsvp)

* This study is supported by the OCEANS Esprit project.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

GCDS:

Résumé : Dans cet article nous proposons une nouvelle approche appelée GCDS pour l'optimisation d'applications pour systèmes embarqués où les contraintes de performance et de taille de code sont fortes. GCDS explore un certain nombre d'optimisations sur chaque fragment de code et choisit ensuite le meilleur compromis par rapport aux contraintes fixées. Pour la mise en œuvre nous proposons une infrastructure modulaire qui permet l'exploration de plusieurs séquences d'optimisations sur un même fragment de code.

Mots-clé : VLIW, compilateur, pipeline logiciel, compromis, taille de code, temps d'exécution, systèmes embarqués

1 Introduction

In embedded applications, both performance and code size have traditionally been critical, leading to the use of hand-optimized codes (at least for critical kernels). But during the past few years, embedded applications have become more and more complex involving more and more controls. At the same time, the architecture of high-end micro-controllers and DSPs have become increasingly complex. Instruction-level parallelism (VLIW architectures) and deep pipelines are used in the new generation of multimedia processors (e.g., Philips TriMedia or TI TMS320C620). In this context, manual optimization of codes is becoming quite unrealistic, even for critical loops. Moreover, most optimizations used to improve performance also increase code size [3]. This may be counter-productive in many cases due to a raise of instruction cache misses. For embedded applications, code size inflation may even be more disastrous: the code is often stored on a non-volatile support of limited size (e.g., a Flash EEPROM).

Therefore, a compiler for an embedded system must be able to achieve high performance while keeping code size under control. Ideally, one wants it to compete favorably with human experts on small critical kernels, but also to globally control the size of the code. To achieve the first goal, the compiler must be able to explore many different strategies for scheduling, register allocation, superblock optimization, loop unrolling, etc. To reach the second goal, it must implement a strategy that balances code size with execution time in a global way on the entire application. The impact of each code fragment has to be evaluated on both the overall performance and the overall code size. Furthermore, one may also want to master other criteria, such as instruction cache constraints. In other words, for any application, the compiler must be able to answer globally either of the following questions: "Given a maximum code size, what is the highest performance that can be achieved?", or "Given a performance goal, what is the smallest code size that can be achieved?"

In this paper, we present a new compiler strategy that addresses this issue, not on a single loop or a single loop nest, but on a complete application; we call it GCDS (Global Constraints-Driven Strategy).

GCDS differs from traditional compiler heuristics. First, as compilation speed is not a critical issue for embedded systems, many optimization sequences may be explored for each code fragment. This allows to evaluate alternatives that may not be the best in the common case, but may lead to better code size/performance trade-offs in particular cases. Second, choices of the precise optimization sequence for each code fragment is done globally according to performance or code size criteria. To guide the choices, for many embedded applications, profiling results can be gathered. These results are used for weighting the impact of different code fragment on the generation decision.

In this paper, GCDS is illustrated on low-level optimizations such as scheduling [13], unrolling [3], superblock [11] and software pipelining [14, 18, 22] applied to loops. However, GCDS may also be applied to other code portions as well as at other compiler stages.

The remainder of the paper is organized as follows: Section 2 introduces the GCDS strategy. In Section 3 we give an overview of SALTO and SEA, an experimental framework for implementing assembly-level optimization sequences. Section 4 presents the application

of the GCDS strategy to loop optimizations at assembly code level, as this is where the most critical and most specific optimizations are implemented for application-specific processor and DSPs. Using a simple example, we show that balancing code size and execution time cannot be done using a gross approximation but depends on fine grain information such as the number of iterations for each loop being processed. In Section 5 we apply GCDS to the H263 video application for Philips TriMedia processor. Finally, we give an overview of related work and conclude with some directions for further investigations.

2 GCDS: Basic Principles

Traditional compilers for general-purpose applications do not take into account global constraints on applications such as code size, real time execution, etc. They have a single built-in transformation path and will apply locally a set of heuristics to optimize the execution time. In most compilers, a single optimization heuristic will be tried on each code fragment. In the best case, a set of heuristics is tried but the choice of the selected code is done locally and is generally determined only through a rough performance evaluation. Such compilers do not control code size explosion.

A contrario, the philosophy of our Global Constraints-Driven Strategy (GCDS) is to use global information on the complete application to select the precise heuristic that will be applied locally on each code fragment.

GCDS relies on three main ideas:

1. For each code fragment, there exists several code generation alternatives that lead to different implementations and different code sizes and execution times. These code generation alternatives may all be analyzed to determine the resulting code sizes and execution times.

2. A *global* selector function (for instance `select` function in Figure 6) chooses the best code generation among alternatives for each code fragment. The choice is controlled by the total execution time, code size, or any other global constraint.

3. Code fragments in an application must be weighted to enable this global selection. These weights should be determined from profiling information, allowing GCDS to focus on the most time consuming parts of the code.

GCDS is illustrated in this paper on low-level loop transformations. However, such a global strategy may be applied for other types of code optimizations. Examples of such are data layout (for instance should a matrix or its transpose be used in an application?), procedure or function call inlining, etc.

Before presenting in details the strategy, we overview the experimental framework.

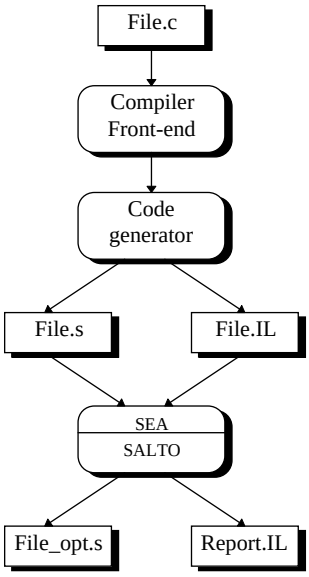

Figure 1: Compilation Process.

3 Experimental Framework

The implementation infrastructure consists in two levels. The first one, called SALTO [17], is a general purpose tool that make manipulation of assembly codes easier. The second level, called SEA, is built on top of SALTO and is dedicated to code optimization for VLIW and DSP processors. An important property of SEA is to separate implementation of code transformations from the global compiler strategy. As we are working at assembly level, additional information is transmitted to the optimizer by upstream compiler stages via an interface language called IL [5]. Figure 1 illustrates the global organization of the compilation process. In the remainder of this section we briefly present these components.

3.1 Salto: A Retargetable System for Assembly Language Transformation and Optimization

SALTO is a retargetable framework for developing a whole spectrum of tools that manipulate programs expressed in assembly language. The objective of such a system is to provide the user with a single environment that permits him to implement algorithms needed for performance tuning of low-level codes. This set of tools includes assembly code schedulers (e.g. software pipeline), profiling, and tracing tools. SALTO is easier to retarget than a compiler and, at the same time, it does not operate on executable codes. Most of the

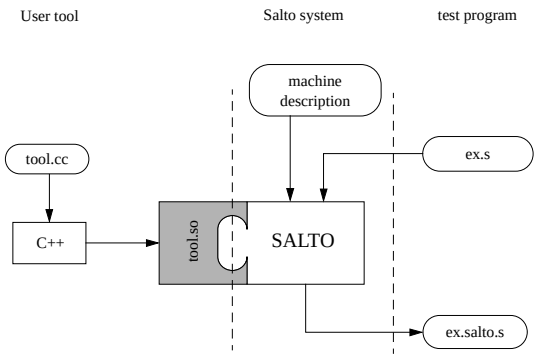

Figure 2: SALTO

information needed for building the control flow graph, performing register allocation, etc. is still available at the assembly code level.

An application built with SALTO consists of three parts, as shown in Figure 2: a kernel (SALTO), a machine description file and an optimization or instrumentation algorithm (tool.so).

- The kernel performs common house-keeping tasks the user doesn't want to worry about: the parsing of the assembly code and of the machine description file, and the construction of an internal representation.
- The machine description file provides a model of hardware configuration and the complete description of the instruction set, including per-instruction resources reservation tables.
- The optimization or instrumentation algorithm is supplied by the user. Once the system has read the machine description file and the assembly code, and the internal representation is built, the control is passed to the user-supplied function called `Salto_hook`.

The user interface to access the internal representation of assembly code of SALTO is object-oriented. The classes provided allow to see each function in the form a control-flow graph and to access to resources usage of each instruction. For further information on SALTO the reader can refer to [17].

3.2 SEA : Salto Enhanced Abstraction

To easily implement the code optimization we designed a set of classes, SEA, that provides an abstract view of the assembly code which is more specific to code scheduling and register allocation. The most important features of SEA are to allow the evaluation of various code

optimizations before producing the final code, and to separate the implementation of the global compiling strategy from the implementation of individual optimization sequences. The SEA model contains two kinds of objects:

subgraphs, which are parts of the control flow graph. The following objects can be used in SEA: `seaINST`: an instruction object; `seaCF`, an unstructured set of code fragments; `seaSCF`, a structured subset of control flow graph with a unique entry point; `seaSBk`, a superblock; `seaBBk`, a basic block; and finally, `seaLoop`, a structured piece of code that has loop properties. Figure 3 illustrates the corresponding class hierarchy.

transformations to be applied to subgraphs. All transformations are characterized by the following main methods: `preCond()` returns the set of control flow subgraphs that qualifies for the transformation; `apply()` applies the transformation to a given subgraph, and finally, `getStatus()` that returns the status of the transformation after application. The status can be *success* or *failure*, and the reason of the failure can be extracted through a generic diagnostic mechanism.

Optimizations currently available are the following:

register renaming: renames local registers in each basic blocks. This aims at removing false dependences.

loop unrolling: unrolls loop bodies.

local/superblock scheduling: this transformation performs the scheduling of basic blocks or of superblocks [16, 11, 12].

superblocks construction: gathers a set of basic block into a superblock [11].

guard insertion adds guards to instructions to remove jumps and thus allows scheduling across jumps [10].

software pipeline generates a modulo scheduling of the loop body. Registers are renamed to achieve low latency. The algorithm used to compute a modulo scheduling is based on the method proposed in [22].

register allocation: This transformation allocates registers either before or after scheduling of the instructions. If guards are added to instructions then the allocator assumes that destination registers may not be killed.

The usage of the SEA objects is shown in Figure 4. A transformation is tried on a cloned piece of code, then according to performance or size criteria one of the solutions found is chosen and propagated to the low-level program representation using the `rebuild()` method.

For the purpose of this study, four optimization sequences, shown in Figure 5, are used:

sq0 is the simplest transformation sequence. First, registers are renamed to remove as much anti-dependences as possible to improve code compaction. The code is then scheduled locally; register allocation is performed as the last step.

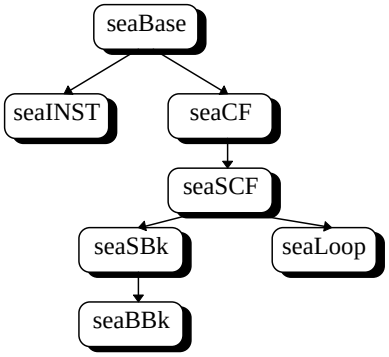

Figure 3: Sea class abstraction

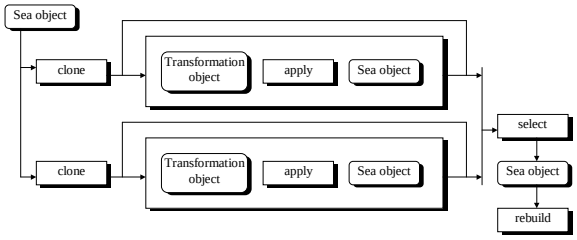

Figure 4: Typical usage of sea classes.

	sq0	**sq1(UF)**	**sq2(UF)**	**sq3**
step 1	reg. rename	reg. rename	reg. rename	soft. Pip.
step 2	local schedul.	unroll $\times UF$	unroll $\times UF$	
step 3	reg. alloc.	superblock	superblock	
step 4		scheduling	reg. alloc.	
step 5		reg. alloc.	scheduling	

Figure 5: Sequence of optimizations. UF is the unroll factor.

sq1(UF) is based on unrolling the loop body **UF** times. The unrolled body is transformed into a superblock, and conditional instructions are eliminated through the insertion of guards, resulting in a large basic block. As in **sq0**, register allocation is performed after local scheduling. Register renaming is not applied, since the introduction of guards hides actual register writes to the renaming algorithm.

sq2(UF) is similar to **sq1(UF)** but register allocation is performed before scheduling. This decreases the code compaction potential, but usually requires less registers, allowing this sequence to succeed when **sq1(UF)** fails due to a lack of register. Currently, this is the least effective optimization sequence, since register allocation introduces many anti-dependences that were removed by the register renaming phase.

sq3 consists in applying a software pipeline algorithm. This sequence is limited to loops whose bodies are a single basic block.

Other optimization sequences might as well be tested.

3.3 A Target Architecture: the Philips TriMedia

The Philips TriMedia TM-1000 [6] is a processor intended for high-performance multimedia applications, from video phones to multi-purpose programmable plug-in cards for personal computers. The TM-1000 consists of a general purpose CPU core (DSPCPU) and a range of media-specific units: video and audio interfaces and coprocessors, memory interface, PCI bus interface, etc. The different units are interconnected through a high-bandwidth internal bus.

The 32-bit DSPCPU consists of 27 functional units, and 128 general-purpose registers. It is supported by separate instruction and data caches (32 and 16 Kbytes, respectively). The instruction set uses a VLIW [8] architecture and allows up to five operations to be issued in each clock cycle, at a clock rate of 100+ MHz. An instruction can also be guarded by a condition register.

Operation latencies are of one, two, three, and seventeen cycles. Most operations are pipelined and can be issued every cycle. However, the choice of a VLIW instruction set architecture requires the issue of instructions to be controlled in software, and introduces program structure constraints at assembly level: delayed branches, and mapping restrictions between instruction issue units (slots) and functional units. There are three cycles between the issue of a branch or jump instruction and the effective change in control flow. In addition, certain operations can only be issued in specific slots (i.e., at certain locations in the instruction word), introducing slot allocation conflicts.

4 Applying GCDS to low-level optimization on loops

The application of GCDS to low-level optimization on loops is illustrated in Figure 6:

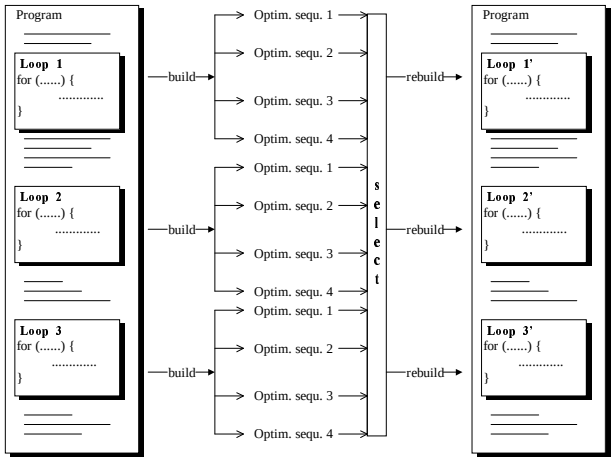

Figure 6: GCDS overall organization.

1. a set of transformation sequences (for instance: unrolling, superblock, scheduling and register allocation) is applied to each loop to produce many different implementations for each loop;
2. profiling is used to extract the number of iterations executed by each loop and the number of times the loops are executed[1]; using this profiling information, GCDS focuses on the most time-consuming parts of the code;
3. a global function (i.e. the `select` function in Figure 6) that chooses globally, under an execution time or code size constraint, the best implementation of each loop.

The first point is directly related to the capabilities of the compiler infrastructure: it requires the compiler to generate multiple codes for each loop. The third point depends on the ability to model the optimized code and to formalize the selection process. To achieve this, we use a linear algebra approach to be able to formulate the choices as an integer simplex problem in which either the code size or execution time has to be minimized.

In next section we present the performance model used for loops, then in section 4.2 we show how the linear inequalities system is built and solved. An example is used to illustrate the approach.

4.1 Loop Performance Model

To model the cost and execution time of loop L_i, we use a number of execution cycles per iteration of the original loop, denoted a_i, and an overhead cost, denoted b_i, corresponding to

[1] Accuracy of loop iteration numbers is important if they are small, which is frequently the case in video application such as H263.

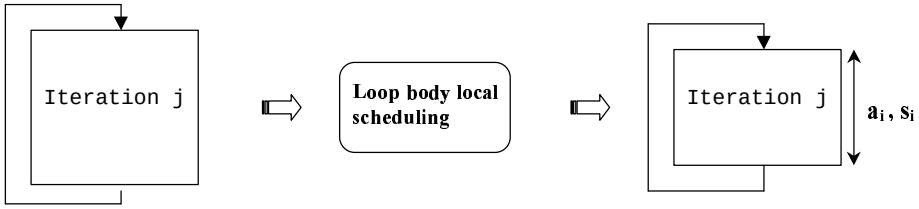

Figure 7: Loop model for the loop body scheduling only.

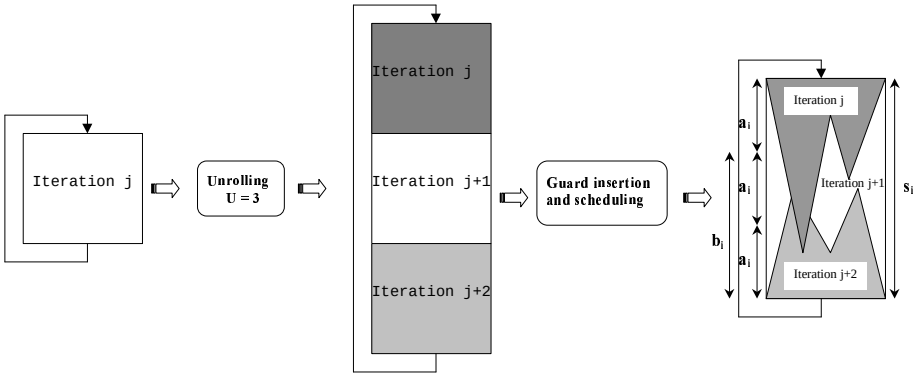

Figure 8: Loop model for loop unrolling technique.

the loop epilogue/prologue. The value of b_i may vary in function of the number of iterations of the loop. To each loop we also attach a weight, W_i, which corresponds to the number of times the loop is executed. Using this simple model the execution time, t, in machine cycles, spent executing loop L_i, is given by:

$$t = W_i \times (a_i \times n_i + b_i(n_i))$$

where n_i is the number of iterations of the loop. While this model may not be very accurate for loops with complex body, in most of cases, it is sufficient, especially when if-statements are converted to straight line code using guarded instructions. Furthermore, the number of cycles does not take into account the impact of data cache misses. However in practice, the selection of a scheduling algorithm does not depend on the memory behavior. Optimizations concerning the memory hierarchy are usually performed by the front-end. The framework we are proposing can easily be extended to support more complicated loop models as soon as the execution time remains written as linear expressions.

The loop code size is denoted s_i and expressed in VLIW instructions. On architectures such as the TriMedia this is clearly an approximation, since instructions are compressed in memory.

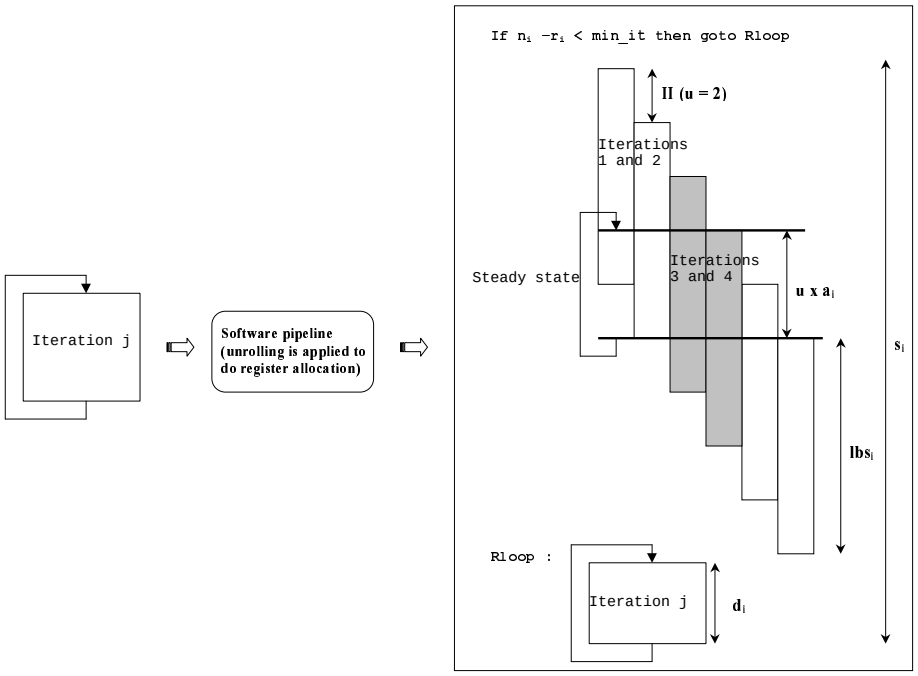

Figure 9: Loop model for the software pipeline technique.

4.1.1 Computing a_i and b_i

In the following we call *unroll factor*, denoted u, the number of times the unrolling is applied to a loop. The values of the coefficients are obtained according to the algorithm used to perform the loop scheduling:

body local scheduling: in this case the value of a_i is simply given by the number of cycles the loop body lasts after scheduling. In this case b_i is zero cycle. This is illustrated in Figure 7.

loop unrolling: unrolling in our case is followed by superblock construction and addition of guards to allow instructions to move across jumps. In this case b_i depends on the value of n_i. Indeed, the entire loop body is executed in this case, so the overhead cost $b_i(n_i)$ is given by:

$$b_i(n_i) = \begin{cases} \text{if } ((n_i \text{ modulo } u) \neq 0) & then & (u - (n_i \text{ modulo } u)) \times a_i \\ & else & 0 \end{cases}$$

This is illustrated in Figure 8.

software pipeline: The software pipelined loop issues an iteration every II cycles. A remainder loop is added to execute the extra iterations when the value of n_i is not a multiple of the unroll factor used by the software pipeline[2] or when the number of iterations is smaller than the minimum number of iterations, denoted min_it, that the software pipeline can perform (since in current implementation no guard are added in the resulting code). The value of a_i and the overhead cost $b_i(n_i)$ are given by:

$$\begin{array}{ll} \text{if } \; n_i - r_i > min\_it & \left\{ \begin{array}{l} a_i = II \\ b_i(n_i) = r_i \times d_i + lbs_i - a_i \times r_i \end{array} \right. \\ \text{else} & \left\{ \begin{array}{l} a_i = 0 \\ b_i(n_i) = n_i \times d_i \end{array} \right. \end{array}$$

where $r_i = (n_i \text{ modulo}(u))$, d_i is the cost per iteration of the remainder loop and lbs_i is the number of cycles of the software pipelined loop epilogue. The negative term takes into account that not all the iterations are executed by the software pipelined loop. This is illustrated in Figure 9.

Given l the number of loops in the code the total number of cycles spent in the loops is modelized by:

$$\sum_{i=1}^{l} W_i \times (a_i \times n_i + b_i(n_i))$$

A difficulty arises when the values of W_i and n_i vary from one run to another (see the values for H263 on Figure 13 for instance). Variations of W_i do not matter as soon as they keep the same relative values which is usually the case (i.e. the most time-consuming part of a code tends to remain the same). In this case it is possible to optimize for an average run by considering each value of n_i. For a loop L_i instead of $W_i \times (a_i \times n_i + b_i(n_i))$ we have:

$$\sum_{k=1}^{nbn} W_i^k \times (a_i \times n_i^k + b_i(n_i^k))$$

where nbn is the number of possible values for n_i. In next section we propose a select function based on the Simplex algorithm. For the sake of clarity we shall assume that there is only one possible value of n_i per loop in the next section.

4.2 The Select Function as a Simplex Problem

In this section we present how the global selection of code generation alternatives can be represented as the resolution of a Simplex problem.

Let L_i be a loop. Let opt_i^j, $j \in [1, nbopt]$ be the optimization sequences applied to loop L_i. Let s_i^j be the size of L_i's code for optimization j. Let $t_i^j(n) = a_i^j \times n + b_i^j(n)$

[2] Unrolling is used by the software pipeline algorithm to perform a cyclic register allocation.

be the number of cycles for n iterations of loop L_i. We denote δ_{ik} the integer variables used to encode the optimization alternatives in the equations. If $\delta_{ik} = 1$, for loop L_i, the optimization sequence k is the one to choose. For each loop we get the following integer equations:

$$\begin{array}{rcl} T_i(n) & = & \sum_{k=1}^{nbopt} t_i^k(n) \times \delta_{ik} \\ S_i & = & \sum_{k=1}^{nbopt} s_i^k \times \delta_{ik} \\ 1 & = & \sum_{k=1}^{nbopt} \delta_{ik} \end{array}$$

where $T_i(n)$ is the execution time in cycle for n iterations, S_i is the code size. The variables δ_{ik} are used to express the choice of a sequence: only one of the δ_{ik} can have the value 1 for a given value of i.

Let n_i be the number of iterations in loop L_i and W_i be the number of times the loop is executed. Assuming there is nbl loops in the code, we get the following linear expressions :

$$\begin{array}{rcl} nbCycle & = & \sum_{l=1}^{nbl} W_l \times T_l(n_l) \\ totalSize & = & \sum_{l=1}^{nbl} S_l \end{array}$$

Choosing the optimization sequences for each loop can then be obtained by minimizing $nbCycle$ with the constraint $S_{max} \geq totalSize$, S_{max} being the maximum code size allowed for the set of loops, or by minimizing $totalSize$ with the constraint $C_{max} \geq nbCycle$, C_{max} being the maximum number of cycles allowed for the loops.

Illustration Example Consider the two loops, L_1 and L_2, and the respective results for three sequences of optimization as shown on Figure 10. This leads to the following constraint system:

$$\begin{array}{rcl} nbCycle & = & W_1 \times (a_1^1 \times n_1 + b_1^1) \times \delta_{1,1} + \\ & & W_1 \times (a_1^2 \times n_1 + b_1^2) \times \delta_{1,2} + \\ & & W_1 \times (a_1^3 \times n_1 + b_1^3) \times \delta_{1,3} + \\ & & W_2 \times (a_2^1 \times n_2 + b_2^1) \times \delta_{2,1} + \\ & & W_2 \times (a_2^2 \times n_2 + b_2^2) \times \delta_{2,2} + \\ & & W_2 \times (a_2^3 \times n_2 + b_2^3) \times \delta_{2,3} \\ totalSize & = & s_1^1 \times \delta_{1,1} + s_1^2 \times \delta_{1,2} + s_1^3 \times \delta_{1,3} + \\ & & s_2^1 \times \delta_{2,1} + s_2^2 \times \delta_{2,2} + s_2^3 \times \delta_{2,3} \end{array}$$

Assuming W_1 and W_2 can only take the values 1 and 4, the optimization of code size under the maximum cycle count constraint gives the solutions presented in Figure 11. Columns L_1, L_2, $Size$ and $Cycles$, respectively, indicate the strategy used for loop 1 and 2, the size and execution time of the code corresponding to the solution. Trying to get the best performance for a given maximum size of code produces the results shown in Figure 12. As expected the execution cycles decreases as the maximum size allowed increases.

To solve the constraint system we use the *lp_ solve package* [19].

Loop 1

```
for(i=0; i < n; i++) {
    a[i] = b[i] + c[i];
}
```

	sq0	**sq1(3)**	**sq3**
a_1	8	7	5 (U = 3, min_it = 7)
b_1	0	$b(n)$	$b'(n)$
s_1	8	20	64

Loop 2

```
// outer loop
// for(j=0; j < m; j++) {
    for(i=0; i < n; i++) {
       a[j][i] = b[j][i]*s+val;
    }
//} outer loop end
```

	sq0	**sq1(3)**	**sq3**
a_2	14	10	5 (U = 4, min_it = 10)
b_2	0	$b(n)$	$b'(n)$
s_2	14	31	95

Figure 10: Data for loop 1 and loop 2 when applying the three sequences of optimization.

		$n = 9$ $nbCycle \leq C_{max} = 400$				$n = 10$ $nbCycle \leq C_{max} = 400$				$n = 100$ $nbCycle \leq C_{max} = 4000$			
W_1	W_2	L_1	L_2	***Size***	*Cycles*	L_1	L_2	***Size***	*Cycles*	L_1	L_2	***Size***	*Cycles*
1	1	sq0	sq0	**22**	198	sq0	sq0	**22**	220	sq0	sq0	**22**	2200
4	1	sq1	sq0	**34**	378	N/A (C_{max} too tight)				sq1	sq1	**51**	3876
1	4	N/A (C_{max} too tight)				N/A (C_{max} too tight)				sq0	sq3	**103**	2800

Figure 11: Result of minimizing code size under execution time constraint

		$n = 100$ $totalSize \leq S_{max} = 50$				$n = 100$ $totalSize \leq S_{max} = 100$				$n = 100$ $totalSize \leq S_{max} = 200$			
W_1	W_2	L_1	L_2	***Cycles***	*Size*	L_1	L_2	***Cycles***	*Size*	L_1	L_2	***Cycles***	*Size*
1	1	sq0	sq1	**1820**	39	sq3	sq1	**1684**	95	sq3	sq3	**1239**	159
4	1	sq0	sq1	**4220**	39	sq3	sq1	**3676**	95	sq3	sq3	**3231**	159
1	4	sq0	sq1	**4880**	39	sq3	sq1	**4744**	95	sq3	sq3	**2964**	159

Figure 12: Minimizing cycle count under code size constraint

5 Preliminary Results

To experiment GCDS we considered a set loops from the multimedia applications H263[1]. The target architecture is the VLIW architecture TriMedia from Philips.

5.1 Experiments

Figure 13 presents some of the time-consuming loops and the values for n_l the number of iterations and W_l the number of execution of the loops for a set of runs. It should be noted that the loops have a small number of iterations which emphasizes the effect of the loop remainders. According to various constraints we obtain on H263 the optimization choices given Figure 15.

6 Related Works

As stated in section 2, optimizations need to focus on critical parts of applications and thus rely heavily on profiling. A number of tools exist to provide this information. PIXIE [20] can instrument executables to count execution frequency of basic blocks. Ball and Larus reduced the number of required counters by computing a minimal spanning tree of the control flow graph in a tool called QPT [4]. They prooved their approach to be optimal. ATOM [21] provides a framework for building various tools that manipulate executable files. ATOM provides a list of standard analysis tools that includes instrumentation. *prof* and *gprof* are common UNIX utilities used to detect time consuming parts of a program. They rely on sampling techniques. A recent trend yields to incorporate instrumentation facilities into the operating system. In this approach, instrumentation is transparent to the user and re-optimization can be applied to the applications when the system is idle. Morph [23] combines system and compiler technology to automatically collect profile information. A high overhead is avoided through statistical profile sampling. A similar solution is proposed by the Digital Systems Research Center: the Digital Continuous Profiling Infrastructure (DCPI) [2].

Once critical regions are identified, transformations have to be applied. IMPACT [7] is a prototype of optimizing compiler developed at the University of Illinois. It enables

Loop 1	$n_1 = \{4, 8\}$	$W_1 = \{1137984, 574912\}$	`for (i = xa; i < xb; i++) {`
			`  d[i] = s[i] * om[i];`
			`}`

Loop 2	$n_2 = \{8\}$	$W_2 = \{568768\}$	`for (i = xa; i < xb; i++) {`
			`  d[i] += s[i] * om[i];`
			`}`

Loop 3	$n_3 = \{4, 8\}$	$W_3 = \{188888, 92480\}$	`for (i = xa; i < xb; i++) {`
			`  dp[i] += (((unsigned int)(sp[i]`
			`  + sp2[i]+1))»1)*om[i];`
			`}`

Loop 4	$n_4 = \{4, 8\}$	$W_4 = \{167784, 82936\}$	`for (i = xa; i < xb; i++) {`
			`  dp[i] += (((unsigned int)(sp[i] +`
			`  sp[i+1]+1))»1)*OM[c][j][i];`
			`}`

Loop 5	$n_5 = \{8\}$	$W_5 = \{166400\}$	`for (i = xa; i < xb; i++) {`
			`  dp[i] += (((unsigned int)(sp[i]+sp2[i]`
			`  +sp[i+1]+sp2[i+1]+2))»2)*om[i];`
			`}`

Loop 6	$n_6 = \{5\}$	$W_6 = \{114196\}$	`for (k = 0; k < 5; k++) {`
			`  xint[k] = nx[k]»1;`
			`  xh[k] = nx[k] & 1;`
			`  yint[k] = ny[k]»1;`
			`  yh[k] = ny[k] & 1;`
			`  s[k] = src+lx2*(y+yint[k])+x+xint[k];`
			`}`

Figure 13: Time consuming loops extracted from H263 : n_i is the number of iterations and W_i is the number of loop executions.

	sq0	sq1(2)	sq1(3)	sq1(4)	sq2(2)	sq3
a_1	8	6	5	5	7	3 (U = 6, min_it = 16)
b_1	0	0	{10,5}	0	0	{32,64}
s_1	8	12	16	20	13	75
a_2	9	7	6	6	10	5 (U=3, min_it = 6)
b_2	0	0	6	0	0	22
s_2	9	13	18	22	19	55
a_3	12	8	8	7	12	6 (U=6, min_it = 12)
b_3	0	0	{16,8}	0	0	{48,96}
s_3	12	16	24	28	24	121
a_4	15	10	9	9	16	6 (U=9, min_it = 18)
b_4	0	0	{18,9}	0	0	{60, 120}
s_4	15	20	28	34	31	172
a_5	15	10	10	8	17	7 (U=8, min_it = 16)
b_5	0	0	10	0	0	120
s_5	15	19	29	33	33	179
a_6	19	13	12	11	30	NA
b_6	0	13	12	33	30	Not enough
s_6	19	25	36	44	59	registers

Figure 14: Results for the H263 loops. NA indicates that the optimization sequence failed.

S_{max}	C_{max}	*Cycles*	*Size*	**Loop 1**	**Loop 2**	**Loop 3**	**Loop 4**	**Loop 5**	**Loop 6**
78	$+\infty$	182941836	78	sq0	sq0	sq0	sq0	sq0	sq0
117	$+\infty$	122527020	116	sq1(4)	sq1(4)	sq1(3)	sq1(3)	sq1(3)	sq0
156	$+\infty$	116427896	148	sq1(4)	sq1(4)	sq1(4)	sq1(2)	sq1(4)	sq1(2)

Figure 15: Solution of H263.

performance analysis of various optimization techniques and supports machine-independent optimizations as well as machine-dependent. The University of Stanford gives an alternative with SUIF [9]. SUIF builds an intermediate representation of the program and applies multiple optimizations passes on this abstraction before the code is generated. These two approaches involve a complete compiler. Larus and Schnarr suggest to directly transform an executable using EEL [15]. EEL is a library that simplifies the development of tools that aim at modifying executables. With SALTO we also chose to avoid the complexity of a full compiler and code generator, but we decided to work with assembly programs. The main drawback is the necesssity to have source files. On the other hand, assembly language contain much more information about memory alignment, name of segments, ... Further-

more the labels are available and we can guarantee an exact construction of the control flow graph in the majority of cases.

Concerning the choice of a compilation strategy, we are not aware of any previous work.

7 Conclusion

In this paper we present a new compiler strategy that addresses the issue of balancing code size with execution time in a global way on an entire application. It should be noted that GCDS is more a framework for code optimization rather than a solution. Many more optimizations or constraints not discussed in this paper can be added.

To implement this strategy we built a complete infrastructure based on the SALTO system. To our knowledge this infrastructure is unique. We have not found any other system that allows to implement all code transformations needed as well as the ability of exploring different sequences of optimizations and choosing one *a posteriori*.

References

[1] `http://www.nta.no/brukere/DVC/h263_software/`.

[2] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henziger, S. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Symposium on Operating Systems Principles*, October 1997.

[3] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[4] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *ACM Transactions on Programming Languages and Systems*, volume 16, pages 1319–1360, July 1994.

[5] F. Bodin and E. Rohou. D2.3a: Definition of the low-level/high-level interface language. Technical report, Esprit Project OCEANS Deliverable, 1997.

[6] Brian Case. Philips hopes to displace DSPs with VLIW. *Microprocessor Report*, pages 12–15, December 1994.

[7] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Nancy J. Warter, and Wen-mei W. Hwu. Impact: An architectural framework for multiple-instruction-issue processors. In *International Symposium on Computer Architecture*, pages 266–275, 1991.

[8] Franco Gasperoni. *Scheduling for horizontal systems : the VLIW paradigm in perspective.* PhD thesis, New-York University, 1991.

[9] Stanford SUIF Compiler Group. SUIF: A parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University, May 1994.

[10] Wen-mei W. Hwu, Richard E. Hank, David M. Gallagher, Scott A. Mahlke, Daniel M. Lavery, Grant E. Haab, John C. Gyllenhaal, and David I. August. Compiler technology for future microprocessors. In *Proceedings of the IEEE*, volume 83, pages 1625–1639, December 1995.

[11] Wen-mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. In *The Journal of Supercomputing*, volume 7, pages 229–248, May 1993.

[12] Daniel R. Kerns and Susan J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Conference on Programming Language Design and Implementation*, pages 278–289, 1993.

[13] M. Lam. *A Systolic Array Optimizing Compiler*. PhD thesis, Carnegie Mellon University, May 1987.

[14] Monica Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Conference on Programming Language Design and Implementation*, pages 318–328. ACM SIGPLAN, June 1988.

[15] James R. Larus and Eric Schnarr. EEL: Machine-independant executable editing. In *Conference on Programming Language Design and Implementation*, pages 291–300. ACM SIGMETRICS, June 1995.

[16] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Fourteenth Annual Workshop on Microprogramming*, pages 183–198. IEEE, 1981.

[17] E. Rohou, F. Bodin, A. Seznec, G. Le Fol, F. Charot, and F. Raimbault. SALTO: System for Assembly-Language Transformation and Optimization (`http://www.irisa.fr/caps/Salto`). Technical Report 1032, IRISA, 1996.

[18] J. Ruttenberg, G. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. *Sigplan Conference on Programming Language Design and Implementation*, May 1996.

[19] Hartmut Schwab. `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

[20] Michael D. Smith. Tracing with pixie. Technical report, Harvard university, 1991.

[21] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. In *Conference on Programming Language Design and Implementation*. SIGPLAN, 1994.

[22] Jian Wang and Christine Eisenbeis. Decomposed software pipelining. Technical Report 1838, INRIA, 1993.

[23] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System support for automatic profiling and optimization. In *Symposium on Operating Systems Principles*, October 1997.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399