International Journal of Parallel Programming, Vol. 28, No. 4, 2000

Handling Global Constraints in Compiler Strategy

Erven Rohou,[1,4] François Bodin,[2] Christine Eisenbeis,[3] and André Seznec[2]

Received July 29, 1999; revised March 13, 2000

To achieve high-performance on processors featuring ILP, most compilers apply *locally* a set of heuristics. This leads to a *potentially* high-performance on separate code fragments. Unfortunately, most optimizations also increase code size, which may lead to a *global* net performance loss. In this paper, we propose a Global Constraints-Driven Strategy (GCDS) for guiding code optimization. When using GCDS, the final code optimization decision is taken according to global criteria rather than local criteria. For instance, such criteria might be performance, code size, instruction cache behavior, etc. The performance/code size trade-off is a particularly important problem for embedded systems. We show how GCDS can be used to master code size while optimizing performance.

KEY WORDS: Compiler; optimization; instruction level parallelism; code size.

1. INTRODUCTION

For the last ten years, processors are featuring more and more instruction-level parallelism (ILP) and deeper pipelines. In order to exploit this ILP

[1] ST Microelectronics, 60 Rue Lavoisier, 38330 Mont Bonnot St. Martin, France. E-mail: Erven.Rohou@st.com.

[2] IRISA, Campus Universitaire de Beaulieu, 35042 Rennes, France. E-mail: {Bodin,Seznec}@irisa.fr.

[3] INRIA Centre de Rocquencourt, Domaine de Voluceau-Rocquencourt, BP 105, 78153 Le Chesnay, France. E-mail: Christine.Eisenbeis@inria.fr.

[4] This work was done while the first author was with IRISA.

0885-7458/00/0800-0325$18.00/0 © 2000 Plenum Publishing Corporation

and therefore to enable high-performance, numerous code optimizations have been proposed;[1–11] for instance function inlining, loop unrolling, software pipelining, etc. These transformations have been implemented in academic compiler suites such as SUIF[12] or IMPACT[13] as well as in commercial compilers.

Conventional compilers are not designed to deal with global issues such as code size and execution time. They rely on a in-built strategy that applies locally a set of heuristics on code fragments to optimize the execution time. However, most optimizations used to improve performance also increase code size,[1] therefore increase the overall instruction cache footprint of the program. Then, some code generation decision which seemed to *locally* increase performance may result in a *global* net performance loss due to an increase of instruction cache misses. In the context of embedded application the increase in instruction memory impact directly on the cost of the system. Final code production decision should be taken *globally* rather than *locally*! This has lead us to propose the new general compiler strategy presented in this paper: Global Constraints-Driven Strategy (GCDS).

As in traditional optimizing compilers, several optimization sequences for each code fragment are explored and parameters such as execution time and code size are estimated for each alternative. However GCDS differs from traditional compiler heuristics in the following direction: the choice of the precise optimization sequence for each code fragment is not decided locally after the evaluations of these alternatives, but globally according to performance or code size criteria on the overall application.

In this paper, we show how applying GCDS allows to master the code size/performance trade-off on embedded applications. GCDS will be illustrated on low-level optimizations such as scheduling,[7] unrolling,[1] superblock,[6] and software pipelining[7, 14] applied to loops. However, GCDS may also be applied to other code fragments as well as at other compiler stages (e.g., function inlining at a high-level).

The remainder of the paper is organized as follows. Section 2 briefly presents the problem of code size/performance trade-off for embedded systems. Section 3 introduces the GCDS strategy. GCDS is illustrated on the performance/code size tradeoff for compute-intensive loops since this is a major issue for DSPs and application-specific processors. Section 4 overviews our experimental framework for implementing assembly-level optimization sequences, the target architecture and the benchmarks. Section 5 presents the application of the GCDS strategy to critical loops of the benchmarks and analyzes the results. Finally, Section 6 overviews related works and Section 7 concludes with some directions for further investigations.

2. CODE SIZE/PERFORMANCE TRADE-OFF ON EMBEDDED APPLICATIONS

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).

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 overall size of the code. To achieve the first goal, the compiler must be able to explore locally many different strategies for scheduling, register allocation, superblock optimization, loop unrolling, etc. To reach the second goal, the compiler must balance code size with execution time in a global way on the whole application. The impact of each code fragment has to be evaluated on both the overall performance and the overall code size. In other words, for embedded applications, the compiler must be able to answer either of the following global 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?"

The more aggressive compilers are, the more code size is an issue to address. To guide the code optimization decision, for many embedded applications, profiling results can be gathered. It should be noted that hand written critical kernels also rely on such accurate profiling. These results may be used for weighting the impact of different code fragments on the optimization decision.

3. GCDS

Traditional compilers do not take into account global constraints on applications such as code size, real time execution, etc. They have a single in-built strategy based on locally applying a set of heuristics to hopefully minimize 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 (e.g., based on register constraints). Such compilers do not control any global parameter such as the overall code size.

A contrario, the philosophy of our Global Constraints-Driven Strategy (GCDS) is to use global information on the complete application to select the precise transformation that will be applied locally on each code fragment. GCDS is illustrated on Fig. 1.

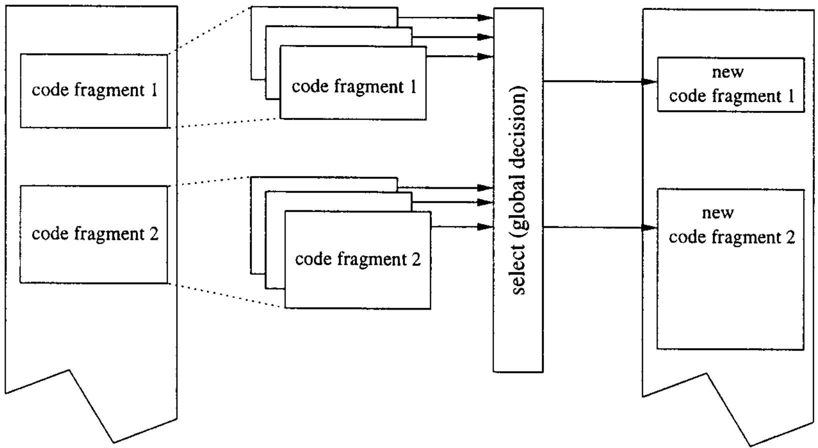


Fig. 1. GCDS overall organization.

1. For each code fragment, several code optimization alternatives leading to different implementations are analyzed. The resulting code sizes and execution times are computed/estimated.
2. A global selector function (the "select" function in Fig. 1) is used to choose a code optimization among alternatives for each code fragment. The choice depends on an estimate of the criteria such as the total execution time, code size, or any other global constraint.

3.1. Applying GCDS for Code Size/Performance Trade-Off

As already mentioned, we illustrate the GCDS principle on the performance/code size trade-off. The code optimizations that will be considered are low-level loop transformations along with function inlining. We first describe the cost model we used for loops. Then we propose a select function that can be used to trade off code size against performance.

A set of transformations is applied to each code fragment. The different results are evaluated: for each piece of code the size and the number of static cycles per loop iteration are measured. Finally, these data are used to decide which transformation must be applied to each code fragment to obtain the best possible global performance.

3.1.1. A Cost Model for Loops

For any particular instance of the code generated for a loop L, its cost and execution time is modeled as follows: a number of execution cycles per

iteration of the loop, denoted a, and an overhead cost, denoted b, corresponding to the loop epilogue and prologue. The value b may vary as a function of the number of iterations (i.e., the iteration space) of the loop. Each loop is also weighted with W the number of times it is executed. Some transformations (e.g., software pipelining) require that a minimal number of iterations min_{it} are executed. Let n be the number of iterations of the loop. Hence using this simple model, the execution time t in machine cycles spent executing loop L, is given by:

$$\text{if} \quad n > min_{it}: t = W \times (a \times n + b(n)) \tag{3.1}$$

This model considers loops without conditional branch instructions. This may not be completely accurate for loops with complex body. However it is sufficient in most cases, especially when if-statements are converted to straight line code using guarded instructions by if-conversion. The cost model can be easily extended. The current version expects the size and the speed to be linear expressions. Note that this is not a limitation of our approach, but rather a choice we made regarding the select function.

A difficulty arises when the values of W and n vary from one run to another. Variations of W do not matter as long 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. Let nbn be the number of possible values for n. For a loop L instead of (3.1) we can use:

$$\sum_{k=1}^{nbn} W^{(k)} \times (a \times n^{(k)} + b(n^{(k)}))$$

For the sake of clarity we shall assume that there is only one possible value of n per loop in the next section.

Computing a and b. The values of a and b directly depend on the particular optimization applied to a loop. In the case of local scheduling of the loop body, a is simply the number of cycles the loop body lasts after scheduling, b is zero cycle and the size $s = a$. It might be more complex when loop unrolling is involved: unrolling in our experiment is followed by superblock construction[6] and addition of guards to allow instructions to move across jumps. In this case b depends on the value of n as well as the unrolling factor u. Indeed, the entire loop body is executed in this case, so the overhead cost $b(n)$ is given by:

$$b(n) = \begin{cases} \text{if } (n \text{ modulo } u = 0) \text{ then } 0 \\ \text{else } (u - (n \text{ modulo } u)) \times a \end{cases}$$

Therefore if x code optimization alternatives are explored, we get x possible pairs of cost and execution time for the loop. This approach expresses the time spent in the loops as a system of linear equations. Our selection function has to implement the choice of the best optimization combinaiton according to this system.

3.1.2. The Select Function as a Simplex Problem

In this section we present how the global selection of code optimization 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, it is measured on actual code and expressed as a number of VLIW instructions. Let $t_i^j(n) = a_i^j \times n + b_i^j(n)$ 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 $\delta_{ik} = 1$, then for loop L_i, the optimization sequence k is the one to choose. Otherwise $\delta_{ik} = 0$. For each loop we get the following integer equations:

$$T_i(n) = \sum_{k=1}^{nbopt} t_i^k(n) \times \delta_{ik}$$

$$S_i = \sum_{k=1}^{nbopt} s_i^k \times \delta_{ik}$$

$$1 = \sum_{k=1}^{nbopt} \delta_{ik}$$

where $T_i(n)$ is the execution time in cycle for n iterations, S_i is the code size.

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 are *nbl* loops in the code, we get the following linear expressions:

$$nbCycle = \sum_{l=1}^{nbl} W_l \times T_l(n_l) \qquad totalSize = \sum_{l=1}^{nbl} S_l$$

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

3.2. Weighting Code Fragments

In the previous example we have implicitly assumed that the weights of the different code fragments are given by their respective number of occurrences and the number of loop iterations. Such numbers may be collected by profiling. This approach is clearly acceptable for embedded systems because the user (i.e., the integrator) should have a very good knowledge of its workload.

Profiling cannot be easily used during code development for general purpose applications. However, for production codes, optimizations based on extensive profiling is quite acceptable. For instance such an approach is used by the FX!32 system which optimizes the code based on the usage of each particular user.[15]

4. EXPERIMENTAL FRAMEWORK

It is our belief that, during the process of optimizing an application, the repetitive task of applying code transformations to code fragments cannot be done without a software infrastructure. This section first overviews the target architecture we used, then details the environment we developed to experiment our new strategy. Next, we present the benchmarks we used.

4.1. Target Architecture: The Philips TriMedia

The Philips TriMedia TM-1000[16] is a processor used in 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 features 27 functional units, and 128 general-purpose registers. It implements split instruction and data caches (32 and 16 Kbytes, respectively). The TM-1000 implements a VLIW[17] architecture. Up to five operations are issued on each clock cycle, at a clock rate of 100+ MHz. Every operation is guarded by a 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.

4.2. Implementation Infrastructure

Figure 2 illustrates the global organization of a compilation chain for embedded applications. Our group only focuses on the back-end part in this chain.

The low-level implementation infrastructure we have built makes use of SALTO.[18] GCDS is built on top of it and is used to evaluate different code transformations on code fragments 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.[19] [Note: As the front end is still under construction in the experiments described in this paper, the IL files were hand generated.] This file contains information that is lost during the compilation phase, such as

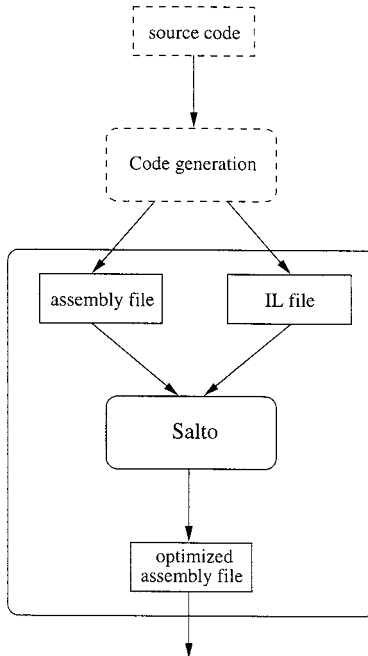


Fig. 2. Compilation Process.

dependences between memory accesses, loop iteration space and stride when available.

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/her to implement algorithms needed for performance tuning of low-level codes. This set of tools includes assembly code schedulers (e.g., software pipelines), profiling, and tracing tools. SALTO is easier to retarget than a compiler, but it does not operate on executable codes. Most of the 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 Fig. 3: a kernel (SALTO), a machine description file and an optimization or instrumentation algorithm.

- The kernel performs common house-keeping tasks the user does not want to worry about: the parsing of the assembly code and 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 a user-supplied function.

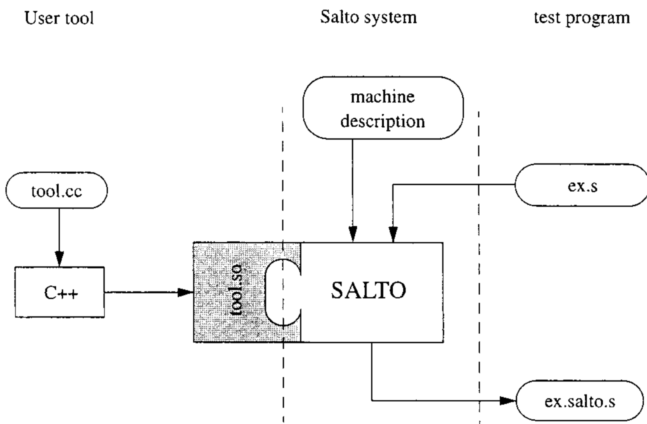


Fig. 3. Salto.

An object-oriented user interface provides an easy access to the internal representation of assembly code. The classes allow the user to manipulate each function in the form of a control-flow graph and to access to resource usage of each instruction. For further information on SALTO the reader can refer to Rohou *et al.*[18]

The usage of the specific infrastructure we built for implementing GCDS conforms with Fig. 1. A set of transformations is applied on a cloned piece of code. Then according to performance or size criteria, one of the transformations is selected for each considered loop. We used the *lp_solve*[20] package to solve the Simplex problem.

4.3. Optimizations Suite

In this paper we only consider optimizations on loops. The original assembly files were generated using the Philips compiler. The suite of optimizations we implemented includes loop unrolling, construction of superblocks,[6] scheduling of basic blocks and superblocks,[6, 21, 22] software pipelining and inlining.[4] Insertion of guards to instructions removes jumps and thus allows scheduling across jumps.[23] Software pipeline generates a modulo scheduling of the loop body. The algorithm used to compute a modulo scheduling is based on the method proposed by Wang *et al.*[14] Register allocation can be performed either before or after scheduling of the instructions. For the experiments, no variable spilling is performed when not enough registers are available.

For the purpose of this study, five optimization sequences have been tested on each loop. These transformations are summarized in Fig. 4.

- S_0 is the simplest transformation sequence. First, registers are renamed to remove as many anti-dependences as possible to improve code compaction. The code is then scheduled locally; register allocation is performed as the last step.

	S_0	U_n	U'_n	SP	I
step 1	local schedul.	unroll $\times n$	unroll $\times n$	soft. Pipeline	inlining
step 2	reg. alloc.	superblock	superblock		scheduling
step 3		scheduling	reg. alloc.		reg. alloc.
step 4		reg. alloc.	scheduling		

Fig. 4. Optimization sequences (n is the unroll factor).

- U_n is based on unrolling the loop body n times. The unrolled body is transformed into a superblock, and conditional jumps are eliminated through the insertion of guards, resulting in a large basic block. As in S_0, 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.
- U'_n is similar to U_n but register allocation is performed before scheduling. This decreases the code compaction potential, but usually requires less registers, allowing this sequence to succeed when U_n fails due to a lack of register.(24)
- *SP* consists in applying a software pipeline algorithm.(7) This sequence is limited to loops whose body is a single basic block. In measurements for *SP*, we will also report the unroll factor *U*.
- *I* causes the function calls to be inlined. The large resulting body is scheduled before register allocation is performed. Inlining results in a larger block with more potential parallelism. It can also enable further transformations prevented by the function call, for instance software pipelining.

Other optimization sequences might as well be tested. It should be noted that the code is not generated after the application of each transformation to each code fragment. Instead we only measure the values relevant for our global selection.

4.4. Benchmarks

Since the TM-1000 architecture targets multimedia applications such as wireless phones or high resolution TV, we decided to experiment our strategy on this class of applications. Both applications have the following in common: they contain a number of small loops that are very frequently executed but with a small number of iterations, typically 4 or 8.

We selected the following two benchmarks:

H263 is a new improved standard for low bit-rate video compression. We used "`tmndecode`" which is a decoder and player for H263 bitstreams from Telenor R&D. Six different bitstreams were used to collect the profiling information.(25)

mpeg2play is a decoder and viewer for MPEG2 encoded video files. We used only the decoder part to avoid taking into account the X11 libraries. The measures were done with five different video sequences.

For both applications and for most loops, the same iteration space was encountered, whatever workload is used.

5. EXPERIMENTAL RESULTS

As the front-end is not fully integrated with the back-end we were not able to get a complete propagation of memory dependences from the source code. Thus the IL-files containing these information were generated by hand. As such this effort would be unrealistic for a complete application; we chose instead to focus on the most time-consuming loops in both applications. The loops accounted on average for 40% of the running time. They are presented in Fig. 5 with the values for n (the number of iterations) and W (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. This emphasizes the impact of the loop remainders.

Despite the limited number of code fragments tested in our experiments, our approach could be applied to an arbitrarily large number of loops. Current solvers can support large problems. [For instance, the solver gives, generally, a result in a few minutes with systems up to 200 loops on an UltraSparc 200 MHz.] A survey of various linear programming softwares is available.[26]

5.1. Applying Transformations

The original assembly files were generated using the Philips compiler. Our system then tried to apply each transformation to each loop. In each case we measured the values of a, b and s as defined in Section 3.1.1. These measures are given in Figs. 6 and 7. A dash indicates that the transformation is not applicable and *failed* indicates a failure of the register allocator.

In order to further increase performance, we planned to unroll the loops of mpeg2play after inlining (only loops 1 and 2 contain a function call). Unfortunately the transformations U_2 and SP failed because of insufficient number of registers.

When the software pipeline could be applied, it gives a result with a low number of cycles per iteration. However the minimum number of iterations required (min_{it}) is greater than the effective numbers gathered by profiling. This means that in all our cases, the application would have executed the remainder loop and would never have benefited from the transformation.

5.2. Analysis

Figure 8 shows the performance achieved by the benchmarks as a function of the code size constraint. This figure was obtained by computing the number of static cycles for each code size constraint. As foreseeable the

#	n	W	C code
H263			

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

2	{8}	{568768}	for (i = xa; i < xb; i++) {
			    d[i] += s[i] * om[i];
			}

3	{4,8}	{188888,92480}	for (i = xa; i < xb; i++) {
			    dp[i] += (((unsigned int)(sp[i]
			       + sp2[i]+1))>>1)*om[i];
			}

4	{4,8}	{167784,82936}	for (i = xa; i < xb; i++) {
			    dp[i] += (((unsigned int)(sp[i] +
			       sp[i+1]+1))>>1)*OM[c][j][i];
			}

5	{8}	{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];
			}

6	{5}	{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];
			}

mpeg2play			

1	{8}	{363258}	for (i=0; i<8; i++) {
			    idctrow(block+8*i);
			}

2	{8}	{363258}	for (i=0; i<8; i++) {
			    idctcol(block+i);
			}

3	{8}	{211668}	for (i=0; i<8; i++) {
			    rfp[0] = clp2[bp[0]];
			    ...
			    rfp[7] = clp2[bp[7]];
			    rfp += iincr;
			    bp += 8;
			}

4	{8}	{151590}	for (i=0; i<8; i++) {
			    rfp[0] = clp2[bp[0] + rfp[0]];
			    ....
			    rfp[7] = clp2[bp[7] + rfp[7]];
			    rfp += iincr;
			    bp += 8;
			}

5	{1,2,3}	{977286,83741,311}	do {
			    bfr |= *ld->ptr++ << (24-incnt);
			    incnt += 8;
			} while (incnt <= 24);

Fig. 5. Time consuming loops extracted from H-263 and mpeg2play: n is the number of iterations and W is the number of loop executions.

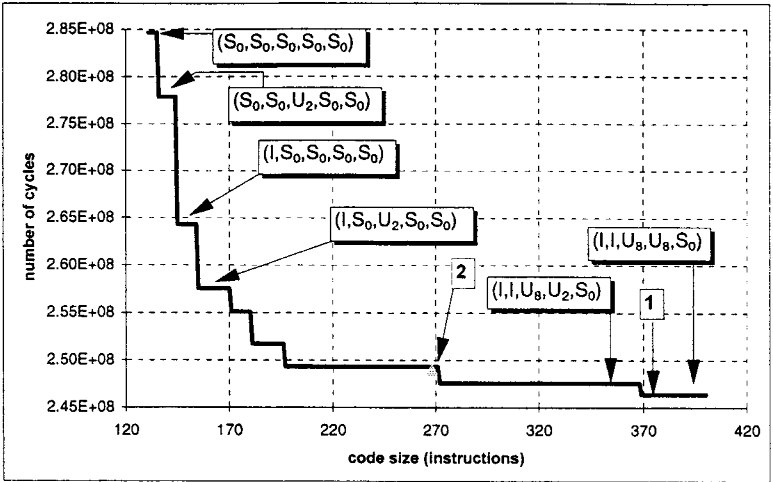
	S_0	U_2	U_3	U_4	U_2'	SP
a_1	8	6	5	5	7	3 (U = 6, min_{it} = 16)
b_1	0	0	{10,5}	0	0	{32,64}
s_1	8	12	16	20	13	75
a_2	9	7	6	6	10	5 (U=3, min_{it} = 6)
b_2	0	0	6	0	0	22
s_2	9	13	18	22	19	55
a_3	12	8	8	7	12	6 (U=6, min_{it} = 12)
b_3	0	0	{16,8}	0	0	{48,96}
s_3	12	16	24	28	24	121
a_4	15	10	9	9	16	6 (U=9, min_{it} = 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

Fig. 6. Applying transformations to H263 loops. a_i denotes the number of cycles per iteration, b_i the loop overhead and s_i the code size.

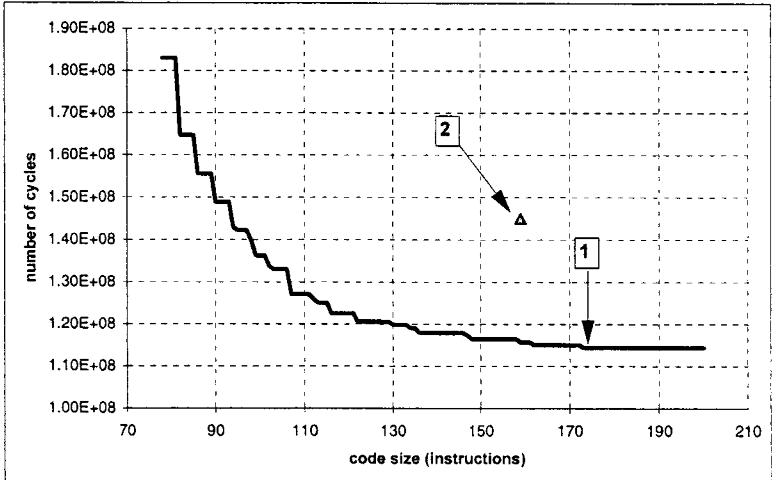
performance increases with the code size. The merit of our strategy is to provide the user (or the compiler) with a tool that takes decisions. While on a small program involving very few loops a decision could be handled manually by an expert, for large applications involving hundreds of code fragments such decision has to be supported by an automatic tool.

	S_0	U_2	U_3	U_4	U_8	U_2'	SP	I
a_1	36					42	-	29
b_1	0	failed	failed	failed	failed	0	-	0
s_1	36					83	-	55
a_2	38					48	-	36
b_2	0	failed	failed	failed	failed	0	-	0
s_2	38					95	-	64
a_3	18	14	13	14	13	67	13 (U=3,min_{it}=9)	-
b_3	0	0	13	0	0	0	48	-
s_3	18	28	39	55	103	134	124	-
a_4	20	18	18	18	17	69	16 (U=3,min_{it}=9)	-
b_4	0	0	18	0	0	0	54	-
s_4	20	36	53	69	133	138	150	-
a_5	14	11	10	9	9	24	5 (U=4,min_{it}=16)	-
b_5	0	{11,0,11 }	{20,10,0}	{27,18,9}	{63,54,45}	0	{37,46,55 }	-
s_5	14	21	29	36	65	48	85	

Fig. 7. Applying transformations to mpeg2play loops. a_i denotes the number of cycles per iteration, b_i the loop overhead and s_i the code size.



mpeg2play



H263

Fig. 8. Code size vs. performance.

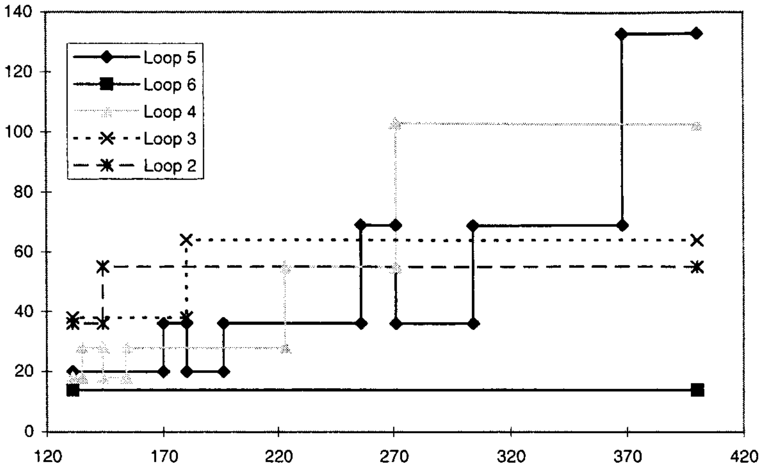


Fig. 9. Evolution of the sizes of the loops for mpeg2play. The x axis is the maximum global code size allowed. The y axis is the code size for each loop.

We studied how optimizations of different loops interact with each other. It appears that the sequence globally applied to loops when the size changes is rather tricky. We illustrate this behavior for mpeg2play in Fig. 9: each curve represents the size allocated to a loop as a function of the constraint. When the allocated space shrinks, loop 5 is first unrolled 8 times, then 4, then only 2 times. But when the size still decreases, the loop is unrolled 4 times again, to the detriment of loop 4 which drops away from U_8 to U_4. This can also be seen on Fig. 8 for loop 3 on the third component of the tuples shown for mpeg2play. This is a consequence of the global interaction of the different pieces of code with respect to the optimization. Therefore the best transformation to apply to a loop cannot be determined by only considering each loop independently. Instead, this study shows that optimization is a complex problem that can only be handled in a global fashion.

Let us try to better emphasize the efficiency of our approach compared to local strategies that do not take advantage of a global view of the program and thus take local decisions. Let us consider the following two distinct compiler strategies:

1. The first approach locally chooses the best performance among the transformations presented in the previous section according to profiling information. This strategy is equivalent to GCDS with unlimited code size. The results are represented in Fig. 8 by the label number 1.

2. The second one systematically selects the optimization that gives the highest asymptotic performance, that is the smallest a_i for each loop. It is not guided by profiling but, of course, makes use of static information when available (e.g., constant loops bounds). In order to limit code size expansion, we assume that the compiler does not choose a larger unroll factor if it does not increase performance by more than 10%. The points are plotted in Fig. 8 with the label number 2.

Since the first approach does not take code size into account, it would produce the fastest code by inlining every function and unrolling loops to the maximum degree. Although this code allows to reach the highest possible performance, it wastes much room compared to GCDS. For instance, a loss of 1% in static cycles for mpeg2play permits to divide the space allocated to the loops by a factor of 2 (from 400 to 200 words). Similarly allowing a 3% increase in cycles for H263 gives a saving of 21% in space. This phenomenon would be far more pronounced in the presence of tens or hundreds of loops.

The second approach does not rely on profiling information. This can cause a substantial loss in space or performance. For instance, loop 5 of mpeg2play will be software pipelined, since this optimization gives a much better loop speed ($a_5 = 5$). Unfortunately the value of min_{it} will never be reached at run-time and the application will always use the remainder loop. This causes space to be wasted and a nonoptimal code to be executed ($a_5 = 14$ for the remainder loop, compared to 9 if the loop were unrolled four times). Nevertheless a fast code is generated for mpeg2play, but the same performance level could be achieved with a 26% smaller code. A contrario this strategy results in neither fast nor small code for H263.

GCDS, compared to both previous strategies, emphasizes the importance of a global decision and the need for reliable profiling information. It should also be pointed out that, if the user is facing a hard code size constraint (e.g., the size of the instruction EEPROM), no local strategy can provide him/her a solution. Then he/she has to choose (maybe guided by profiling) a trade-off on which code fragments to optimize.

5.3. Using Profiling Information

Our strategy heavily relies on profiles. Wall[27] studied the predictability of a program behavior using profiles and presented somewhat pessimistic results. However his results illustrated general purpose applications. Therefore we had to check whether the decisions taken are still valid when different data sets are used for the applications.

First, we checked that the most time-consuming loops do not depend on the program input. However one must pay attention to this issue when the number of code fragments to optimize gets bigger.

The second point consists in verifying that the generated code using one profile set is still performant for other data sets. We proceeded as follows: given different inputs, each of them was used to determine an optimal sequence of optimizations and the number of cycles is measured. At the same time, we measured the number of cycles achieved for this input with the application optimized for the another set of inputs. This is repeated for each value of the size constraint. We the checked the ratio between these two values. We checked seven video with mpeg2play and six bitstreams with H263. We found that generally both inputs result in the same best found sequence and never exceeds 1.3% more cycles than the best found sequence for mpeg2play and 6% for H263.

6. RELATED WORK

Two issues are addressed by GCDS: the application of a number of transformations to various pieces of code and the global decision of best transformation for a particular fragment.

Compilers and optimizers are faced to the well-known *phase ordering* problem. Many code optimizations exist and the majority have an impact on the efficiency of the others. Several attempts have been made to integrate different phases in order to avoid interference. Bradlee *et al.* combined register allocation and instruction scheduling in MARION.[2] Goodman and Hsu[24] proposed an alternative with two techniques. Carr[28] optimized loop nests by simultaneously considering ILP and data reuse. Wolf *et al.*[29] presented a model that statically predicts the execution time of a loop nest with respect to data cache and number of instructions. The best optimization is found by traversing the possible cases. Berson *et al.*[30] integrated code generation with other transformations such as redundancy elimination and DAG restructuring when they are beneficial.

Fewer approaches consider the whole program. Fisher[31] studied scheduling according to a global execution trace. Among work that focuses on the global interaction of different optimizations applied to different code fragments, Gupta[32] used the Program Dependence Graph to distribute parallelism across code regions. Hank *et al.*[33] defined code regions to enlarge the vision of the compiler and possibly enable some optimizations.

7. CONCLUSIONS

During compilation processes, many alternative codes may be chosen by the compiler. In order to chose among these alternatives, most compilers

apply local heuristics for the trade-off between code size and performance. As global constraints on the whole application such as code size are not considered, this may lead to code size explosion and/or poor overall performance.

In this paper, we propose GCDS a new compiler strategy for addressing global issues on an entire application. The basic principle of GCDS for code generation consists in separately evaluating a set of code optimization heuristics on each code fragment and then to rely on a global select function for final code production. The complexity of the first step in this process is essentially linear with the source code size and the number of local code generation heuristics while the second step is essentially a linear integer problem resolution.

We have illustrated GCDS for low-level loop body code generation on the tradeoff between code size and execution time.

Our belief is that other issues in code optimization should be addressed globally on the whole application, and then that GCDS should be applied. Here are two such issues. Code optimization for a complete library could be addressed globally as follows: the different functions in the library might be weighted, then their respective maximum code sizes might be determined, and finally balancing code size with execution time could be done at the function level. The instruction cache usage is another issue related to code size that should be addressed globally at the application levels: sets of code fragments that should be alive at the same time in the cache might be determined, thus providing a set of constraints.

The application of GCDS is also conditioned by the ability of weighting code fragments. For embedded applications one can rely on profiling. For general purpose applications, optimization based on extensive profiling can also be efficient as recently demonstrated by the FX!32 system.[15]

ACKNOWLEDGMENT

This work was partially supported by the Esprit project OCEANS and by the French Ministry of Defense.

REFERENCES

1. David F. Bacon, Susan L. Graham, and Oliver J. Sharp, Compiler transormation for high-performance computing, *ACM Computing Surveys*, **26**(4):345–420 (December 1994).
2. David G. Bradlee, Susan J. Eggers, and Robert R. Henry, Integrating register allocation and instruction scheduling for RISCs, *Proc. Fourth Int'l. Conf. Architectural Support Progr. Lang. Operat. Syst.*, pp. 122–131, Santa Clara, California (April 8–11, 1991). ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society.

3. William Y. Chen, Pohua P. Chang, Thomas M. Conte, and Wen-mei W. Hwu, The effect of code expanding optimizations on instruction cache design, *Trans. Computers*, **42**(9): 1045–1057 (September 1993).
4. Jack W. Davidson and Anne M. Holler, Subprogram inlining: A study of its effects on program execution time, *IEEE Trans. Software Engng*. **18**(2):89–101 (February 1992).
5. Jack W. Davidson and Sanjay Jinturkar, Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation, *Proc. 28th Ann. Int'l. Symp. Microarchitecture*, pp. 125–132, Ann Arbor, Michigan, November 29–December 1, 1995. IEEE Computer Society TC-MICRO and ACM SIGMICRO.
6. 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, *J. Supercomputing*, **8**:229–248 (May 1993).
7. M. Lam, Software pipelining: An effective scheduling technique for VLIW machines, *SIGPLAN Conf. Progr. Lang. Design and Implementation*, Atlanta, ACM, pp. 318–328 (1988).
8. Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann, Effective compiler support for predicated execution using the hyperblock, *Proc. 25th Ann. Int'l. Symp. on Microarchitecture*, pp. 45–54, Portland, Oregon (December 1–4).
9. Scott McFarling, Procedure merging with instruction caches, *ACM SIGPLAN Conf. Progr. Lang. Design and Implementation*, Toronto, Canada, pp. 71–79 (June 1991).
10. Todd C. Mowry, Monica S. Lam, and Anoop Gupta, Design and evaluation of a compiler algorithm for prefetching, *Conf. Architecture Support Progr. Lang. Operat. Syst*., pp. 62–73 (October 1992).
11. B. R. Rau, Iterative modulo scheduling: An algorithm for software pipelining loops, *Proc. 27th Int'l. Symp. Microarchitecture*, pp. 63–74 (December 1994).
12. Stanford SUIF Compiler Group, SUIF: A parallelizing and optimizing research compiler, Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University (May 1994).
13. 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, *Int'l. Symp. Computer Architecture*, pp. 266–275 (1991).
14. Jiang Wang, Andreas Krall, and M. Anton Ertl, Decomposed software pipelining with reduced register requirement, Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, (eds.), *Proc. IFIP WG 10.3 Working Conf. Parallel Architectures and Compilation Techniques, PACT'95*, pp. 277–280, Limassol, Cyprus, June 27–29, 1995. ACM Press.
15. Digital Semiconductor, White paper: How DIGITAL FX!32 works. http://www.digital.com/semiconductor/amt/fx32/fx-white.html (September 1997).
16. Brian Case, Philips hopes to displace DSPs with VLIW, *Microprocessor Report*, pp. 12–15 (December 1994).
17. Franco Gasperoni, Scheduling for horizontal systems: The VLIW paradigm in perspective. Ph.D. thesis, New York University (1991).
18. 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).
19. F. Bodin and E. Rohou, D2.3a: Definition of the low-level/high-level interface language. Technical Report, Esprit Project OCEANS Deliverable (1997).
20. Michel Berkelaar, lp_solve software. Available at ftp://ftp.es.ele.tue.nl/pub/lp_solve.

21. Daniel R. Kerns and Susan J. Eggers, Balanced scheduling: Instruction scheduling when memory latency is uncertain, *SIGPLAN Notices*, **28**(6):278–289 (June 1993). *Proc. ACM SIGPLAN Conf. Progr. Lang. Design and Implementation.*
22. B. R. Rau and C. D. Glaeser, Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing, *Proc. 14th Ann. Workshop on Microprogramming*, IEEE, pp. 183–198 (1981).
23. 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, *Proc. IEEE*, **83**:1625–1639 (December 1995).
24. James R. Goodman and Wei-Chung Hsu, Code scheduling and register allocation in large basic blocks, *Int'l. Conf. Supercomputing*, pp. 442–452 (1998).
25. Karl Olav Lillevold, H263 Software. Available at http://www.nta.no/brukere/DVC/h263_software/ (1995) Copyright © 1995 Telenor R&D.
26. Robert (4er@iems.nwu.edu) Fourer and John W. (ashbury@skypoint.com) Gregory, Linear Programming FAQ, World Wide Web http://www.mcs.anl.gov/home/otc/faq/linear-programming-faq.html, Usenet sci.answers, anonymous FTP /pub/usenet/sci.answers/linear-programming-faq from rtfm.mit.edu (1997).
27. David W. Wall, Predicting program behavior using real or estimated profiles, *Conf. Progr. Lang. Design and Implementation*, pp. 59–70 (June 1991).
28. Steve Carr, Combining optimization for cache and instruction-level parallelism, *Proc. Conf. Parallel Architectures and Compilation Techniques (PACT'96)*, pp. 238–247, Boston, Massachusetts (October 20–23, 1996). IEEE Computer Society Press.
29. Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen, Combining loop transformations considering caches and scheduling, *Proc. 29th Ann. Int'l. Symp. Microarchitecture*, pp. 274–286, Paris, France (December 2–4, 1996). IEEE Computer Society TC-MICRO and ACM SIGMICRO.
30. D. A. Berson, P. Chang, R. Gupta, and M. L. Soffa, Integrating program optimizations and transformations with the scheduling of instruction level parallelism, *Lecture Notes in Computer Science*, **1239** (1997).
31. J. A. Fisher, Trace scheduling: A technique for global microcode compaction, *IEEE Trans. Computers*, pp. 478–490 (July 1981).
32. R. Gupta and M. L. Soffa, Region scheduling: An approach for detecting and redistributing parallelism, *IEEE Trans. Software Engng*. **16**(4):421–431 (April 1990).
33. Richard E. Hank, Wen-mei W. Hwu, and B. Ramakrishna Rau, Region-based compilation: An introduction and motivation, *Proc. 28th Ann. Int'l. Symp. Microarchitecture*, pp. 158–168, Ann Arbor, Michigan (November 29–December 1, 1995). IEEE Computer Society TC-MICRO and ACM SIGMICRO.