Overcoming the Limitations of the Traditional Loop Parallelization

Ireneusz Karkowski and Henk Corporaal

Faculty of Information Technology and Systems, Delft University of Technology
Mekelweg 4, PO Box 5031, 2600GA Delft, The Netherlands

Abstract

Previous research has shown existence of a huge potential of the coarse-grain parallelism in programs. This parallelism is however not always easy to exploit. Especially, when applying today's parallelizing compilers to typical applications from the "embedded" domain. This is mainly due to the deficiencies of the static data dependency analysis they relay on. This paper investigates the potentials of the loops parallelization techniques using dynamic loop analysis techniques. For a set of "embedded" benchmarks (including an MPEG-2 encoder) ~ 4 times more loops could be parallelized, in comparison with a state-of-the-art compiler (SUIF [1]), leading to average speedups of 2.85 (on a 4 processor system). Dynamic analysis is however not "full-proof" - we intent to use it exclusively in cases when static analysis fails to give any answer, and only if the user asserts its applicability.

Key words: multiprocessing, loop parallelization techniques, data dependency analysis, high performance embedded system design

1 Introduction

As the application area of the embedded processors widens, the demands on their performance are constantly growing. Until now, instruction level parallelism has been successfully exploited to satisfy these high performance requirements. Practice shows however that increasing the number of slots in the instructions of the typical VLIW architectures above a certain level does not necessary lead to significant performance gains. Instead, high hardware costs and inefficient use of this hardware occur. The advent of sub-micron processing, allowing integration of several millions of transistors on a single carrier, has brought new opportunities in the embedded system design. A multiprocessor embedded system becomes nowadays a very interesting alternative. This both in terms of the hardware cost and performance. Especially, if the system consists of several (different) ASIP processors, each with functionality optimized for the subtasks which they have to perform.

Preprint submitted to Elsevier Preprint *31 March 1999*

Code partitioning among the processors leads then to exploitation of the course-grain parallelism (task parallelism and data parallelism in loops [3,8]), while the fine-grain (instruction level) parallelism [7] is exploited locally by each of the processors.

The most straight-forward code generation approach would be to use one of the existing parallelizing compilers, meant for symmetric shared-memory multiprocessors [1,14]. Such a compiler takes as input sequential code written in a high-level language and automatically generates a parallel executable. The code parallelization is obtained by applying a set of transformations to the FOR loops of the program, and partitioning of their iteration space among the processors (in other words exploitation of data parallelism in loops). To verify if the transformations and the partitioning is legal, static data dependency analysis (further called static DDA) is applied. For typical numerical applications this works very well. Unfortunately for the "embedded" class of programs this is often not the case. The reason for this lies mainly in the nature of the static data dependency disambiguation techniques. Meant to be used in fully automatic, general purpose compilers, they must deliver absolutely "safe" results. If unable to prove that some statements are independent, the static DDA must therefore conservatively assume them dependent [5]. While usually not critical, in extreme cases, a single conservatively assumed dependence may result in a failure to effectively parallelize a program.

The number of cases when conservative dependence assumptions have to be taken varies for different application domains. The "embedded" programs, for instance, are almost exclusively written in the C language. Convoluted programming style, intensive use of pointers, distribution of calculations among a deep call hierarchy, complex control flow and data structures are common. This together makes the static dependency analysis tedious and leads to many conservative assumptions.

The dynamic data dependency analysis, on the other hand, is not "full-proof". Absolutely no compiler can depend on it. There are however several major differences between a general purpose multiprocessor computer and a multiprocessor "embedded" system. The first one will probably run under a multi-user, multi-tasking operating system. Since the processors of such a system can always be kept busy with many other user/system tasks, our failure to effectively parallelize the program, will not lead to inefficient hardware use. For typical "embedded" systems the situation looks however completely different. Often they run just a single program, which is compiled only during the system development. One of the fundamental questions that a hardware/software co-designer [9] of such system has to answer is: Can the embedded software be parallelized to a degree, which justifies the use of a multi-processor framework? If the significant performance improvements can be obtained, long data dependency analysis times (needed to apply long enough input data sets), and even user interaction (based upon compiler analysis feedback), are well justified. Therefore we decided to use dynamic DDA (in addition to static DDA) as an option within our software framework.

To verify the usefulness of the dynamic analysis in effectively parallelizing typical embedded programs we implemented a prototype of a dynamic DDA system. Our experiments with it show that indeed, if we apply dynamic instead of static DDA, about 4 times more loops in our set of benchmarks may be parallelized, usually leading to significant performance gains.

The remainder of the paper is organized as follows. Section 2 is devoted to the presentation of our dynamic DDA methodology. Experimental results are presented in section 3. In section 4 we discuss failures of the static DDA, which we have observed and explain how we handle the deficiencies of the dynamic DDA. Section 5 concludes this paper.

2 Methodology

The dynamic data dependency information can be calculated by tracing data dependency at runtime. Since we are interested in full dependency information, including values of the distance vectors, the trace stream should contain additional information about the current position within the program and about the values of the loops' index variables. This is difficult, if we instrument binaries (like most tracing systems do [10]). Instead, we decided to instrument the source code. We take an arbitrary "C" sources, insert trace calls and compile the resulting sources with the system's C compiler. This has the advantage of being extremely portable ("traditional" tracing systems are able to instrument only binaries in a certain format). Figure 1 shows our dynamic DDA system. It has been developed on Sun Sparc 20 workstation (4 CPU's) running under Solaris ver. 2.4. All programs have been written in C++ and compiled with GNU C++ compiler. The two main components, **tr_cc** and **dda** are described next.

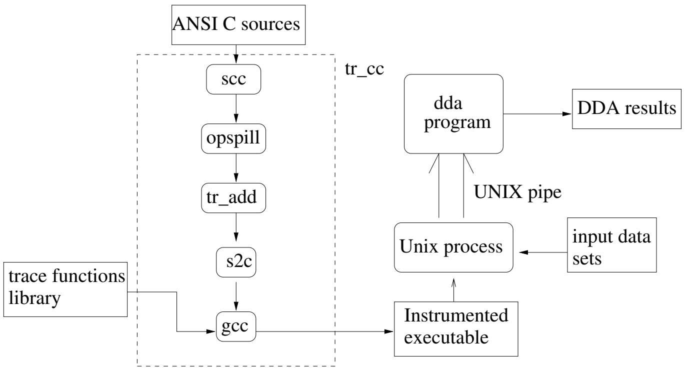

Fig. 1. The dynamic DDA system overview.

Code instrumentation

The **tr_cc** program is based on the SUIF C compiler *scc*, which proved to be a very good and convenient research platform. C sources are first compiled and initially optimized by its standard passes. We wrote two extra passes. Both operate on the SUIF internal representation. The first pass spills all operands, being function calls, to temporary variables.

This is necessary to properly trace the results of the function calls without calling the function twice (the call might have some side effects, what could result in changed functionality). The second pass adds trace calls for all memory accesses, procedure entry/exit, loop entry/exit and changes of the loop index variables.

Dynamic Data Dependency Analysis

The **dda** program calculates the detailed data dependency information, taking the generated program trace as input. Let us start with an example. Consider the C code in figure 2. Clearly, within the **for** loop f_2 there is a true data dependency from loop

```
int A[10];
main(){
  for(i=1;i<10;i++){   /*f1*/
      proc1(i);        /*st1*/
      ...
  }
}
proc1(int i){
  for(j=0;j<10;j++)    /*f2*/
      for(k=0;k<10;k++)/*f3*/
          proc2(i);    /*st2*/
      proc3();         /*st3*/
}

proc2(int i){
  for(n=0;n<10;n++){ /*f5*/
    if(i%2==0)
       A[n]=...;       /*st4*/
    else
       ...=A[n];       /*st5*/
    B[n]=...;          /*st6*/
  }
}
proc3(i){
  for(n=0;n<10;n++)    /*f4*/
     ...=B[n];         /*st7*/
}
```

Fig. 2. Example program.

f_3 to st_3 (via array **B**) with distance vector $\vec{d} = [0]$. Another true dependency goes from st_1 to itself in the procedure **main()**, with $\vec{d} = [1]$ (via array **A**). Note that there is no such dependence within **for** statements f_3 and f_5. To calculate them, for every memory access we should remember not only the place where it happened, but also a detailed execution stack which lead to it; this includes all nested loops (with the current index values) and the call statements. In order to explain the calculation of distance vectors the following terminology is introduced.

Definition 1 *A* context tree $CT(p)$ *of a program* p *is a tree with three kinds of vertices (procedure, loop and call/access statements). There is an edge between two vertices if they are nested within each other, or if one calls another. The* root *vertex of* $CT(p)$ *is the procedure* $main()$. *Complex statements are mapped into a set of vertices, each for one access.*

Definition 2 *A* context *is a path in* $CT(p)$ *from the* root *vertex to any node. Let* $S(a)$ *denote the statement where access* a *takes place. A* context $C(a)$ *of an access*

a, is a path in $CT(p)$ from the root *vertex to the statement node $S(a)$. A* procedure (loop) context *is a context ending at a procedure (loop) vertex. A context consists of one or more* procedure sections*. A* procedure section *is a longest sub-path in context C containing only one procedure vertex with which it starts.*

In figure 2 the read access a (e.g. $A[2]$) in procedure **proc2**() has: $C(a) = (main\ \ f_1\ st_1\ proc1\ f_2\ f_3\ st_2\ proc2\ f_5\ st_5)$, and its three *procedure sections* are $S_1 = (main\ f_1\ st_1)$, $S_2 = (proc1\ f_2\ f_3\ st_2)$ and $S_3 = (proc2\ f_5\ st_5)$. The program may of course contain recursive routines. A context for an access in a recursive routine $rproc()$ at recursion level of 3 might look like: $C(a) = (\ldots\ rproc\ st_{23}\ rproc\ st_{23}\ rproc\ st_{18})$ [1] .

Definition 3 *The* iteration vector chain *$IVC(a)$ of access a is a list with iteration vectors for all* procedure sections *in $C(a)$, at the moment of the access a. If a section does not contain any loops, then the list contains $[\emptyset]$, where $[\emptyset]$ is a vector of dimension* 0.

For the read access in **proc2**(), st_5 we could have: $IVC(a) = ([3], [4, 1], [5])$. Note, that there can be many *iteration vector chains* for the same context.

Definition 4 *The* virtual point *of an access a is a pair $VP(a) = (C(a),\ IVC(a))$.*

The algorithm in figure 3 can now be used to calculate true dependencies in the program. Example 5 illustrates this algorithm.

Example 5 Calculation of the distance vectors. *Recall the program from figure 2. Consider access a_r in $proc2()$ for loop indices values $i = 3, j = 2, k = 3, n = 1$. Statement st_5 reads $A[1]$. The* virtual point *of this a_r access is $VP(C(a_r), IVC(a_r))$, where $C(a_r) = (main\ f_1\ st_1\ proc1\ f_2\ f_3\ st_2\ proc2\ f_5\ st_5)$, $IVC(a_r) = ([3], [2, 3], [1])$. The virtual point of the last write a_w to $A[1]$ is $VP(C(a_w), IVC(a_w))$, where $C(a_w) = (main\ f_1\ st_1\ proc1\ f_2\ f_3\ st_2\ proc2\ f_5\ st_4)$ and $IVC(a_w) = ([2], [10, 10], [1])$. The longest common sub-path of $C(a_r)$ and $C(a_w)$ can be found to be $C_L = (main\ f_1\ st_1\ proc1\ f_2\ f_3\ st_2\ proc2\ f_5)$. Next we find $C_m = (main\ f_1)$. Note, that the longer procedure context $C_m^{'} = (main\ f_1\ st_1\ proc1)$ does not satisfy the requirement since its first* procedure section *contains different iteration vectors in $IVC(a_r)$ and $IVC(a_w)$ ($[3] \neq [2]$). For the* procedure section *in C_m we retrieve vectors $\vec{r} = [3]$ and $\vec{w} = [2]$. The distance vector is $\vec{d} = [3] - [2] = [1]$. We add this vector to the list of the dependencies of statement st_1 in procedure $main()$.*

Very similar algorithms can be devised for calculation of *anti* and *output* dependencies. Implementation of the algorithm (fig.3) involved solving several software engineering problems. The most important was that of the memory efficiency. The following optimizations were applied:

[1] A practical implementation will however only store and consider a limited number of recursion levels.

For every memory location store the *virtual points* of the last read and write accesses. At every read access a_r do:

(1) Determine the memory location m affected by a_r.
(2) Find where the last write access a_w to the same memory location m took place (in other words retrieve $VP(a_w)$).
(3) Retrieve from the $VP(a_w)$ and $VP(a_r)$ the values of the *iteration vectors chains* $IVC(C(a_w))$ and $IVC(C(a_r))$.
(4) Determine context C_L being the longest common sub-path of $C(a_w)$ and $C(a_r)$ in the *context tree* $CT(p)$.
(5) Find context C_m, being the longest procedure or loop sub-context of C_L such that all but its last *procedure section* have the same iteration vectors in $IVC(C(a_w))$ and $IVC(C(a_r))$.
(6) Retrieve from $IVC(C(a_w))$ and $IVC(C(a_r))$ iteration vectors $\vec{w}$ and $\vec{r}$ for the last *procedure section* in C_m. If $dim(\vec{r}) > 0$ then calculate the dependence distance $\vec{d} = \vec{r} - \vec{w}$, otherwise set $\vec{d} = [\emptyset]$.
(7) Set st_1 to be a vertex in $C(a_w)$ directly following C_m and st_2 to be a vertex in $C(a_r)$ directly following C_m. Add a new dependence from the statement st_1 to st_2 with distance vector $\vec{d}$.

Fig. 3. The algorithm to calculate the dependence vectors.

(1) We store only the last read and write access to every memory location. It would be prohibitively expensive to store all reads since the last modification. The consequence of this choice is that the algorithm will fail to calculate all existing *anti* dependence vectors, between any pair of statements. Only a vector with the minimal distance will be found. Fortunately only *true* dependencies inherently limit parallelism [14].
(2) It is clear that storing the $VP(a)$ for every read and write to all memory locations in the program would quickly result in serious memory consumption problems. To avoid them we maintain two special data structures. One implements the *context tree* $CT(p)$, the other all occuring *iteration vector chains*. Thanks to that, the $VP(a)$ can be implemented as a pair of pointers to the appropriate objects within both data structures.
(3) The memory usage can be additionally controlled by selecting the size of smallest memory chunk for which we record $VP(a)$ separately.
(4) Many VP's of, especially stack allocated objects, can be removed once a context is left.

Thanks to all these optimizations we are able trace programs working on for example 2MByte arrays, and with 1-byte access accuracy, without any problems.

Benchmark	Description	Operations	Lines	Input data
arfreq	Autoregressive freq. estim.	15M	367	audio sample
g722	Adaptive differential PCM	13M	891	audio sample
instf	Frequency tracking	3M	436	audio sample
interp3	Sample rate conversion	4M	504	audio sample
mulaw	Speech compression	440K	207	audio sample
music	Music synthesis	50M	321	audio sample
radproc	Doppler radar processing	32M	387	audio sample
rfast	Fast convolution using FFT	3M	559	audio sample
rtpse	Spectrum analysis	2M	388	audio sample
mpeg2enc	MPEG-2 encoder	5G	23K	50 video frames

Table 1
Benchmark characteristics. Source size (lines) is given excluding header files and library code.

3 Experimental results

To verify the effectiveness of the dynamic DDA system in parallelizing typical DSP applications, and to compare it with the static analysis, we ran it on the set of ASP (Audio Signal Processing) programs taken from [6] and on the MPEG-2 video encoder program from SSG [12]. Table 1 presents the characteristics of all tested programs. Most audio programs were slightly modified, so that the input and output data is read/written as whole blocks into input/output buffers. In case of $mpeg2enc$ it was not necessary, since the program naturally works on whole image frames. The sources obtained in this way were used as input to both static and dynamic DDA analyzers.

First, we compiled all the programs with the **pscc** parallelizing compiler, being part of the SUIF system, version 1.1.2. This latest available release contains only the baseline static DDA. After that, all the sources were compiled again using the **tr_cc** program. We ran instrumented executables to obtain dynamic data dependency vectors. Since our research is mainly oriented towards generation of the multiprocessor systems on basis of the MOVE (transport triggered architecture) ASIP processors [4], we compiled all the programs also with the MOVE C compiler, then scheduled the sequential code to exploit the instruction level parallelism (assuming oversized machine configuration). The resulting MOVE executable was simulated with the MOVE simulator. In this way we obtained exact cycle counts for all basic blocks within the programs. Next, we investigated how many of the **for** loops could be parallelized.

Table 2 presents the obtained results. We observe that dynamic DDA allows about 4 times more loops to be parallelized than static dda. However, counting only the number of parallelized loops does not necessarily give a complete picture. Therefore table 2 contains other metrics as well. Many candidate loops are nested within each other and spread among called procedures. Since our goal is the exploration of the coarse-grain parallelism, in most cases only the outermost, possible to parallelize, loop in the outermost procedure is selected for parallelization. Therefore we present also the number of these loops. In addition, the results may depend on the *granularity of parallelism*, here defined as the maximal execution time of parallel region (total time spent in the parallelized loop). From the results it is clear that the number of parallelizable loops and their average *granularity* is much better when using dynamic DDA. This is a very important since frequent synchronization on the boundary of fine-grain parallel computation may lead to a slow down instead of the expected speed up. Finally, the speedup is intended as an estimation of the overall effectiveness of a parallel system. Unfortunately it is highly machine dependent. To measure the program speedup obtained by the loop parallelization, we assumed therefore an idealized execution model with zero communication and synchronization costs, but with limited number of processing units. The loop iteration space is always equally divided between these processors.

For the sake of a fair comparison, in our experiments, we limited ourselves to the index set partitioning. Many more loops could be parallelized if we used another execution model, as for example *a doacross* model [10]. Our average speedup on 4 CPUs is 2.85. This is much more in comparison with the SUIF compiler (average speedup of 1.02). This very low speedup is mainly caused by the low count and *granularity of parallelism* of the, by SUIF, parallelized loops.

4 Analysis

In figure 4 the domain of all reference pairs in a program is presented. The area on the right hand side represents all truly independent pairs, while the area on the left all truly dependent ones. The dynamic analysis will never claim that a pair is dependent if it is not. It might however fail to recognize some truly dependent pairs, if for example, some paths within the program are never triggered by used input data (white subregion on the left). The dark gray subregions - represent subsets of pairs not properly recognized by the static DDA. Simply for some truly dependent and independent pairs the static analysis fails to give any answer. Existing compilers conservatively assume them dependent. In case of truly dependent pairs it is correct (dark gray subregion on the left), but some truly independent pairs are erroneously determined to be dependent (the dark gray subregion on the right). Of course we would like to minimize these subregions; especially the right dark gray one - because its size hinders the performance gains, which we can obtain; and the white one - to avoid obtaining programs with changed functionality.

Bench-mark	Total # loops	Total clock cycles	Dynamic # par. loops	#outer-most par. loops	Max. cycles in par. loop	Max. speedup with 4 procs	Static # par. loops	Max. cycles in par. loop	Max. speedup with 4 procs
arfreq	3	14.97M	2	1	14.96M	3.979	0	0	1
g722	13	12.79M	11	11	1.68M	1.56	6	260K	1.01
instf	14	3.33M	5	1	3.33M	3.999	3	636K	1.17
interp3	4	4.16M	3	3	3.64M	2.909	2	84	1.00003
mulaw	1	440K	1	1	290K	1.97	0	0	1
music	4	49.63M	4	1	48.54M	3.75	1	7280	1.00011
radproc	13	31.89M	6	2	31.81M	3.97	3	2.30M	1.099
rfast	12	3.54M	5	5	1.51M	1.51	0	0	1
rtpse	14	2.32M	6	6	174K	1.09	4	174K	1.08
mpeg2enc	119	5.13G	95	18	5.07G	3.86	15	126M	1.026
Total	195		138	51			34		
Average						2.85			1.02

Table 2
The parallelization results obtained using static and dynamic DDA.

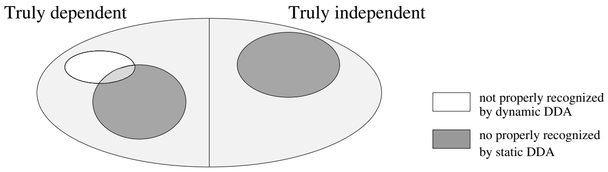

Fig. 4. The domain of all reference pairs.

Analysis of the dark gray region To provide a better insight into the results from table 2, in this section we analyze different types of the static DDA system failures, observed in our benchmarks. We can divide them into the following categories:

- *Memory disambiguation*. These failures are due to the memory disambiguation restriction in [11]. The static DDA algorithm uses a slightly different definition of true dependence (*A read is truly dependent on a write if the location referred to by the read is the same location referred to by the write in an earlier iteration*). This gives rise to an increase in number of calculated dependencies, since a read is considered dependent on many previous writes (while it actually only reads the last written value). The failures of this kind can be further divided into dataflow, dynamic or harmless flow (see [5] for more details). An example of a memory disambiguation failure is shown below:

```
for(i=1;i<n;i++){
   for(j=1;j<k;j++)
      A[j]=...; /* st1 */
   for(j=1;j<k;j++)
      ...=A[j]; /* st2 */
}
```

Consider the outermost loop. Clearly there is no loop carried true dependence from st_1 to st_2. In every iteration the array A is reinitialized and then used. The locations within A have however been written in all previous iterations of the outermost loop. The memory disambiguation system of SUIF will therefore deliver several loop carried dependencies from st_1 to st_2.

- *Affine failures*. The static analysis system is able to handle only data dependence problems having loops whose bounds are integer linear functions of more outwardly nested loop variables, and array reference functions which are integer linear functions of the loop variables. Among the affine failures the most common are indirect array references, nonlinear expressions, symbolic references and dynamic dependencies. As example, iterations of the loop containing statements (taken from the $mulaw$ benchmark):

```
if(j > 0x1fff)   j = 0x1fff;
   k = invmutab[j >> 1];
if(i >= 0)   k |= 0x80;
   out = mutab[k];
```

will be assumed dependent, while the dynamic system shows that they are not.
- *Interprocedural*. Often the loop body contains a function call. Because interprocedural DDA is not applied, parallelization of the whole loop has to be aborted.
- *C pointers*. If any of the statements within a loop contains C pointers the baseline static DDA will usually fail (with exception of very simple cases), since it is unable to properly disambiguate pointer references.
- *Anti/output dependencies in array references*. Anti/output dependencies can be removed by array privatization and reduction techniques. This is not really a shortcoming of the static DDA; however the baseline SUIF is unable to perform these transformations.
- *Other*. To this category belong all loops containing often used C expressions which are too complicated to be handled by static DDA. Look for example at the following for loop (the $g722$ benchmark):

```
for(i = 0;i < 6; i++) {
   bli[i] = (int)((255L*bli[i]) >>
8L);
}
```

Apparently, expressions containing type conversions or bitwise operations are not supported by SUIF parallelizer.

Table 3 presents the distribution of the types of failures in our benchmarks. We observe that the memory disambiguation failures are rare. This was caused by the fact that most loops in the benchmarks contain just a single, or even no, array references. The other failure categories are roughly equally important. As can be seen almost $2/3$ of all cases were caused by at least the presence of pointers. More detailed analysis shows however that most parallelization failures are not exclusively caused by only one type of failure. The last row in the table shows the number of loops with only one type of parallelization failure. There are only 21 of these loops.

The interesting question arises: How many of the failures would be avoided if the static DDA was able to perfectly disambiguate pointers, handle function calls and do array privatization and reduction (the next, not yet available, version of SUIF does it). The answer in our case is - only 29%. This again assures us that our choice of dynamic DDA as the vehicle for maximal exploitation of the parallelism deserves attention.

	Failures						
Benchmark	Total	Memory disambiguation	Affine	Inter-procedural	C pointers	Anti/ output deps	Other
arfreq	2			1	2	1	
g722	5				4	1	4
instf	2				1	2	
interp3	1				1		
mulaw	1		1	1		1	
music	3			3	1	1	
radproc	3			2		2	
rfast	5	1		1			4
rtpse	2			2		1	
mpeg2enc	80	4	53	19	59	31	39
Total	104	5	54	28	68	40	47
Excl.		0	0	10	6	2	3

Table 3
The static DDA failures.

Handling the "white" region Recall that the "white" region in figure 4 represents all reference pairs in the program not properly recognized by the dynamic DDA. The failures may be caused by the use of a not-representative input data during the analysis. It is impossible to guarantee that the results of the dynamic analysis are safe. Obviously, this is very dangerous since not considering some dependencies may lead to incorrect programs. We decided therefore to use the following overall policy:

- We plan to implement an interactive system for code transformations (similar to [2]). Every time when code transformations are attempted, we will relay on the dynamic DDA only when the static DDA fails to give any answer. If according to the dynamic DDA a transformation is legal (no conflicting dynamic dependencies found), the operation is attempted. In such case however the user's confirmation is always required. Alternatively, the user may opt to change the source code in such a way that static independence can be proven.
- During analysis the user is warned if some parts of code are not triggered at all, or triggered much less often than the surrounding statements. This is an indication that not all possible dependencies have been observed.
- The dependency vectors calculated for different input data sets can be merged. In this way the covering is improved. For example, the **MPEG-2** encoder contains several dedicated routines for different types of input video sequences (field or frame pictures). To obtain complete results the program has to be analyzed with both kinds of inputs and the resulting dependency information merged.
- Optionally extra code is added to perform run-time dependency check [13].

5 Conclusions and future research

In this paper we compared static and dynamic data dependency analysis methods. We have shown that the use of the dynamic DDA as the basis of a parallelization system may lead to major performance improvements. For our set of benchmarks about 4 times more loops could be parallelized, giving average speedup of 2.85 on a 4 processor system (considering loop index partitioning only). A state of the art parallelizing compiler, SUIF version 1.1.2, achieved only an average speedup of 1.02 on the same set of benchmarks. We conclude that the dynamic analysis used in cases where static analysis fails to give any answer can dramatically improve the effectiveness of a system for multiprocessor embedded system design.

In the future we plan to implement a general C code transformation system. The tools will allow interactive code transformations, leading to effective code parallelization. It is not our intention to generate parallel code for the symmetric multiprocessors only. Configurations with several (different) ASIP processors, each with functionality optimized for the subtasks which they have to perform, seem to be much more interesting from our point of view. We hope to obtain coarse-grain code parallelization by combined exploitation of task parallelism and data parallelism in loops. To maximize the performance gains, we plan to use advanced synchronization and data communication schemes, possibly combined with run-time disambiguation.

References

[1] Saman P. Amarasinghe, Jennifer M. Anderson, Christopher S. Wilson, Shin-Wei Liao, Brian R. Murphy, Robert S. French, Monica S. Lam, and Mary W. Hall. Multiprocessors From a Software Perspective. *IEEE micro*, pages 52–61, June 1996.

[2] Aart J.C. Bik. A Prototype Restructuring Compiler. Technical Report INF/SCR-92-11, Utrecht University, Utrecht, the Netherlands, November 1994.

[3] Henk Corporaal. *Transport Triggered Architectures; Design and Evaluation*. PhD thesis, Delft Univ. of Technology, September 1995. ISBN 90-9008662-5.

[4] Henk Corporaal and Hans Mulder. MOVE: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albuquerque, November 1991.

[5] J. L. Hennessy D. E. Maydan and M. S. Lam. Effectiveness of Data Dependence Analysis. *Proceedings of the NSF-NCRD Workshop on Advanced Compilation Techniques for Novel Architectures*, 1992.

[6] P.M. Embree. *C Language Algorithms for Real-Time DSP*. Prentice-Hall, 1995.

[7] Jan Hoogerbrugge. *Code generation for Transport Triggered Architectures*. PhD thesis, Delft Univ. of Technology, February 1996.

[8] Jeroen Hordijk and Henk Corporaal. The Impact of Data Communication and Control Synchronization on Coarse-Grain Task Parallelism. In *Second Annual Conf. of ASCI*, Lommel, Belgium, June 1996.

[9] I. Karkowski and R.H.J.M. Otten. An Automatic Hardware-Software Partitioner Based on the Possibilistic Programming. In *Proceedings of the ED&TC Conference*, Paris, March 1996.

[10] James R. Larus. Loop-Level Parallelism in Numeric and Symbolic Programs. *IEEE Transactions on Parallel and Distributed Systems*, 7:812–826, 1993.

[11] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and Exact Data Dependency Analysis. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 1–14, June 1991.

[12] MPEG Software Simulation Group, http://www.mpeg.org/index.html/MSSG/#source. *MPEG-2 Video Codec*, 1996.

[13] Alexandru Nicolay. Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies. *IEEE Transactions on Computers*, 38(5), May 1989.

[14] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.