Scalable and Modular Scheduling

Paul Feautrier

LIP, Ecole Normale Superieure de Lyon
69364 Lyon Cedex 07, France
Paul.Feautrier@ens-lyon.fr

Abstract. Scheduling a program (i.e. constructing a timetable for the execution of its operations) is one of the most powerful methods for automatic parallelization. A schedule gives a blueprint for constructing a synchronous program, suitable for an ASIC or a VLIW processor. However, constructing a schedule entails solving a large linear program. Even if one accepts the (experimental) fact that the Simplex is almost always polynomial, the scheduling time is of the order of a large power of the program size and of the maximum nesting level of its loops. Hence the method is not scalable. The present paper presents two methods for improving this situation. Firstly, a big program can be divided into smaller units (processes) which can be scheduled separately. This is *modular scheduling*. Second, one can use projection methods for solving linear programming problems incrementally. This is especially efficient if the dependence graph is sparse.

1 Introduction

One of the challenges in the design of embedded system is to devise methods for the automatic or semi-automatic construction of application-specific devices from a behavioral specification.

I only consider here the case of compute intensive systems, which are mostly found in signal processing applications (audio and video processing, radar software, telephony, etc.). Here the computing time cannot be neglected, the amount of data is huge, and the need for safety is not stringent. At present, applications (or parts thereof) in this field are first modeled in very high level languages (mostly, Matlab), then mapped by hand on a variety of architectures, and then implemented in a mixture of medium level code (C), assembly code and hardware specification languages like VHDL. The design process is lengthy, complex, error-prone, and does not lend itself to the exploration of the solution space.

The aim of this paper is to sketch another approach, in which the application is specified as a system of communicating processes, each process being written in a medium-level language like C. I will explain how such a specification can be converted to a synchronous program, suitable for instance for a VLIW processor or as a first step in the design of an ASIC. One begins by constructing a schedule, which gives the instant at which each operation in the program is executed. The problem of regenerating a program from a schedule has been first studied by

A. Pimentel and S. Vassiliadis (Eds.): SAMOS 2004, LNCS 3133, pp. 433–442, 2004.
© Springer-Verlag Berlin Heidelberg 2004

Irigoin [1] and considered by many other scholars. Very efficient solutions (with associated software) [2, 3] are available today.

The aim of this paper is to propose two methods for applying scheduling to large applications. The first method consists in modifying the basic scheduling algorithm to achieve better scalability. In the second method, I investigate under which conditions a program can be divided into modules which can be scheduled independently. This second method has the added advantage that it may be the key to reuse of hardware or software components in parallel applications.

In the next sections I define which type of modules are suitable for parallel programming and review the basic scheduling algorithm. In section 4, I explain how to improve the scheduling time of one process provided that the dependence graph is sparse. Section 5 explains how to do modular scheduling. In the conclusion, I present some open problems and discuss future work.

2 Communicating Regular Processes

Communicating Regular Processes are a variant of Kahn Process Networks [4] and of Communicating Sequential Processes [5]. The main difference is that regularity conditions are imposed on processes in order to allow temporal analysis of the full system.

2.1 Definitions

Processes. A process is a sequential program which can communicate with other processes through channels (see below). With the exception of channels, all variables are local to one process and are not visible from other processes. The code of a process can be written in any convenient algorithmic language. I use C here, but other choices are possible: Pascal, Fortran and others.

The code of a process is *regular*, or has static control [6]. Statements are assignments and bounded loop statements. All variables are considered part of some array, scalars being zero-dimensional arrays. Loops are of the arithmetics progression variety, and the loop upper and lower bounds are affine forms in numerical or symbolic constants and surrounding loop counters. The only method of address calculation is subscripting into arrays of arbitrary dimension. The subscripts must be affine forms in constants and surrounding loop counters.

The iteration vector of a statement is a list of its surrounding loop counters, from outside inward. The iteration vector of S must belong to the iteration domain of S, D_S, which is constructed from the bounds of the surrounding loops. Under the assumption that the program is regular, iterations domains are convex polyhedra (or, more precisely, sets of integral points inside polyhedra).

An iteration of S or *operation* is written $\langle S, x\rangle, x \in D_S$. The set of operations of a process P is the disjoint union:

$$E_P = \bigcup_{S \in P} \{\langle S, x\rangle \mid x \in D_S\},$$

and the set of operations of a process system is $E = \cup_P E_P$. In more abstract contexts, I may simply write $u \in E$ for a generic operation. $<_{\text{seq}}$ is the sequential order of execution.

Channels. A channel is an array of arbitrary dimension which is used as a communication medium from a process to another process. Channels are unidirectional. One process is declared as the writer to a channel. Considered as an array, each cell of the channel must be written no more than once by its writer. Writing to a channel is non-blocking.

On the other hand, a channel may have any number of readers, and there are no constraints on the pattern of reading. Reading is not destructive: a value remains in a channel at least as long as some process may have some use for it. If a process reads a cell which has not yet been defined, it blocks until a definition happens.

One can prove that these restrictions on processes and channels are enough to guarantee that the channel contents are the same for all executions of a CRP system, and are independent of relative processor speeds or scheduling decisions. The detailed proof will be given elsewhere.

2.2 An Example

The following trivial example specify a system in which a producer generates an infinite stream of values which are sent to a consumer which compute a sliding mean. I hope that the extensions to C are clear for the reader.

```
channel float A[];

process producer(){
  int i;
  for(i=0;;i++)
W:   A[i] = f(i);
}
```

```
process consumer(){
   float s;
   int i;
Z: s = 0.0;
   for(i=0;;i++)
R:    s = 0.5*(s + A[i]);
}
```

2.3 Dependences

Data dependences were defined, as early as 1966, for the purpose of parallelization [7]. For each operation u, let $R(u)$ be the set of memory cells that are read by u and $W(u)$ be the set of cells which are modified by u. Two operations u and v from the same process ($u <_{\text{seq}} v$) are in dependence if at least one of the three sets $W(u) \cap W(v)$ (output dependence), $W(u) \cap R(v)$ (flow dependence) and $R(u) \cap W(v)$ (anti-dependence) is not empty. The dependence relation is written $u \,\delta\, v$.

Assume now that the cell which causes the dependence is a channel cell. Before defining a dependence, one has to decide how to order the two dependent operations. This is not self-evident in the case of channels, since the dependent operations do not belong to the same process. In accordance with the intended semantics, I assume that the dependence is always from the write operation to a read, or that the write always occurs before any read to the same cell.

3 Scheduling

3.1 Target Architectures

In contrast to the above programming model, most of today electronic systems are synchronous: there is one global clock, and all changes of state occur in relation to the clock. Example of synchronous systems are VLIW processors and ASIC/FPGA special purpose circuits. A generic VLIW processor will be the main target architecture in what follows.

3.2 Schedules

A schedule is a function which assign a starting time to all operations in a program. In other words, a schedule is a function from E to the set of time values, T, and is a way of specifying an execution order. T may be any ordered set. The order associated to θ is:

$$u <_\theta v = \theta(u) < \theta(v).$$

The favorites for T are $\mathbb{N}$ and $\mathbb{N}^d$, lexicographically ordered. This second case gives rise to the so-called multidimensional schedules.

The execution order which is defined by a schedule must be legal, i.e. it must extend the dependence relation:

$$\forall u, v \in E : u \,\delta\, v \Rightarrow \theta(u) < \theta(v). \tag{1}$$

To solve this functional inequality, one has to postulate a shape for θ. The usual choice is that $\theta(\langle S, x\rangle)$ is an affine form in the iteration vector, x:

$$\theta(\langle S, x\rangle) = h_S.x + k_S, \tag{2}$$

where h_S is the timing vector of S and k_S is a scalar offset. For regular programs, this choice has the advantage that everything in (1) becomes affine, and that powerful results from the theory of linear inequalities, like Farkas lemma [8], can be used to characterize the solutions. The reader is refered to [6, 9] for details. A short review of the method will be given below.

3.3 Solving the Scheduling Constraints

The first step of the solution consists in splitting formula (1) according to the source and sink of the dependence. For a given pair of statements, S and T, the constraint now reads:

$$\forall x \in D_S, y \in D_T : \langle S, x\rangle \,\delta\, \langle T, y\rangle \Rightarrow \theta(\langle S, x\rangle) < \theta(\langle T, y\rangle). \tag{3}$$

Then, one has to eliminate the quantifiers on x and y. This can be done either by the vertex method [10], or by making use of Farkas lemma. Whatever the method, (3) can be shown to be equivalent to a system of affine constraints[1]:

[1] Here and in what follows, I assume that constant terms have been included in the matrices by the well known homogeneous coordinate trick.

$$M_{ST}(h_S, k_S)^T + N_{ST}(h_T, k_T)^T \geq 0. \quad (4)$$

M_{ST} and N_{ST} are constant matrices which can be computed from the program text. The direct application of Farkas lemma introduces new unknowns, the Farkas multipliers, which can be eliminated along the lines of [6].

Lastly, one can solve (4) using any convenient linear programming algorithm.

In some cases, the scheduling constraints are not feasible, because not all programs can be executed in linear time on many processors. One can resort in this case to multidimensional schedules, whose parallel latency is polynomial. The construction of multidimensional schedules is explained in [9]. I will ignore this point in this preliminary paper.

4 Scalability

The number of unknown in a scheduling problem is of the order of the number of statements times the mean depth of loop nests. The number of dependences is in general quadratic in the program size, and the number of constraints per dependences is again proportional to the mean nesting depth. Lastly, the Simplex algorithm, while exponential in the worst case, has a high probability of being cubic in the number of unknowns or constraints, when these two numbers are of the same order of magnitude. Hence, the direct solution of the scheduling constraints by linear programming does not scale well.

4.1 Stepwise Scheduling

To go further, one has to observe that the constraint matrix is sparse, or, rather, block sparse: see (4). If one compress each block M_{ST} or N_{ST} to a single cell, one gets the incidence matrix of the dependence graph.

If the scheduling problem is solved by a variant of the simplex algorithm, one cannot make use of this sparsity to speed up the resolution: the simplex has *fillup*. The solution is to use projection algorithms. The projection of a set D in $\mathbb{R}^{n+1}$ along its first dimension is:

$$\text{Proj}(D, y) = \{x \mid \exists y : y.x \in D\}. \quad (5)$$

It is well known that if D is a polyhedron, so is $\text{Proj}(D, y)$. For polyhedra, there are several projection algorithms:

- The simplest one is the Fourier-Motzkin algorithm. Its complexity is super exponential. Part of this complexity is due to the fact that the resulting system of constraints contains many redundant inequalities.
- One can also use parametric linear programming as in PIP [11]. The complexity is less, but the result still has many redundancies.
- Lastly, if one knows the Minkowski representation of D, it is easy to find the Minkowski representation of its projection. From that, one can reconstruct an irredundant constraint system with the Chernikova algorithm.

The last solution is probably the best one, especially since there exists an efficient implementation [12]. However, for the preliminary experiments that are reported here, I have used the Fourier-Motzkin algorithm coupled to a naive redundancy elimination method.

Whatever the projection algorithm, once the final feasibility test has succeeded, one can reconstruct values for the eliminated variables by back-propagation. This suggest the use of the following algorithm:

- For each statement S:
  - Collect all the rows of the constraint matrix where h_S has a non-zero coefficient.
  - Eliminate h_S.
  - Remember the bounds for h_S.
- If the resulting system is trivially unfeasible (like $-1 \geq 0$) stop. No schedule exists.
- For each statement S in reverse order:
  - The bounds for h_S are constants. Select a value within the bounds for h_S (e.g. the lower bound).
  - Substitute these values in all other bounds.

Experience with a limited set of programs shows that while this technique does not reduce much the number of constraints, the number of unknown at each elimination step decreases sharply, which is a big improvement since the Fourier-Motzkin algorithm is super exponential in the number of unknowns.

The order in which statements are eliminated is obviously important for the scalability of the algorithm. One may devise many heuristics for selecting the next victim: select for instance the statement with lowest degree in the dependence graph. Systematic evaluation of this and others heuristics is left for future work.

5 Modularity

In language and compiler design, the standard definition of a module is "a part of a program which can be *partially* compiled without reference to other parts". Traditionally, the result of partial compilation is called an *object*. When all modules have been compiled, another processor, the *linker*, is needed to finish the construction of the program. Modularity has many advantages. Modules promote reuse and increases the readability of programs. Also, in case of a modification, one recompile only the affected module(s). As we have seen earlier, the natural unit of compilation for a parallel program is the *process*.

5.1 Channel Schedules

Going back to the scheduling constraints (1), one can see that processes are not independent, as there will be relations between the schedules of the writer and the readers of each channel. This does not allow modular scheduling. The solution is to provide some "insulation" between processes.

Observe that each cell in a channel A is written only once at a definite time by statements from only one process. Therefore, one can postulate the existence of a channel schedule $\theta(\langle A, x\rangle)$ such that the value $A[x]$ is guaranteed to be available at time $\theta(\langle A, x\rangle)$ (and later). For simplicity, I will assume that θ is affine. There is a loss of generality here, but I believe it is not important for most programs and can be easily corrected in other cases.

With this definition, a channel dependence can be split in two parts:

- On the write side, a cell is not available before it has been written. Let $S : A[\omega_A(x)] := \cdots$ be a statement that writes into A:

$$\theta(\langle A, \omega(x)\rangle) \geq \theta(\langle S, x\rangle) + 1 \tag{6}$$

- On the other side, a cell cannot be read before it is available. Let $R : \cdots := \cdots A[\rho_A(x)] \cdots$ be a statement that read A:

$$\theta(R, x) \geq \theta(\langle A, \rho_A(x)\rangle). \tag{7}$$

The 1 in formula (6) is intended to represent a propagation delay through the channel. I have arbitrarily inserted this delay on the write side, but many other configurations can be used without changing the overall method.

5.2 The Modular Algorithm

Let h_P be the concatenation of the timing vectors for all statements in process P, and let h_A be the timing vector for array A. After application of the Farkas algorithm to (6) or (7) and elimination of the Farkas multipliers, the shape of the constraint matrix is as follows.

For each process P there is a system $U_P h_P \geq 0$ which represents the constraints generated by the inner dependences in P. The matrix U_P is block sparse, and each of its blocks is one of the M_S or N_S blocks in formula (4). For each process P and each channel A which is connected to P there is a system $V_{AP} h_P + W_{AP} h_A \geq 0$ which represents the constraints generated by the communication dependences of the system. These observations suggest the following modular scheduling algorithm.

1. Construct the constraint matrix for each process and its adjacent channels.
2. For each process P eliminate h_P from the constraints:

$$U_P h_P \geq 0,\ V_{PA} h_P + W_{PA} h_A \geq 0,\ \text{for all } A \text{ connected to } P \tag{8}$$

   This first pass of compilation is modular, in so far as this can be done one process at a time, without reference to other processes. The result is a system of constraints on channel schedules.
3. When all such *communication constraints* have been computed (or collected from a repository), they can be solved as a whole, giving a solution for the channel schedules. Again, the communication constraints matrix is block-isomorphic to the communication graph of the whole system, and has a high probability of being sparse. This is the only place where the system has to be considered *in toto*.

4. The solution for the channel schedules can then be substituted in the bounds for the coefficients of the schedules, and these coefficients can be recovered by back-substitution.
5. It remains to gather all schedules and submit them to a code generator. With present day tools [3], there is no hope of staying modular there, unless one deals with highly specialized architectures. However, tools like CLooG are quite efficient and can handle very large programs.

Consider the example of Sect. 2.2. The first step is to compile the two processes. Let:

$$\theta(\langle W, i\rangle) = \alpha i,\ \theta(\langle Z\rangle) = \beta,\ \theta(\langle R, i\rangle) = \gamma i + \delta,\ \theta(\langle A, x\rangle) = \epsilon x + \phi.$$

The producer has no data dependence, hence the only constraint is a communication constraint:

$$i \geq 0 \Rightarrow \epsilon i + \phi \geq \alpha i + 1.$$

Application of the Farkas algorithm gives $\phi \geq 1$ and $\epsilon \geq \alpha$ after elimination of the multipliers. After elimination of α, the only remaining constraint is $\phi \geq 1$.

In the consumer there is a flow dependence from Z to R, which gives $\delta \geq \beta+1$, and a flow dependence from R to itself, which gives $\gamma \geq 1$. Lastly, there is a communication dependence from A to R which entails $\phi \leq \delta$ and $\epsilon \leq \gamma$. The next step is the elimination of β, γ and δ from the system of constraints:

$$\delta - \beta - 1 \geq 0,\ \gamma \geq 1,\ \phi - \delta \geq 0,\ \gamma - \epsilon.$$

The resulting system is empty. The only communication constraint is $\phi \geq 1$ whose smallest solution is $\phi = 1$. From there, one may reconstruct the schedules:

$$\theta(\langle W, i\rangle) = 0,\ \theta(\langle Z\rangle) = 0,\ \theta(\langle R, i\rangle) = i + 1,\ \theta(\langle A, x\rangle) = 1.$$

This solution is not satisfactory, since one has to deposit an infinite number of values in A in one clock cycle. An easy way out is to slow down the producer by introducing a dependence:

```
C:  t = f(i);
W:  A[i] = t;
```

The schedules become:

$$\theta(\langle C, i\rangle) = 2i,\ \theta(\langle Z\rangle) = 0,\ \theta(\langle W, i\rangle) = 2i + 1,\ \theta(\langle A, x\rangle) = 2i + 2,\ \theta(\langle R, i\rangle) = 2i + 2.$$

Notice that it was not necessary to recompile the consumer. These schedules correspond to a VLIW program whose kernel is:

clock cycle	C	W	R
even	*		*
odd		*	

6 Related Work

While the literature on automatic parallelization is enormous, and the literature on scheduling is only slightly smaller, the problems of modular parallelization and of modular scheduling have not been extensively considered by the academic community, let alone industry.

In [13], the unit of modularity is the procedure, whose effect is summarized by computing *regions*. The drawback of this method is that one can find parallelism between procedure calls, and also inside procedures, but not parallelism that requires a transformation involving both a procedure and its calling context.

Nearer to the subject of this paper, Risset and Quinton [14] have defined structured scheduling for *systems* in the ALPHA specification language [15]. Systems can be scheduled independently. The schedules of several systems are then composed to give the global schedule. This is possible only if somewhat stringent restrictions are imposed on systems.

The use of processes in parallel programming dates back to the commencement of the subject. Kahn Process Networks [4] have been a source of inspiration for the present paper. The main difference is that in KPN, there are no constraints on the definition of each process – which may not be a program in the usual sense – hence their *a priori* analysis and compilation is almost impossible. This results in the present situation, where KPN are only used for simulation or even direct execution. In contrast, CRP systems can be checked statically or compiled into synchronous programs.

7 Conclusion and Future Work

This paper is very preliminary and many problems have to be solved if this proposal is to become a practical solution for the design of embedded systems. Let me quote some of them.

In the above description, there is nothing to bound the size of a channel. One needs a way of constructing schedules under the additional constraint that each channel uses no more than a given amount of memory. Let us note that the inverse problem (finding the amount of memory needed to support a given schedule) has been the subject of much research and that good solutions are known [16, 17].

For complexity reasons, as soon as resources are in a fixed finite amount, the restriction to affine schedules is no longer tenable. One has to use *many-dimensional schedules*. While there are methods for constructing such schedules [9], building their modular extension is by no means obvious.

Many problems in, e.g., image processing, are outside the regular (or polytope) model. One may sometime obviate this difficulty by overestimating dependences, or by encapsulating the irregular program parts, or by asking for help from the programmer. There is much work to be done in this direction.

References

1. Ancourt, C., Irigoin, F.: Scanning polyhedra with DO loops. In: Proc. third SIGPLAN Symp. on Principles and Practice of Parallel Programming, ACM Press (1991) 39–50
2. Quilleré, F., Rajopadhye, S., Wilde, D.: Generation of Efficient Nested Loops from Polyhedra. International Journal of Parallel Programming **28** (2000) 469–498
3. Bastoul, C.: Efficient code generation for automatic parallelization and optimization. In: ISPDC'03 IEEE International Symposium on Parallel and Distributed Computing. (2003) to appear, `http://www.prism.uvsq.fr/users/cedb/`.
4. Kahn, G.: The semantics of a simple language for parallel programming. In Holland, N., ed.: IFIP'94. (1974) 471–475
5. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21** (1978)
6. Feautrier, P.: Some efficient solutions to the affine scheduling problem, I, one dimensional time. Int. J. of Parallel Programming **21** (1992) 313–348
7. Bernstein, A.J.: Analysis of programs for parallel processing. IEEE Trans. on El. Computers **EC-15** (1966)
8. Schrijver, A.: Theory of linear and integer programming. Wiley, NewYork (1986)
9. Feautrier, P.: Some efficient solutions to the affine scheduling problem, II, multidimensional time. Int. J. of Parallel Programming **21** (1992) 389–420
10. Quinton, P.: The systematic design of systolic arrays. In Fogelman, F., Robert, Y., Tschuente, M., eds.: Automata networks in Computer Science, Manchester University Press (1987) 229–260
11. Feautrier, P.: Semantical analysis and mathematical programming; application to parallelization and vectorization. In Cosnard, M., Robert, Y., Quinton, P., Raynal, M., eds.: Workshop on Parallel and Distributed Algorithms, Bonas, North Holland (1989) 309–320
12. Wilde, D.: A library for doing polyhedral operations. Technical Report 785, Irisa, Rennes, France (1993)
13. Triolet, R., Irigoin, F., Feautrier, P.: Automatic parallelization of FORTRAN programs in the presence of procedure calls. In Robinet, B., Wilhelm, R., eds.: ESOP 1986, LNCS 213, Springer-Verlag (1986)
14. Quinton, P., Risset, T.: Structured scheduling of recurrence equations: Theory and practice. In: Proc. of the System Architecture MOdelling and Simulation Workshop. Lecture Notes in Computer Science, 2268, Samos, Greece, Springer Verlag (2001)
15. Leverge, H., Mauras, C., Quinton, P.: The ALPHA language and its use for the design of systolic arrays. Journal of VLSI Signal Processing **3** (1991) 173–182
16. Lefebvre, V., Feautrier, P.: Optimizing storage size for static control programs in automatic parallelizers. In Lengauer, C., Griebl, M., Gorlatch, S., eds.: Europar'97. Volume 1300 of LNCS., Springer (1997) 356–363
17. Darte, A., Schreiber, R., Villard, G.: Lattice-based memory allocation. In: 6th ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2003). (2003)