Implicitly parallel real-time signal processing

Phillip L. Nico
Department of Computer Science
University of California, Davis
Davis, CA 95616
nico@cs.ucdavis.edu

John Feo
Tera Computer Company
2815 Eastlake Avenue East
Seattle, WA 98102
john@tera.com

Robert Kim Yates
Lawrence Livermore National Laboratory
L-560, P.O. Box 808, Livermore, CA 94550
rkyates@llnl.gov

ABSTRACT

This paper describes a compiler system for automatic parallelization, partitioning, and scheduling of real-time signal processing applications written in a functional language running on general-purpose hardware. The high performance levels achieved by commodity processors can meet the demands of many real-time signal processing applications; however, they are not used effectively in practice because tools are not available to automatically create parallel partitions and schedules for DSP application codes that have critical time needs. We have extended SISAL, a high-performance functional language for scientific computations, with constructs to express latency and throughput constraints. Given a description of the target architecture and an application with specified timing requirements, the SERT compiler generates a precise, minimal data dependency graph and then partitions and creates a parallel schedule for the graph to satisfy the specified constraints if possible. When complete, SERT's ability to rapidly and automatically generate parallel schedules for different system configurations will make it an effective aide to real-time systems developers.

1 INTRODUCTION

Traditionally, most real-time signal processing (RTSP) applications have required special-purpose hardware and software, since only extremely carefully designed systems could deliver the performance demanded of them. In spite of the great cost of developing such systems, they soon become obsolete and have to be replaced. They cannot easily be reprogrammed to meet changes in system specifications or to take advantage of new algorithms. Recent advances in processor technology have produced a large number of commercially available general purpose "commodity" processors sufficiently powerful to handle most real-time applications. Real-time system development time and expense can be greatly reduced by using these processors in conjunction with high-level languages for system implementation. At the same time, system lifetimes can be increased since general-purpose processors compete in a larger arena where advances within product lines come frequently and remain compatible with older versions.

In order to use general-purpose hardware and programming methods, developers of RTSP systems must be able to prove that their software meets the performance constraints imposed by their applications. Doing this analysis by hand is, at best, difficult and time-consuming, and, at worst, erroneous. Hence there is a need for high-level languages and compilers that automatically partition a program into parallel components and schedule them to exploit available hardware resources in such a way that performance requirements can be guaranteed to be met.

We are developing and evaluating a language and compiler for specifying time-critical applications and automatically creating a parallel schedule for them. The language for these experiments is SERT which stands for SISAL Extended for Real-Time. Advantages of our approach over current explicitly parallel or manual systems include both potentially greater accuracy with respect to timing analysis and increased ease of development. In addition, our approach allows for rapid prototyping of new systems and simplifies porting to new computing platforms as they are developed. The ability to test different system configurations quickly will allow system implementors to determine minimal (*i.e.*, least expensive) hardware configurations easily, yielding potentially substantial economic benefits.

The imperative semantics of conventional programming languages, such as Fortran and C, make it difficult to identify safe orders of execution, reducing the number of parallel partitions and schedules available. On the other hand, the functional semantics of SISAL permit the compiler to consider all possible choices. Since functional programs are free of side-effects and referentially transparent their execution order is constrained only by data dependencies; the compiler can generate exact data dependency graphs. Any transformation of a graph that does not violate the data dependencies of the original graph is legal. Parallelism is implicit since any two independent operations, whether instructions or large tasks, may execute in parallel. Additionally, the precision of the graphs (i.e., the lack of spurious edges) makes it possible to calculate more accurate timings for each program module.

Our chosen application domain also offers significant advantages. Signal processing tasks generally operate as filters, consuming a stream of input datasets from a bank of sensors or other "real-world" input devices whose operations are well-defined and regular [Nit89]. Computationally, the filters are regular and their behavior does not vary widely in response to input *values*. This determinism facilitates timing estimation and makes program transformation easier [Den95].

In this paper, we describe work in progress to define

SERT and develop the necessary compiler technology to meet the needs of the RTSP community. We have extended the high-performance functional language SISAL to include timing specifications, and are developing parallel partitioning and scheduling heuristics capable of organizing the code to meet the specification, if possible.

We make no claims about the universal suitability of SERT's functional programming model to arbitrary real-time applications. Moreover, the simple functional model used here excludes general higher-order functions and lazy evaluation. We have restricted our attention primarily to numerically intensive deterministic DSP algorithms, which represent an important class of problems. The question of how far purely functional languages should be extended (e.g., with nondeterministic merges, mutable variables, and interactive I/O) to accommodate real-time and other reactive applications is fascinating, controversial, and unresolved. It is worth mentioning that SERT, like SISAL, can be used in a mixed-language programming environment, with the core computations written in SERT called from Fortran or C. Our approach can work with any programming language for which the compiler can generate a data dependency graph; however, its value is affected by the number of spurious edges and the accuracy of estimated timings. Of course, we believe a language like SERT to be especially well matched to these requirements.

The rest of this paper is organized as follows: Section 2 describes the SISAL language and its suitability as a language for signal processing. Section 3 describes the extensions made to SISAL for managing timing constraints, while Section 4 looks briefly at how they might be applied to scheduling applications. Section 5 discusses some related work, and, finally, Section 6 offers some conclusions.

2 THE SISAL LANGUAGE

SISAL [M+85], "Streams and Iteration in a Single Assignment Language," is a functional language designed for scientific applications on parallel computers. The language supports array operations, parallel and iterative loops, user-defined reductions, and mixed-language programming By modifying SISAL to deal with real-time constraints on latency and throughput we hope to extend the benefits of implicitly parallel functional programming to a wider application domain.

SISAL is a pure functional language, with no side effects and no nondeterministic behavior. Compared to conventional languages the lack of side effects enhances modularity and makes data dependencies much clearer to both the programmer and the compiler. The software engineering advantages are manifold: programs are much easier to write, modify, test and validate [DFRM94, Hen91]. Determinism guarantees that programs will run correctly regardless of the architecture features (operating system, memory hierarchy, communication network, number of processors) of the target machine. The benefits of determinism, particularly on parallel machines, should be obvious to anyone who has debugged a program with race conditions.

The greatest difference between SISAL and other functional languages such as Haskell [H+92] is that SISAL is designed to allow efficient implementations on parallel supercomputers. The language does not require lazy evaluation, and there are many built-in operators to facilitate parallel construction and manipulation of arrays of data. The result is that SISAL is as fast or faster than Fortran in many scientific applications [Can92]. Studies have shown that SISAL is a suitable signal processing language. It has demonstrated high performance in numerically intensive applications. The language includes a *stream* data type that supports potentially infinite sequences of values and potential for producer-consumer parallelism. Researchers have implemented important signal processing computations (FFT, convolution, correlation, and smoothing) in SISAL and have integrated the functions into single systems.

The parallelism in SISAL programs is implicit: it is restricted only by data dependence. SISAL programs compile to dataflow graphs in which nodes represent operations and edges represent data dependencies. Unlike dataflow graphs generated by compilers for conventional languages which may include spurious edges to ensure correctness, SISAL dataflow graphs have a minimum number of edges. The clean graphs make it possible for the SISAL compiler to apply many more transformations to the intermediate code, and to apply each more effectively, than can conventional language compilers. In particular, this allows the SISAL compiler to estimate accurately the execution time of code segments and evaluate different scheduling and partitioning alternatives. Several research projects studying the program mapping problem have used SISAL and its intermediate form IF1 [Sar87, Wol94, PAM94b].

In the next section we show how we have extended SISAL to express real-time performance constraints and how these constraints are propagated and managed within the compiler.

3 EXTENDING SISAL FOR REAL-TIME: SERT

The SISAL language, as described above, already provides everything semantically necessary to support stream signal-processing applications, but it has no means of expressing timing constraints. To provide this necessary functionality, we are developing the SERT system which extends the language by providing a set of "timing pragmas." With these we can annotate an application's source code with its particular timing requirements. The compiler then propagates these source-level constraints down through the internal dataflow graphs and uses them to guide program transformation and scheduling.

In the most general sense, the types of timing constraints on RTSP applications fall into two orthogonal categories: *latency* and *throughput*. A task which has a deadline has a latency constraint. A throughput constraint arises when a sensor produces samples a particular rate and the program must be able to keep up with it. A special case of a throughput constraint is a *period* constraint. Here samples are required to be produced or consumed with exactly the specified interval, *e.g.* every two seconds, while throughput only specifies the rate, perhaps 2 every 4 seconds.

3.1 TIMING CONSTRAINT ANNOTATIONS

The timing annotations added to create SERT were chosen according to our belief that it should be possible to express timing requirements in a natural way which mimics how a system designer would think while easing the scheduler's burden by not over constraining it. To do this, we have developed a system of pragmas, outlined below, which place requirements only on externally visible parts of a program subtask, it's inputs and outputs. A pragma's scope is the nearest enclosing language construction with control semantics; *i.e.,* a function, loop, or conditional expression. We find that this allows timing requirements to be expressed in a natural way without unnecessarily cluttering up the language.

Constraint Pragmas	
Aggregate	%$ latency = <*time-expr*> %$ throughput = <*expr*> per <*time-expr*>
Element	%$ latency <*stream-ident*> = <*time-expr*> %$ throughput <*stream-ident*> = <*expr*> per <*time-expr*> %$ period <*stream-ident*> = <*time-expr*> %$ max skew <*stream-ident-1*> from <*stream-ident-2*> = <*time-expr*> %$ offset <*stream-ident-1*> from <*stream-ident-2*> = <*time-expr*>
Advice Pragmas	
	%$ max iterations = <*expr*> %$ max recursion = <*expr*> %$ decomposition <*array-ident*> = <*type*>

Table 1: Compiler directives for expressing timing constraints

Table 1 lists the ten possible source-level timing annotations for an application. They are first subdivided into *constraint* and *advice* pragmas. A *constraint* pragma informs the compiler of some requirement for program behavior, for instance a latency bound on a function invocation or a minimum throughput for a data stream. An *advice* pragma gives the compiler some information that it might not otherwise be able to determine, for example an algorithmic bound on a loop's iteration count that might not be evident to the timing analyzer. The constraint pragmas are further subdivided into *aggregate* and *element* constraints. The former place requirements for the completion of the entire scope to which they refer while the latter constrain the production or consumption of individual data elements.

The ten statements listed in Table 1 allow the programmer to specify arbitrarily complex sets of timing constraints as they are necessary. Additionally, they support `offset` and `max skew` directives which permit streams of data to be offset with respect to one another. An `offset` rigidly displaces two streams, while a `max skew` places a bound on the amount of slippage. Relaxing synchronization requirements between data streams increases scheduling options.

The advice pragmas allow the programmer to impose bounds on iteration counts or recursion depths where the scheduler might not be able to deduce them. In addition, the `decomposition` directive suggests a mode of data-decomposition. These directives are considered advisory by the compiler and will be ignored if the compiler can determine better bounds. They must be used with caution, however, since the compiler accepts them as correct and an incorrect (low) iteration bound can cause a task to miss deadlines.

All of the above will be made clearer through the example in Figure 1. Here we have an animation routine from some sort of visualization system which consumes a stream of states and returns a video and audio stream. The constraints refer to the video stream as "returns 1" and to the audio stream as "returns 2" corresponding to their positions in the function header. The rate at which states are consumed is unimportant, but the pictures and sound streams must be produced at intervals of $\frac{1}{30}s$. The `max skew` annotation indicates that the programmer is willing to tolerate up to $\frac{1}{2}s$ skew between the two streams, if necessary, to accommodate a better schedule. Finally, the `latency` annotation requires that the time between the arrival of a frame and the production of its corresponding audio and video samples be less than 5 seconds effectively requiring a reasonable response time from the "go" button to the beginning of the display.

As a program is compiled, the optimizing SISAL compiler(OSC) generates internal IF1 dataflow graphs which it marks with timing requirements corresponding to those given in the source. At this stage, there is one additional type of pragma, the *group* pragma, which allows a constraint to be applied as an aggregate to a set of dataflow nodes and edges. This is useful for pragma propagation as it exposes the relationships between different parts of the program.

3.2 PROPAGATING TIMING CONSTRAINTS

We expect that most of the timing behavior of interest to RTSP system designers will be relatively high-level. That is, the system architect is more likely to be interested in keeping up with a sensor input stream and system response time than in the lowest level interleaving of fragments of computations. Actually scheduling an application, however, requires knowledge of the timing requirements of each individual piece of a program at least in relation to others. To this end, the compiler iteratively propagates the given constraints until each node and edge of the dataflow graph has a complete set of all of its requirements, usually as members of groups. These sets of constraints effectively yield a system of timing equations which can be used to guide parallelization and scheduling decisions.

Node Id	Assigned Constraints	Computed Constraints
g1	latency = $0.1s$	latency = $0.1s$
n2	—	group 0
n3	—	group 0
n4	—	group 0
Edge Id	**Assigned Constraints**	**Computed Constraints**
e1	—	group 0
e2	—	group 0
e3	—	group 0
e4	—	group 0
e5	—	group 0
Group Constraints		
Group	**Size**	**Constraint**
0	8	latency= 0.1s

Table 2: Constraints for the dataflow graph in Figure 2

To illustrate constraint propagation, Figure 2 shows a simple latency-constrained SISAL function. Its initial and final propagated constraints are given in Table 2 as *assigned* and *computed* constraints respectively.

As we see, initially, the only constraint is that the entire program graph, labeled *g1*, must complete in $0.1s$. Once the pragmas have been propagated, we see that all the nodes and edges of the function `main()` have been added to a group (group 0) with a group latency of $0.1s$. What is not shown

```
function animate(S_in:stream[state] returns stream[picture], stream[sound])
  %$ period returns 1 = 1/30 sec
  %$ period returns 2 = 1/30 sec
  %$ max skew returns 1 from returns 2 = 500 msec
  %$ latency S_in = 5 sec
  for s in S_in
  returns stream of make_frame(s)
  returns stream of make_sound(s)
  end for
end function
```

Figure 1: Example: Animation function with timing constraints.

```
function main(n,m:integer returns integer)
 %$ latency = 0.1s
 crunch(n) * crunch(m)
end function
```

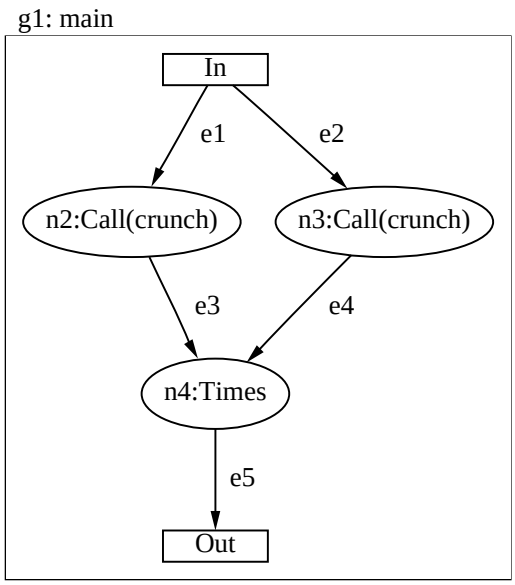

Figure 2: Example: A simple latency-constrained SISAL function and the generated dataflow graph.

here is that the constraint will also propagate down hierarchically through the function graph of `crunch()` until all of its nodes are also part of group 0.

The propagation mechanism of constraints from call nodes onto function graphs is noteworthy. In larger programs a function may inherit constraints from many different callers in this fashion. If they are not the same, either they will be merged into one that satisfies both, or the function will be replicated and one version will be scheduled for each timing environment. The second choice clearly provides the best chance of meeting overall timing bounds, but can cause substantial code growth with all its corresponding problems.

While this group constraint above expresses the same time bound as the original graph constraint, it contains more practical information—a list of all the other elements with which it is competing for time. On a uniprocessor any node can determine the time available to it by looking at the group's bound. For example, for one of the calls to `crunch()`:

$$\begin{aligned} T_a(n2) &= 0.1s - T_n(n3) - T_n(n4) - T_n(e1) - T_n(e2) \\ &\quad -T_n(e3) - T_n(e4) - T_n(e5) \end{aligned}$$

where T_a and T_n are "time allowed" and "time needed" respectively. On a multiprocessor instead of simply time, both time and resources (space) are allocated so the computation is a little more complicated, but analogous. The scheduling example in the following section illustrates this task.

After constraint propagation each node and edge in the entire dataflow graph knows, in a sense, where it stands with respect to the rest of the program. Once this is known, the scheduler can step in and do its work.

4 SCHEDULING THE CONSTRAINED GRAPHS

Once the compiler has taken the program and constructed the dataflow graph with timing constraints, it must begin searching for a schedule to meet those constraints. It should be borne in mind that compiling and scheduling for real-time systems is inherently different from compiling for traditional systems. Whereas for traditional systems compilers attempt to minimize the *average* latency and maximize the *average* throughput, for real-time systems compilers must limit the *maximum* latency while guaranteeing a *minimum* throughput[1]. In addition, for most real-time systems there is no real advantage in finishing early since there is nothing else to do until a sensor returns another sample, so determinism can often be traded for somewhat higher maximum latencies.

The general problem of generating an appropriate schedule from the constrained dataflow graph is NP-hard, so the SERT scheduler will be iterative and driven by one or more heuristics. Regardless of the driving heuristic, the compiler must divide up the time and available processors in the machine in such a way that all tasks can meet their deadlines and keep up throughput. Since it is not always possible to

[1]This is not strictly true, of course. System utilization is a concern for designers, but it is subordinate to meeting deadlines.

```
function ab_cross(S: stream[image] returns stream[image])
  %$ latency returns 1 = 10s
  %$ period  S = 2s
  for I in S
    out := unborder(thin(threshold(average(border(I)))))
  returns
    stream of out
  end for
end function
```

Figure 3: `ab_cross()` with timing constraints.

know *a priori* which way to make the first move, some backtracking may be necessary.

4.1 EXAMPLE: THE ABINGDON CROSS

We are currently just beginning the phase of this work where we are developing and implementing scheduling heuristics. Here, however, we present a very simple heuristic as a proof of concept.

The Abingdon Cross [Pre86], shown in Figure 3, is an image processing benchmark which consists of a series of filters to extract the skeletal axis of a cross from a larger image embedded in noise. The grayscale input image is first surrounded with a frame of black pixels in `border()`. It is then smoothed in `average()`, mapped to black-and-white in `threshold()`, and the axial cross is extracted by iteratively eroding the frontier of the image in `thin()`. Finally, the border is removed in `unborder()`. For the purposes of this example we have required it to accept one image every $2s$ and to take no longer than $10s$ per image.

The target machine machine model will be a fictional 64-processor shared-memory architecture. Instruction timings are generated from a table of costs borrowed largely from [Wol94]. Table 3 gives the expected execution times of the constituent routines of `ab_cross()` for various numbers of processors derived by manually traversing the IF1 graphs of the program and computing the worst case execution times with respect to the given operation costs.

```
1. Schedule everything sequentially.
2. Estimate execution times
3. while ( deadlines will be missed )
4.   Parallelize longest parallelizable element
     in the offending region
5.   Repropagate and re-estimate timing
6. end while
```

Figure 4: A simple scheduling heuristic

Figure 4 defines a simple scheduling heuristic where the task is initially scheduled sequentially and iteratively parallelized until it meets its deadlines. At each iteration the timing requirements for the current configuration are propagated through the program graph and execution times are estimated. If it appears that a node or group of nodes will miss its deadline the scheduler will choose the subregion with the most potential gain from parallelization and increase it to the next level of parallelization. In this example we have chosen to allocate processors as powers of two, but this is not necessary. If step 4 is reached and there are no more parallelizable regions the compiler must declare failure and either move on to another heuristic or give up. This heuristic guarantees a minimum amount of parallelism which is not a bad thing since there is no benefit to finishing early and sequential code is more efficient.

Before the scheduler tries to choose particular parallelizations to incorporate, it must decide what sort of schedule it is seeking. That is, it can decide to build task instances which meet the latency bound and replicate them until the throughput requirement is satisfied, or it can try to partition each instance across enough processors to satisfy both directly. For this example, it is possible to go in either direction with each leading to a different plausible schedule.

The first decision which must be made is how to subdivide the processing resources of the machine. That is, decide how many processors should be allocated to each instance of the task. This determines the real latency bound on the execution. While the programmer may have specified some minimum latency, the throughput determines an effective latency for the computation. In the case here of the Abingdon Cross, the throughput of 1 per $2s$ implied in the `period` pragma requires an effective latency of $2s$ per instance. That is, if the whole machine consumes a single instance at a time then the latency is $2s$. If the machine consumes two instances at a time (each instance being allocated half the resources), then the effective latency per instance is $4s$. In general, the real latency requirement is given by:

$$L_R = \min\{L_P, L_E * P\}$$

where L_R is the real latency, L_P is the latency bound specified by the programmer, L_E is the effective latency required by the throughput bound, and P is the degree of partitioning of the machine. A corollary of this is that there is no sense in partitioning the machine further than the point where $L_E * P$ becomes greater than L_P. Thus the maximum useful partitioning with this approach, $P_{\max}$, is given by:

$$P_{\max} = \left\lfloor \frac{L_P}{L_E} \right\rfloor$$

For this example, we see the $P_{\max}$ is 5. Since $P = 3$ and $P = 5$ do not divide the 64 processors evenly they will leave leftover processors idle, so we will concern ourselves with $P = 1, 2,$ and 4.

Using the execution times from Table 3 and following the heuristic in Figure 4 we see that we can find a valid schedule for any of the values of P. The final number of processors allocated to each function are given in Table 4 while the resultant schedules are illustrated in Figures 5–7.

	Schedule		
Function	$P = 1$	$P = 2$	$P = 4$
border	1	1	1
average	64	32	16
threshold	4	2	1
thin	64	32	16
unborder	4	2	1
Time:	1.99s	3.98s	7.86

Table 4: Processors allocated to each routine by least-parallelism scheduling.

The expected execution costs shown in Table 3 show that for this machine model speedups are almost linear and the bulk of the available parallelism is to be found in `thin()`. Thus, in each case the first thing the scheduler will do is to expand `thin()` until it has either filled the allocated space or the deadlines are met. Even with all the processors have been allocated, each instance will still miss, so the heuristic turns its attention to the routines with the most potential left, `average()`, and then `unborder()`. At this point, the deadlines will be met so the scheduler stops and lays out

Function	Number of Processors						
	1	2	4	8	16	32	64
border	0.00019s	0.00009s	0.00005s	0.00003s	0.00003s	0.00004s	0.00007s
average	0.70603s	0.35302s	0.17582s	0.08792s	0.04397s	0.02201s	0.01105s
threshold	0.07900s	0.03950s	0.01975s	0.00988s	0.00495s	0.00250s	0.00130s
thin	124.50862s	62.25439s	31.12738s	15.56407s	7.78279s	3.89292s	1.94952s
unborder	0.05283s	0.02642s	0.01321s	0.00661s	0.00332s	0.00168s	0.00089s

Table 3: Expected function execution costs for `ab_cross()`(in seconds)

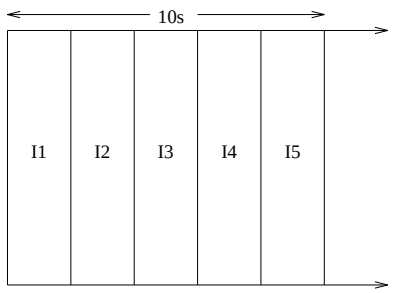

Figure 5: Schedule $P = 1$: Machine is not partitioned.

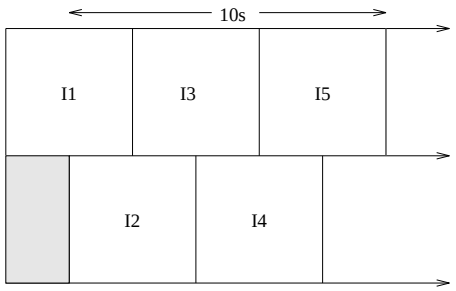

Figure 6: Schedule $P = 2$: Partitioned 2-ways and replicated twice

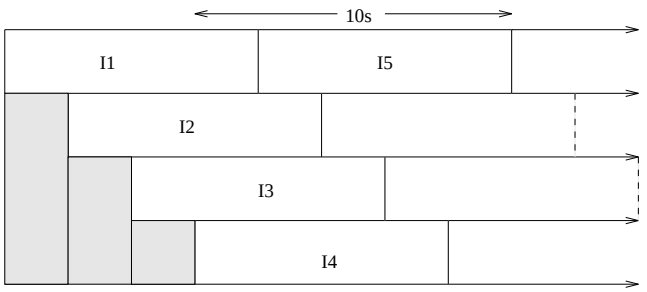

Figure 7: Schedule $P = 4$: Partitioned 4-ways and replicated four times

the tasks offsetting each instance by $2s$ to accommodate the periodicity requirement.

It is the very linearity of the speedups which allows all three of these schedules to work. If architectural or algorithmic changes made speedups even slightly less favorable this might not be the case. The $P = 1$ schedule which allocates 64 processors to each instance would fail even if the speedup for 64 processors was 62.5. However, the more sequential, and thus more efficient, $P = 2$ and $P = 2$ schedules might still succeed.

By inspection of these schedules we can see how even a very naïve scheduling heuristic was able to find three different schedules for this simple example. The development of better heuristics should allow us to handle more difficult tasks as well.

5 RELATED WORK

The idea of using functional programs operating on streams of data to solve signal processing problems dates back to the early days of dataflow [Wen75, DW79]. One of the most elegant results in the theory of concurrency [Kah74] showed that the behavior of networks of time-insensitive processes communicating via streams can be simply described by domain-theoretic techniques. Kahn's model can be extended to time-sensitive (even non-monotonic) processes such as a determinate, timed variant of the nondeterministic fair merge if time is made explicit in the semantics [YG93, Yat93]. Scheduling and buffer allocation in dataflow stream programs have been addressed in [Lee91a, Lee91b, GGP92].

Real-time programming with functional languages has also been done by groups using Signal, Lustre, Erlang and Haskell; however, these groups have not, to our knowledge, provided automatic parallelization to satisfy timing constraints. Signal [LGLL91] is a real-time equational language based on a particular *synchronous* theory of time. The emphasis of work on Signal to date has been on deducing properties of the program rather than on deriving multiprocessor implementations. Lustre [HCRP91] is a real-time functional language based on the same synchronous theory as Signal; as with Signal, work has focussed on program properties rather than on parallel implementations. Erlang [AVW93] is a "partially" functional language (side effects are allowed) for soft real-time systems that has been used in industrial telecommunications switches. In Erlang parallelism must be programmed explicitly. Haskell [H$^+$92] is a lazy functional language that has been used for prototyping some signal processing applications [Gob94].

Much more work has been done on the problem of scheduling for real-time systems than we could possibly report here. [SSDB95] and [ZLJW95] do a nice job of summarizing traditional scheduling results as applied to real-time systems. In some simple cases it is possible to find polynomial time scheduling algorithms, but in general these scheduling problems tend to be NP-hard [GJ79]. Some in-

teresting work on the application of heuristics to real-time scheduling problems can be found in [RSL94, WRS92].

There has been considerable interest lately in the use of compiler directives to express timing constraints and in compiler techniques to transform programs to meet them [GH95, CD95, SMY95]. But that work concentrates on scheduling for a single processor, whereas we are primarily interested in creating a parallel partition of a program with concurrent scheduling on parallel processors.

The main implementations of SISAL have been on shared memory multicomputers. However, some research has been done on implementing SISAL on distributed memory and non-uniform memory access architectures [HB91, PAM94a, WF93, Bea95].

6 CONCLUSIONS

In this paper we have described a functional language and compiler system for automatic parallelization, partitioning, and scheduling of real-time signal processing applications on general-purpose hardware. By adapting the existing SISAL language and compiler we take advantage of established technology that has already been shown effective in creating automatically parallelized code from very high-level programs. Side-effect-free semantics and implicitly parallel array operators already allow the SISAL compiler to do an excellent job of transforming program graphs to run effectively on a wide variety of multicomputer architectures. With the addition of the timing semantics described here, and with the development of further scheduling heuristics and transformations, SERT will be able to provide a functionality and ease of use difficult to match with other development systems.

ACKNOWLEDGMENTS

Thanks to Richard Games and John Ramsdell of the MITRE Corporation and Jack Dennis of MIT for many interesting discussions about implicit parallelism for signal processing.

REFERENCES

[AVW93] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 1993.

[Bea95] Patrick C. Beard. An implementation of SISAL for distributed-memory architectures. Master's thesis, U. of California, Davis, June 1995. Also available as Lawrence Livermore National Laboratory Report UCRL-LR-122353.

[Can92] David Cann. Retire FORTRAN? a debate rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[CD95] Tai M. Chung and Hank G. Dietz. Language constructs and transformation for hard real-time systems. In *Proc. of the 1995 ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 45–53, La Jolla, June 1995.

[Den95] Jack B. Dennis. Static mapping of functional programs: An example in signal processing. In *Proc. of the High-Performance Functional Computing Conference '95*, Denver, Colorado, April 1995.

[DFRM94] T. DeBoni, J. Feo, G. Rodrigue, and J. Muller. Implementation and performance of a domain decomposition algorithm in Sisal. In *Proc. of the Twenty-Seventh Ann. Hawaii Int. Conf. on System Sciences*, pages 605–614, 1994.

[DW79] J. B. Dennis and K. S. Weng. An abstract implementation for concurrent computations with streams. In *Proc. of the 1979 Int. Conf. on Parallel Processing*, pages 35–45, Bellaire, Michigan, August 21–24, 1979.

[GGP92] G. Gao, R. Govindarajan, and Prakash Panangaden. Well-behaved dataflow for DSP computation. In *ICASSP-92, 1992 International Conference on Acoustics, Speech, and Signal Processing*, March 1992.

[GH95] R. Gerber and S. Hong. Compiling real-time programs with timing constraints refinement and structural code motion. *IEEE Trans. on Software Engineering*, 21(5):389–404, May 1995.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W.H. Freeman, 1979.

[Gob94] David M. Goblirsch. A software system for training phonetic hidden Markov models. Technical report, The MITRE Corporation, May 1994. Notes and programs available by ftp from haskell.systemsz.cs.yale.edu in file pub/haskell/incoming/hmms-2.0.tar.gz.

[H+92] Paul Hudak et al. Report on the functional programming language Haskell. *SIGPLAN Notices*, 27(5), May 1992.

[HB91] M. Haines and W. Bohm. Task management, virtual shared memory, and mutithreading in a distributed memory implementation of Sisal. In *Proc. of PARLE '91 – Parallel Architectures and Languages Europe*, Eindhoven, The Netherlands, June 1991.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D Pilaud. The synchronous data flow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.

[Hen91] Christopher P. Hendrickson. Programming a real code in a functional language. Technical Report UCRL-JC-108326, Lawrence Livermore National Laboratory, September 1991.

[Kah74] G. Kahn. The semantics of a simple language for parallel processing. In *Information Processing 74*, pages 471–475, 1974.

[Lee91a] E.A. Lee. Consistency in dataflow graphs. *IEEE Trans. on Parallel and Distributed Systems*, 2(2):223–235, April 1991.

[Lee91b] E.A. Lee. Static scheduling of data-flow programs for DSP. In J.-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*. Prentice-Hall, 1991.

[LGLL91] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real-time applications with SIGNAL. *Proc. of the IEEE*, 79(9):1321–1336, September 1991.

[M+85] J. R. McGraw et al. SISAL: Streams and iteration in a single assignment language—language reference manual version 1.2. Technical Report M-146, Lawrence Livermore National Laboratory, 1985.

[Nit89] Peter Nitezki. Exploiting data parallelism in signal processing in a data flow machine. *Computer Architecture News*, 17(3):54–61, 1989. Also,Proc. of the 16th International Symposium on Computer Architecture.

[PAM94a] S.S. Pande, D.P. Agrawal, and J. Mauney. Compiling functional parallelism on distributed-memory systems. *IEEE Parallel and Distributed Technology: Systems and Applications*, 2(1), Spring 1994.

[PAM94b] S.S. Pande, D.P. Agrawal, and J. Mauney. A threshold scheduling strategy for Sisal on distributed memory machines. *J. of Parallel and Distributed Computing*, 21(2):223–236, May 1994.

[Pre86] K. Preston, jr. Benchmark results — the abingdon cross. In L. Uhr et al, editor, *Evaluation of multicomputers for Image Processing*. Academic Press, Cambridge, Mass., 1986.

[RSL94] Stefan Rönngren, Behrooz Shirazi, and Dan Lorts. Empirical evaluation of weighted and prioritized static scheduling heuristics for real-time multiprocessing. In *Proceedings of the Second Workshop on Parallel and Distributed Real-Time Systems*, pages 58–63, 1994.

[Sar87] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* PhD thesis, Stanford University, April 1987. Tech. Rpt. CSL-TR-87-328.

[SMY95] A. D. Stoyenko, T. J. Marlowe, and M. F. Younis. A language for complex real-time systems. *Computer Journal*, 38(4):319–338, November 1995.

[SSDB95] J. A. Stankovic, M. Spuri, M. Di Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, June 1995.

[Wen75] K.S. Weng. Stream-oriented computation in recursive data flow schemes. Technical Report MAC/TM–68, MIT Laboratory for Computer Science, October 1975.

[WF93] R. Wolski and J. Feo. Program partitioning for NUMA multiprocessor computer systems. *J. of Parallel and Distributed Computing*, 19(3), November 1993.

[Wol94] Richard M. Wolski. *Program Partitioning and Scheduling for NUMA Computer Architectures.* PhD thesis, U. of California, Davis, 1994. Also available as Lawrence Livermore National Laboratory Report URL-LR-117760.

[WRS92] Fuxing Wang, Krithi Ramamritham, and John Stankovic. Bounds on the performance of heuristic algorithms for multiprocessor scheduling of hard real-time tasks. In *IEEE Real-Time Systems Symposium*, pages 136–145, 1992.

[Yat93] R.K. Yates. Networks of real-time processes. In *Proceedings of Concur '93*, 1993.

[YG93] R.K. Yates and G.R Gao. A Kahn principle for networks of nonmonotonic real-time processes. In *Proceedings of PARLE '93, Parallel Architectures and Languages Europe.* Springer-Verlag, 1993.

[ZLJW95] Jiang Zhu, Ted Lewis, Weldon Jackson, and Russel Wilson. Scheduling in hard real-time applications. *IEEE Software*, 12(3):54–64, May 1995.

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-Eng-48. This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.