Modeling Multi-threaded Architectures in PAMELA for Real-time High Performance Applications*

S. Balakrishnan and S. K. Nandy
Supercomputer Education & Research Centre
Indian Institute of Science
Bangalore, India
{sbalki, nandy}@serc.iisc.ernet.in

Arjan J. C. van Gemund
Faculty of Electrical Engineering
Delft University of Technology
Delft, The Netherlands
a.vgemund@et.tudelft.nl

Abstract

In this paper we present a method to explore the design space of multi-threaded architectures using the PAMELA *[3] modeling language. The domain of applications we consider is digital signal processing (DSP), where high performance is derived by exploiting both fine grain and coarse grain parallelism in the application. The modeling scheme takes an unified view of both fine grain and coarse grain parallelism in a given application to performance meter the architecture. The application – written using a high level language, is compiled, and a trace generated for benchmark data in terms of the instruction set architecture of the processor. The generated trace is for a single uni-threaded, uni-processor system. This trace is preprocessed and retargetted to generate multi-threaded architecture specific* PAMELA *code. Using a material-oriented approach, the resulting* PAMELA *code is executed to evaluate various architecture options over the entire design space iteratively, subject to implementation constraints. We demonstrate the suitability and simplicity of the approach with an example.*

1 Introduction

Simultaneous multi-threading is fast evolving as an alternative architecture for high performance applications. In such architectures multiple threads share all available processor resources to exploit both fine-grain (instruction level parallelism) and coarse grain parallelism (task level parallelism). While simultaneous multi-threading for general purpose applications has been studied [11] and quite a few conclusions drawn in it's favor, there seems to be less of an effort to study these architectures with special attention to real-time Digital Signal Processing (DSP) applications. This is precisely the motivation for our study in which we have arrived at a modeling scheme that can capture both coarse grain and fine grain parallelism in the application and provide a mechanism to performance meter the architecture in terms of its basic computation, communication and storage resources.

In this paper we propose a method to model multi-threaded architectures in the PAMELA language [3] that provides an efficient simulation tool for the performance modeling of parallel architectures. The method is new in that we take an unified view of both fine grain and coarse grain parallelism in the application, and also in the way we explore the design space of the architecture to maximize performance without imposing any artificial restrictions. In the following section a brief overview of PAMELA is given to serve as preliminaries for modeling multi-threaded architectures described in section 3. As a case study, we demonstrate in section 4 how PAMELA can be used to model SYMPHONY, a multi-threaded architecture for media applications [10] and provide instrumentation data obtained by performance metering SYMPHONY in the proposed model. In section 5 we discuss how the proposed modeling scheme can be optimized to reduce the overall time spent exploring the design space and summarize the contributions of the paper in section 6.

2 PAMELA

PAMELA (PerformAnce ModEling LAnguage) is a process-algebraic computer systems description language aimed as a tool for the performance evaluation of parallel algorithms on parallel von Neumann architectures [3]. Similar to simulation languages such as SIMULA-DEMOS [1] and CSIM17 [12] a PAMELA model of a computational system can be compiled and simulated. Unlike other simulation languages, however, PAMELA is specifically designed to enable analytic techniques which allow for significant optimizations in

*This research was supported in part by the Department of Electronics, Government of India, under sponsored project DE-NMC/SP-054, and the TUD-IISc. collaboration project between Technical University, Delft, The Netherlands and The Indian Institute of Science.

1094-7256/97 $10.00 © 1997 IEEE

the simulation cost. The most extreme example of this feature is an entirely analytic compilation mode for a subset of PAMELA simulation models by which at compile-time a closed-form analytic expression is generated that approximates the simulated execution time within reasonable accuracy at very low computational cost. In the following we will informally present the subset of the language that is needed in the paper.

PAMELA supports the following data types, `process`, `resource`, `channel`, and `numeric`. The `numeric` data type comes with all the usual operators for binary, integer, and real-valued arithmetic, and is used to express time parameters, functions, and random distributions but is also used for indices in sequential and parallel loops.

The central concept in PAMELA is the interaction between *processes* (modeling the computations) and *resources* (modeling the computation providers). The `process` type is used to model computational tasks that entails workload (time delay) on some resource. The most basic PAMELA process is the `use` expression as in `use(r,t)` which simulates a time delay of `t` units while occupying the *resource* `r`. The `resource` type inherently implements the notion of mutual exclusion which is used to express the potential sequentialization (queuing) that may occur when multiple processes use (or "run on") the same processing resource. Resources come in FCFS type (First Come First Served, non-preemptive) to model, e.g., critical S/W sections, memories, disks, busses, and PS type (Processor Sharing, preemptive), typically used to model CPU schedulers. In order to model multiservers, resources can have a *multiplicity* larger than one.

In order for processes to be composed to meaningful simulation models the `process` data type comes with composition operators for sequential (infix: `;`, replicated prefix: `seq (..)`), parallel (infix: `||`, replicated prefix: `par (..)`), and conditional (`if-else`) composition. Parallel composition has a fork/join semantics which implies a mutual barrier synchronization at the finish for each task involved. PAMELA also includes a `while` construct which, however, is not used in this paper. In order to allow functional simulation PAMELA also includes (C) inlining facilities. Note that the inlined C code does not affect the simulated time.

For system models that require additional, non-fork/join-style condition synchronization patterns, `channel` type condition variables are used that come with `wait` and `signal` operators. For example, a process executing `wait(c)` will block until another process executes `signal(c)` where `c` is of type `channel`.

In order to allow for the application of compile-time analytic techniques the preferred modeling paradigm in PAMELA is *material-oriented* [8]. In contrast to the typically *machine-oriented* approach [8], found in other simulation languages (where each component is modeled as a process that reacts on stimuli on which it receives, processes, and sends data to other components), in PAMELA processes are used to specify the data processing in which all components that are traversed by the data in the course of its processing are modeled as *resources* that are temporarily `used`. For instance, a parallel algorithm comprising N parallel computation threads mapped on a P processor (multi-threaded) machine is modeled with N processes using either P `cpu` resources of one `cpu_pool` resource with multiplicity of P.

3 Modeling Multi-threaded Architectures

A multi-threaded architecture attempts to hide long latency operations to increase the utilization of the functional units of a processor, exploiting both fine-grain and coarse-grain parallelism. Long latency operations – taking multiple CPU cycles, occur due to either communication between processors or due to memory accesses. A quantitative analysis of a processor design should therefore involve scheduling instructions with the computing, communicating and storage elements as parameters. Since the aim of the modeling process is to derive near optimal values for each of these parameters, it is imperative that the application be specified so that *all* parallelism is exposed. We propose a modeling scheme for multi-threaded architectures with parameterized computing, communicating and storage elements in an unified framework that captures both coarse grain and fine grain parallelism in the application.

For our discussion, we take the view that a *process* is an *actor* and therefore refer to process and actor interchangeably through the paper and will make a distinction between the two only when necessary. An *actor* is a program entity with well defined firing rules. An actor can have many threads of execution that are scheduled dynamically. A *thread* is a statically ordered sequence of instructions that realizes a function.

In DSP applications, programs operate on data streams and at any instant in time several iterations of program modules are simultaneously active. Instruction level parallelism exists within and across threads in an actor, whereas coarse grain parallelism exists across actors belonging to different iterations. Limiting the synchronization losses due to fine grain and coarse grain parallelism can contribute to the overall performance of the architecture. Fine grain synchron-

ization losses can be minimized by overlapping execution of several threads in an actor, whereas coarse grain synchronization losses can be minimized through an optimal schedule of actors in a multi-threaded architecture.

Fine grain synchronization losses can be attributed to two reasons, *viz.* resource sharing within a processor and local data dependency between threads. The former is an artifact of the architecture, whereas the latter is an artifact of the algorithm. A suitable modeling of the architecture can be useful in identifying such limitations and take corrective measures by changing the parameters of the architecture.

Coarse grain synchronization is necessary to resolve global data dependencies. Coarse grain synchronization losses commonly arise due to non-optimal mapping and scheduling of actors onto the architecture. From a system architecture perspective, it can be argued that mapping and scheduling can therefore have a significant impact on the size of shared memory necessary for realizing an application. This is because, in DSP applications, where streams are processed, we need efficient mechanisms to reuse memory. A shared memory location can be reused only when its data has been consumed. Clearly, the schedule of an actor determines the lifetime of a produced data in memory. When the lifetime for every data item is large, we need to provide larger memory. A suitable modeling of the architecture can therefore facilitate evaluating alternate mapping and scheduling of actors over the entire parameter space of the architecture.

As mentioned earlier, we adopt a material oriented approach to model multi-threaded architectures, where the parallel system is modeled as viewed by the application. In this approach the architecture is represented by a parameterized set of passive resources that can be acquired and released in a controlled fashion. The parallel program is therefore a set of active processes with a producer/consumer relationship in a manner that relates to the flow of data through the program modules.

The modeling trajectory we follow involves the following steps.

1. **Resource Definition**: Define a parameterized set of resources in PAMELA that define all resources in the architecture, such as functional units, CPUs, communication controllers, memory, registers.
2. **Code Generation**: Generate a assembly code of the application at hand in terms of the instruction set architecture of the processor.
3. **Generate Traces**: Generate instruction traces for a set of "benchmark" data assuming a single uni-threaded processor, thereby transforming the application to that of a flattened DAG with no control statements.
4. **Re-target Trace**: Run a pre-processor on the trace to generate architecture specific PAMELA code that captures both fine grain and coarse grain parallelism in the application.
5. **Evaluate**: Execute the PAMELA code and iteratively arrive at the optimal numbers for the parameters associated with individual resources in the architecture.

In the following section we provide a walk through the modeling trajectory mentioned above using SYMPHONY [10] as a representative multi-threaded architecture.

4 SYMPHONY: A Case Study

SYMPHONY is an architecture template suitable for several applications within a domain. The basic computation entity in SYMPHONY is a *thread*. SYMPHONY can execute multiple threads concurrently, and is therefore a multi-threaded architecture. By associating with each *actor* a *closure* [4] which defines the set of input tokens necessary for an actor to fire, SYMPHONY provides architectural support necessary for dataflow process networks [9] which is a special type of Kahn process networks.

4.1 Architecture

SYMPHONY consists of symphony processors (SPs) in a linear array. SYMPHONY can also serve as an embedded system, in which case it is controlled by a host processor as shown in figure 1 and the host processor could (in principle) be another SP or any other processor. A set of interconnected SPs (analogous to PEs) operate in a cooperative fashion to implement a program. Program modules that have a strong producer-consumer relationship can be mapped into neighboring SPs hence utilizing SYMPHONY as a processor pipeline.

The host processor addresses each SP via the processor control bus (PCB) (refer figure 1). The PCB is used by the host processor to program the SPs. The host processor maps a part of a global memory address space to each SP. Special instructions are then issued by the host so that the SPs recognize the mapped address ranges. The host processor also initializes the program counter of each SP and then issues commands to the SPs to commence operation. SPs are laid out in a linear array and hence adjacent SPs communicate over links called local data bus (LDB). Each

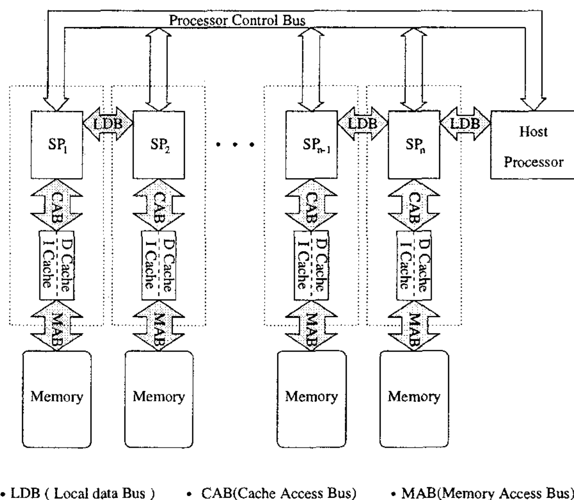

Figure 1: SYMPHONY: Organization of the Machine

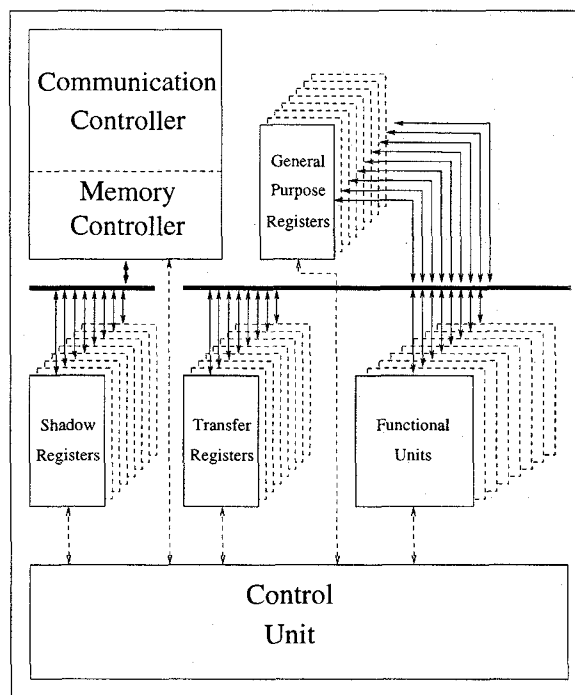

Figure 2: Inside a SYMPHONY Processor

SP has a communication controller (CC) and on-chip instruction and data caches and one or more functional units (FUs). The CC also houses the memory controller. (See figure 2.) All cache accesses take place under the supervision of the memory controller on the cache access bus (CAB). All memory accesses to an SP take place under the control of the CC. An SP can also optionally house on-chip shared single assignment memory.

Each SP can have one or more register files consisting of eight 32-bit registers. Each register file has 2 read ports and 1 write port. A set of small register files that can be managed with the help of compiler techniques is much easier to implement than a large register file with multiple read and write ports [2]. This has been the main motivation behind having a set of small register files in SYMPHONY. Data communicated between neighboring SPs are written onto a set of communication registers. Each SP can have one or more of such communication register files called *transfer registers.* The specific number of transfer registers required for a closure is programmable, depending on the requirements of the application. Every transfer register T has a corresponding shadow register SH as shown in the figure 2 (The SH set of registers are named so to indicate that they are the *"shadows"* of the corresponding T registers). Fine-grained communication mentioned above is achieved using these registers.

During an actor execution the T set of registers is assigned a *red* color to indicate that these registers are currently in use by the FUs of the SP. The SH registers are assigned a color *black* to indicate that they are currently being written into by the neighboring SPs with values that might be used subsequently. The CC can be programmed with a set of configuration instructions to partition the SH set of registers such that subsets T_{left} and T_{right} correspond to communication from the left and right neighbors respectively.

An explicit `switch` instruction has to be executed when the values that have been written into the SH registers are to be used. When such an instruction is executed the current SH registers become their corresponding T counterparts and vice-versa *i.e* the red and black register sets are swapped. Thus it is implicit that the SH set of registers cannot be accessed by the programmer directly.

One issue here, is that, when does the SP know that it can execute a switch instruction without losing data? To alleviate this problem the program should initially set a mask using a `setmask` instruction indicating the set of T registers whose values will be needed in the next actor that is going to be scheduled onto a particular SP (this is further elucidated below). When there is a write to one of these registers a bit is updated indicating that a value has been written into it. On a subsequent switch instruction the processor will block if some of the values have not been updated as yet. Thus the switch instruction can be used for synchronization between actors.

A point to be noted here is that, communication is fine-grained and takes place *implicitly* as opposed to *explicit* communication that takes place in conventional architectures. By implicit communication we mean that no extra instructions are necessary for posting values between processors.

4.2 Performance metering synchronization losses

SYMPHONY operates on multiple data streams simultaneously. The various data streams can pertain to audio, video, text and image for instance. Input to the system are samples, and these samples have an average input arrival rate. If the rate is $\mathcal{R}$, then $t_r = \frac{1}{\mathcal{R}}$ is the time interval between the arrival of two consecutive samples.

Program modules are composed of a collection of actors with a dataflow relationship between them. These actors act on streams of data. Only a window of a stream is used at any instant of time. The window slides on the stream every t_r cycles when new input to the system arrives. The *window of reference*, is this part of the stream of data being used by the actors in the system. Coarse grain parallelism is exploited across iterations within the window, whereas fine grain parallelism is exploited within an iteration. In order to performance meter synchronization losses in SYMPHONY we will define the following terms.

1. Let I denote the input stream to the system and I(j) denote the jth set of input tokens. `Iteration(j)` is then an instance of a program for which the input is `I(j)`. `Iteration(j)` may derive data dependent control from other Iterations, `Iteration(i)` and `Iteration(k)`, where $\mathtt{i} + \mathtt{p} = \mathtt{j}$ and $\mathtt{j} + \mathtt{q} = \mathtt{k}$, where i, j, k $\in Z$.

2. `O(j)` is the output corresponding to `I(j)` and is produced in `Iteration(j)`.

3. t_{max} is the maximum latency that an iteration can incur after exploiting fine grain parallelism in the architecture assuming no overheads for coarse grain synchronization.

4. Let γ denote the average fine grain parallelism exploited by the architecture in the presence of fine grain synchronization losses. The average work associated with an iteration can therefore be expressed as $\gamma * t_{max}$.

5. During t_{max} cycles, we define `W` as the total work that must be performed to produce `O(j)`. If $\mathcal{W}$ is the maximum work that the architecture can perform per cycle, called the work capacity of the architecture per cycle, then it is necessary to assume an architecture that satisfy:

$$t_r \geq \frac{\mathtt{W}}{\mathcal{W}} \tag{1}$$

Equation 1 ensures that the system is stable and doesn't accumulate tokens.

A thread in an actor may block because of the non-availability of data at any instant. An actor is said to block when all the threads constituting the actor are rendered ineligible to execute due to the same reason. An iteration derives data dependent control from other iterations. `Iteration(j)` may block on data produced by other iterations within the window of reference. When `Iteration(j)` is blocked, the relinquished computing resources are used by other iterations.

If we assume interleaved computation across iterations, the latency of an iteration will stretch beyond t_{max} because of data dependent control across iterations. It can be argued that for real-time applications, the effective work capacity of the architecture is therefore $\mathcal{W}$ - β, where β is the work capacity lost due to coarse grain synchronization.

We can therefore rewrite equation 1 as

$$t_r \geq \frac{\gamma * \mathtt{t_{max}}}{\mathcal{W} - \beta} \tag{2}$$

Thus, by performance metering the architecture model for different work capacities $\mathcal{W}$, it is possible to determine the actual values of γ and β and hence derive a lower bound for real-time constraints in the application.

4.3 Modeling

From an architecture perspective, we need to put together the various components of the architecture comprising three types of resources, *viz.* computation resources, communication resources and storage resources.

Each SP in SYMPHONY can comprise one or more FUs as shown in figure 2. Since the number of such units is parameterized, and shared by different threads in execution, the FU's naturally qualify as resource in PAMELA where all FUs are pooled into one resource with parameterized multiplicity. This facilitates optimal utilization of FUs by assuming dynamic assignment.

Transfer registers are declared as resources in PAMELA. A single file contains 16 T registers and 16 SH registers. The number of such register files is a parameter of the architecture and hence pooled into one resource with parameterized multiplicity.

It may also be noted that all data exchanged through transfer registers are known in advance from the static data dependencies between actors and therefore are a part of the coarse grain static data dependencies. On the other hand all dynamic data dependencies are resolved at runtime and this is achieved through a single assignment memory. This single assignment memory which can be reused after a lifetime $\mathcal{L}$ contributes to coarse grain synchronization losses in the architecture. The single assignment memory is modeled as channel in PAMELA. The size of the single assignment memory depends on the application.

Table 1: Representative actor code: The instruction format is identical to that of the DLX instruction set [5].

0x000010f4:	lhi	r7, 0	
0x000010f8:	addui	r7, r7, 20480	
0x000010fc:	add	r5, r0, r2	
0x00001100:	addi	r31, r0, 0	
0x00001104:	add	r4, r0, r5	
0x00001108:	add	r3, r0, r7	
0x0000110c:	addi	r2, r0, 0	
0x00001110:	lw	r1, 0(r8)	; 0x00005c30
0x00001114:	add	r1, r2, r1	
0x00001118:	ld	f2, 0(r3)	; 0x00005000
0x0000111c:	ld	f0, 0(r1)	; 0x00005900
0x00001120:	multd	f2, f2, f0	
0x00001124:	ld	f0, 0(r4)	; 0x00005a50
0x00001128:	addd	f0, f0, f2	
0x0000112c:	sd	0(r4), f0	; 0x00005a50

The other components of the SP, *viz.* general purpose registers (GPRs), buses (both internal and external), caches and local memory are modeled as resources in PAMELA.

In order to performance meter the application we start with an initial assignment of actors to the SPs in SYMPHONY. Recall each actor represents multithreaded code, and the schedule of instructions in the actor is determined dynamically. This is easily captured in PAMELA.

In order to keep the discussion tractable, we will restrict the application to that of performing overlapped transform coding of a image data [6]. Without delving into the details of the application, we will focus on a representative actor code written in terms of the instruction set architecture of an SP as listed in table 1. The trace corresponding to the code above would translate to an equivalent PAMELA code shown in table 2.

Each of the machine instructions is modeled in terms of the above-mentioned resources as in the process equality `add(r1,r2,r3) = use(FU,k*clock)` where the resource `FU` models the pool of functional units and `k` denotes the addition latency in clock cycles. When this trace is retargetted into a program model, PAMELA defines a dynamic schedule for all instructions in the actor, based on the availability of resources. This is a very good abstraction for multithreaded execution of the threads in an actor, wherein threads that block on data relinquish resources making it available for other threads. Fine grain synchronization losses are accounted in PAMELA only when there are free resources, but not ready to run threads.

Table 2: PAMELA code corresponding to actor in table 1

```
1:{
2: lhi (r7, 0)
3: signal(sema_r7_2)
4:} ||
5:{
6: wait(sema_r7_2);
7: addui (r7, r7, 20480);
8: signal(sema_r7_6)
9:} ||
10:{
11: add (r5, r0, r2);
12: signal(sema_r5_5);
13: signal(sema_r2_7_0)
14:} ||
15:{
16: addi (r31, r0, 0)
17:} ||
18:{
19: wait(sema_r5_5);
20: add (r4, r0, r5);
21: signal(sema_r4_13);
22: signal(sema_r4_15)
23:} ||
24:{
25: wait(sema_r7_6);
26: add (r3, r0, r7);
27: signal(sema_r3_10)
28:} ||
29:{
30: wait(sema_r2_7_0);
31: addi (r2, r0, 0);
32: signal(sema_r2_9)
33:} ||
34:{
35: lw (r1, 0, r8);
36: signal(sema_r1_9)
37:} ||
38:{
39: wait(sema_r1_9);
40: wait(sema_r2_9);
41: add (r1, r2, r1);
42: signal(sema_r1_11)
43:} ||
44:{
45: wait(sema_r3_10);
46: ld (f2, 0, r3);
47: signal(sema_f2_12)
48:} ||
49:{
50: wait(sema_r1_11);
51: ld (f0, 0, r1);
52: signal(sema_f0_12)
53:} ||
54:{
55: wait(sema_f2_12);
56: wait(sema_f0_12);
57: multd (f2, f2, f0);
58: signal(sema_f0_13_0);
59: signal(sema_f2_14)
60:} ||
61:{
62: wait(sema_f0_13_0);
63: wait(sema_r4_13);
64: ld (f0, 0, r4);
65: signal(sema_f0_14);
66: signal(sema_00005a50_15_0)
67:/* Signal on the release of
68: * a memory location
69: * 0x00005a50 */
70:} ||
71:{
72: wait(sema_f0_14);
73: wait(sema_f2_14);
74: addd (f0, f0, f2);
75: signal(sema_f0_15)
76:} ||
77:{
78:/* Wait for the release of
79: * the memory location
80: * 0x00005a50 */
81: wait(sema_00005a50_15_0);
82: wait(sema_r4_15);
83: wait(sema_f0_15);
84: sd (0, r4, f0)
85:}
```

Similarly, when all threads in an actor block on data, threads from a new actor are scheduled in PAMELA. When threads from all actors block on data, the synchronization losses accounted in PAMELA are those due to global dependencies, and are usually dynamic in nature and influence the exploitation of coarse grain parallelism. The various factors that can contribute to such synchronization losses can be attributed to:

1. **Sharing of communication resources local to an SP:** This is the case when multiple memory accesses are sequentialized over the processor-memory bus.
2. **Sharing of communication resources between two SPs:** This is due to communication latency between two SPs, when the data produced in SP(i) is consumed by a thread in SP(j).
3. **Sharing of storage resources:** In DSP applications that operate on data streams, memory can be re-used in a cyclo-static fashion. This however depends on the lifetime $\mathcal{L}$ of the data produced. In particular, when coarse grain parallelism is exploited in actors across iterations, the lifetime of a data value can directly restrict the extent to which coarse grain parallelism can be exploited in the application.

Now the synchronization losses as accounted in PAMELA can be used to evaluate various architectural options over the entire design space and hence arrive at numbers for the parameters in the architecture. The various architectural options for overlapped transform coding of a image is given in table 3. Table 3 gives a summary of the various architecture options for performing an example application, the overlapped transfer coding of images, in the SYMPHONY framework. The table shown is for the execution of a 8×16 transform. The total number of FU operations in the application for 1 iteration is 3559 cycles. The number of load/store operations performed is 1280 cycles. For accounting we attribute 1 cycle latency for add and logical operations. The multiply operations takes 7 cycles. However a multiply instruction can be issued every cycle (multiplier is pipelined). The results indicate that in this particular type of application which predominantly operates on vectors, performance scales with an increase in the number of FUs to a cross-over point where in the load-store bottle neck surfaces. This also indicates that in order to achieve a high throughput fast memory can be considered. Also, in the case of an embedded system one can consider special memory for interfacing with the master processor which pumps input to the system.

5 Discussions

Modeling an architecture, and performance metering it can serve two purposes. One, it can enhance the utilization of resources in the architecture; two, it can offer a platform to optimize applications with real-time performance constraints. From the previous discussions it is clear that minimizing synchronization losses leads to overall performance gains.

From equation 2, it is evident that fine grain synchronization loss γ can be reduced significantly through optimal schedule of threads in the actor. Since the threads have a dynamic schedule, it is appropriate to assume an architecture that has mechanisms to provide hardware and software support for scheduling threads without affecting the work capacity $\mathcal{W}$ of the architecture. It may be worth noting that in the application discussed in the previous section, every machine instruction is a PAMELA process and all contend for resources. The fine grain schedule is influenced only by condition synchronization and mutual exclusion, and are artifacts of the application and architecture respectively. As mentioned previously, this approach does not inhibit any dynamic parallelism that is present in the application, and this can be done without the aid of a compiler.

Table 3: Exploring the Design Space

No. of SPs	No. FUs per SP	Latency with 1 Load/Store Unit	Latency with 2 Load/Store Units
1	1	3559 cycles	3559 cycles
	2	1781 cycles	1781 cycles
	4	1282 cycles	891 cycles
	8	1282 cycles	642 cycles
	16	1282 cycles	642 cycles
2	1	1755 cycles	1755 cycles
	2	877 cycles	877 cycles
	4	744 cycles	439 cycles
	8	744 cycles	363 cycles
	16	744 cycles	363 cycles
4	1	918 cycles	918 cycles
	2	459 cycles	459 cycles
	4	336 cycles	230 cycles
	8	336 cycles	168 cycles
	16	336 cycles	168 cycles

Coarse grain synchronization on the other hand are concentrated at the actor entry/exit level. Synchronization losses β can therefore lead to loss of work capacity and hence affect performance. Also, it is reasonable to assume that coarse grain synchronization losses do not directly influence fine grain synchroniz-

ation losses. Therefore an efficient way to explore the design space is to adopt a hierarchical decomposition of the application (problem) into actors, and further each actor into threads. Since coarse grain synchronization is concentrated at the actor boundaries, it is appropriate that the problem at hand be first decomposed into actor and mappings of such actors onto the processors be explored in the modeling scheme (proposed earlier) with minimal coarse grain synchronization loss. Following this, each actor can in turn be performance metered for minimizing fine grain synchronization losses.

6 Conclusions

In this paper we set out to provide a method to model multi-threaded architectures using the PAMELA modeling language. The method facilitates exploring the design space of multi-threaded architectures for high performance applications with real-time constraints. We take an unified view of both fine grain and coarse grain parallelism in the application and performance meter the architecture for the application. We use the PAMELA modeling language to model multi-threaded architectures in a material-oriented fashion instead of a machine-oriented approach. Material-oriented modeling in PAMELA has the added advantage that compile-time analytical techniques can be applied to evaluate performance of algorithms in the architecture.

In this approach we start with a high level language description *viz.* C, C++ or Matlab. This description is then compiled, and a trace generated for a set of benchmark data in terms of the instruction set architecture of the processor. The trace generated is for a single uni-threaded, uni-processor system. This trace is pre-processed and retargetted to generate multi-threaded architecture specific PAMELA code. The resulting PAMELA code is executed to evaluate various architecture options over the entire design space iteratively.

This approach is new in that we simultaneously evaluate architecture and algorithm to satisfy real-time constraints. Further, the modeling scheme is identical for applications in which both fine grain and coarse grain parallelism must be exploited to meet the performance constraints.

We have demonstrated the suitability and simplicity of our approach in modeling multi-threaded architectures through a walk through example.

References

[1] G.M. Birtwhistle, *Demos - Discrete Event Modelling on Simula.* London: Macmillan, 1979.

[2] Henk Corporaal, Transport Triggered Architectures: Design and Evaluation, Ph.D thesis, TU. Delft, The Netherlands, Sept. 1995.

[3] A.J.C. van Gemund, "Performance prediction of parallel processing systems: The PAMELA methodology," in *Proc. 7th ACM Int'l Conf. on Supercomputing*, Tokyo, July 1993, pp. 318–327.

[4] Michael Halbherr, Yuli Zhou and Chris Joerg, "MIMD Style Parallel Programming Based on Continuation Passing Threads", Computation Structures Group Memo 355, Laboratory for Computer Science, MIT, April 8, 1994.

[5] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann, 1996.

[6] Richard Huesdens, Overlapped Transform Coding of Images: Theory, Application and Realization, Ph.D thesis, Delft, The Netherlands, March 1997.

[7] G. Kahn, "A semantics of a simple language for parallel processing", proceedings of IFIP Congress 1974, pp. 471-475, Amsterdam, 1974, Elsevier North Holland.

[8] W. Kreutzer, *System simulation, programming styles and languages.* Addison-Wesley, 1986.

[9] Edward A. Lee and T. M. Parks, "Dataflow Process Networks", Proceedings of the IEEE, Vol 83, No. 5, May 1995.

[10] S. K. Nandy, S. Balakrishnan, and Ed Deprettere, "SYMPHONY: A Scalable High Performance Architecture Framework for Media Applications", Technical Report, Dec. 1996 CAD laboratory, Supercomputer Education and Research Centre, Indian Institute of Science.

[11] Jack L. Lo, Susan J. Eggers *et al.* "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading", ACM Transactions on Computer Systems, 1997.

[12] H. Schwetman, "Object-oriented simulation modeling with C++/CSIM17," in *Proc. 1995 Winter Simulation Conference*, 1995.