Commission of the European Communities

****

ESPRIT III

PROJECT NB 6756

****

CAMAS

COMPUTER AIDED MIGRATION OF APPLICATIONS SYSTEM

****

CAMAS-TR-2.1.1.4
Parasol I Technical Report

****

Date: March 1994 — Review 3.0

ACE - Univ. of Amsterdam - ESI SA - ESI GmbH - FEGS - PARSYTEC - Univ. of Southampton

Contents

Arjan de Mes, Marcel Beemster, Jan de Ronde and Peter Sloot March 1994
Faculty of Mathematics and Computer Science
University of Amsterdam, The Netherlands

1 Proposed Goal

The goal of the Parasol project is make a rough estimate of the execution time of a program, given the program source code and the machine that will execute the program. This estimate must be given in a very short time, short enough to call the estimators response 'interactive'.

The traditional method of estimating or calculating the execution time of a program involves stepping through the program while the state transitions are being monitored and simulated. The behaviour of the (possibly virtual) machine must be specified in the greatest detail. Simulating the execution of a program on a virtual machine introduces at least one order of magnitude overhead to the real execution time of the program, which makes simulation unsuitable for the Parasol project.

The solution that is chosen, is to analyse the program source code and reduce it to a summation of execution times of the instructions in the program. To do this, the program code (Fortran) is translated into an expression called the 'Symbolic Application Description', SAD for short. Due to this translation, most of the context in which the instructions are executed is lost.

Another global restriction, is the type of programs that can be handled. Parasol can only handle 'SPMD' programs. This stands for Single Program Multiple Data-streams. In the context of Parasol this indicates that the programs, of which the execution time is to be estimated, have a common calculation cycle and a common communication cycle in all the processes. At one point in time all processors in the machine will be:

- calculating or waiting for the other processors to finish calculating
- communicating or waiting for the other processors to finish communicating

Part 1 of the Parasol project (Parasol I) is to provide information about the machine on which the code is executed. Information must be gathered concerning the arithmetic speed of the processor(s) and communication speeds. These speeds will be used free of context by the application reducing the SAD expression. A statement in the source code of the program being analyzed, such as `a = b + c`, will inevitably lead to the request "How much time is needed for one addition?" and "How much time is needed for one assignment?" These requests will come from part 2 of the Parasol project (Parasol II). This approach indicates that there is no information available as:

- if any of the variables are stored in registers,
- if any of the variables are stored in cache,
- and if the code being executed is in instruction cache

These are problems typical of approaches where the compilation and optimization process is skipped.

In this technical report the state of the Parasol I will be presented and described in detail.

2 Machine Model Description

The model of the machine must describe all parameters of the machine that could influence the performance of an SPMD program running on it. For Parasol I the parameters have been split into three main groups;

- a description of the processors in the machine,
- a description of the possible topologies,
- and a specification of the behaviour of the SPMD supporting functions in the PVM, Express and MPI programming models.

The parameters for the machine are collected in one datastructure, which can also be saved to disk. In this section the parameters of the machine model are presented and discussed, while traversing the BNF grammar for the file format. The complete BNF grammar for the file format is collected in Appendix A.

2.1 The Sections

The various sections in the machine are described as follows:

machine	::=	**begin machine** *nl machfield+ machsection+*
machsection	::=	*procsection* | *mcombsection* | *topsection* | *pvmsection* | *expresssection* | *mpisection*
procsection	::=	**begin processor** *nl procspec+* **end processor** *nl*
mcombsection	::=	**begin machinecombinations** *nl mcombspec+* **end machinecombinations** *nl*
topsection	::=	**begin topology** *nl topspec+* **end topology** *nl*
pvmsection	::=	**begin pvm** *nl pvmspec+* **end pvm** *nl*
expresssection	::=	**begin express** *nl expressspec+* **end express** *nl*
mpisection	::=	**begin mpi** *nl mpispec+* **end mpi** *nl*

The start symbol for the machine description is *machine*. From within this main section the processors (*procsection*), the machine combinations (*mcombsection*), the topologies (*topsection*) and the SPMD supporting programming models (*pvmspec*, *expresssection* and *mpisection*) can be specified. Multiple processors and topologies can be defined, the other sections are expected once each. If the sections that are expected once are defined more than once, the last definition will be the one used.

2.2 Global Machine Parameters

A few machine parameters do not fit in any of the above mentioned sections, or are considered 'global' to the machine:

machfield	::=	*id* | *name* | *proccount* | *topcount*
id	::=	**id =** *integer nl*
name	::=	**name =** *string nl*
proccount	::=	**processors =** *integer nl*
topcount	::=	**topologies =** *integer nl*

The *id*, *name*, *proccount* and *topcount* fields are each expected to be defined exactly once. If defined more than once, the last definition will be the one used. The definitions for *integer* and *string* can be found in Appendix A.

The *id* field defines a reference key for this machine. When applications working with the Parasol I model refer to a machine, it should be done using this integral number.

The *name* field defines a character string name for the machine. This field is for interactive applications that wish to present the name of the machine.

The *proccount* and *topcount* fields are defined for reference purposes; references are described in section 2.9. The *proccount* field defines the number of processor types in the machine, it has no reference to the number of processors in the machine. The *topcount* field defines the number of (possibly virtual) topologies specified for this machine.

2.3 Processor Parameters

For each processor in the machine, parameters may be defined. These parameters are split up into four types; timings of instructions (and intrinsic functions), specification of the

cache, combined processor parameters and parameters that do not fit in the former three types. The latter is discussed first:

procspec ::= *id* | *name* | *sloopspdup* |
begin timings *nl timings+* **end timings** *nl* |
begin cache *nl cache+* **end cache** *nl* |
begin combinations *nl combined+* **end combinations** *nl*
id ::= **id =** *integer nl*
name ::= **name =** *string nl*
sloopspdup ::= **smallloopspeedup =** *timing nl*

Within the processor specification, no parameters are expected to be defined more than once; if any of them is defined more than once, the last definition will be the one used.

The *id* and *name* parameters fulfill the same purpose for the processor, as they do for the machine in section 2.2.

The *sloopspdup* parameter defines the average speedup of instructions (as specified in section 2.3.1) if they are present in the instruction cache. How this parameter is to be used is discussed in section 3.2. The BNF reference *timing* is discussed in section 2.7; it is a measured value.

2.3.1 Instruction Timings

The performance of a machine is greatly influenced by the performance of the processors in the machine. The most frequently executed instructions by Fortran programs are arithmetic instructions, flow control instructions and intrinsic functions. Instead of presenting the BNF grammar for this section (which can be found in Appendix A), the separate parameters are discussed here according to these three categories.

The **arithmetic** instructions commonly used in Fortran programs are addition (and subtraction), multiplication, division, power with integer exponent and power with real exponent. For ease of use, assignments and memory transfers are also classified as arithmetic instructions. The seven arithmetic instructions listed above are present in the Parasol I machine model for the following numeric types: integer, float, double precision and complex.

The **flow control** instructions commonly found in Fortran programs are either simple goto statements, computed goto statements or procedure call statements. The overhead introduced by loops is also classified as a flow control parameter.

The Fortran **intrinsic functions**, except for those that do character handling, are all specified in the model for each datatype they operate on.

All the instructions listed here are measured using benchmarks, this is described later in section 3.1. The results of these measurements are timings; the number of seconds required to execute the instruction. These include instruction fetch and parameter fetch. For representation of measured values see section 2.7.

2.3.2 Cache Parameters

The cache parameters are purely provided to specify a more complete model of the machine.

cache ::= **size =** *integer nl* |
speed = *timing nl* |
speedratio = *expr nl*

The cache speed is already incorporated in the instruction timings and together with the cache access speed ratio over normal memory they provide little extra information for performance estimation. They do however provide for a better insight for humans as to how the machine is set up. The cache speed is the time needed to retrieve one word (the bandwidth) from cache.

The cache size is useful in combination with the small loop speedup factor presented in section 2.3. When estimating the performance of a program, the estimator might decide that the loop in the program is so small, it fits into the cache. It can then use the small loop speedup factor to see what happens to the performance. This is explained further in section 3.2.

2.3.3 Combined Processor Parameters

In this section parameters concerning the processor speed are compared to communication parameters. It is common practice to specify such ratios.

combined ::= *combparams* = *expr nl*
combparams ::= **ia_spd** | **im_spd** | **da_spd** | **dm_spd** | **dd_spd** | **ds_spd** | **dp_spd**

These parameters stand for integer addition, integer multiplication, double precision addition, double precision multiplication, double precision division, double precision sine and double precision power respectively divided by the maximum amount of bytes that can be output by the processor per second.

The BNF reference *expr* is explained in section 2.8. The expressions used in these definitions usually consist of a reference to a processor timing (section 2.3.1) multiplied by a reference to the topology (section 2.5). Defining these parameters in the processor section might not seem logical, since they combine parameters from this section with those from the topology section, but for ease of reasoning one might say: the maximum output speed is also processor dependant.

2.4 Machine Combinations

Very few parameters give a global impression of the machine. [Markatos92] presents the sum of the maximum transfer rates of all the processors as a useful index for the communications performance. This parameter is therefore specified in de model, though it provides little extra information for performance estimation. This parameter has the following BNF grammar:

mcombspec ::= **maxtotalspd** = *expr nl*

The BNF reference *expr* is explained in section 2.8.

2.5 Topology Parameters

Multiple topologies may be specified in one machine. In this manner virtual topologies can be specified and described per link.

topspec ::= *id* | *name* | *hardware* | *create* | *proccount* | *linkcount* |
begin procid *nl procident+* **end procid** *nl* |
begin linklist *nl link+* **end linklist** *nl*
id ::= **id** = *integer nl*
name ::= **name** = *string nl*
hardware ::= **hardware** = (**0** | **1**) *nl*
create ::= **creation** = *timing nl*
proccount ::= **processors** = *integer nl*
linkcount ::= **links** = *integer nl*

The above parameters are all expected to be defined once per topology. If any of them is defined more than once, the last definition will be used.

The *id* and *name* parameters fulfill the same purpose for the topology, as they do for the machine in section 2.2.

The *hardware* parameter defines if this topology is the hardware topology or a virtual topology. This parameter is not likely to be used in performance estimation, but indicates the default topology for the machine.

The *create* parameter defines the amount of time needed to setup this topology. If a program of which the execution time is to be estimated uses this topology, the creation time must be added. The BNF reference *timing* is discussed in section 2.7; it is a measured value.

The *proccount* field defines the number of processors used in this topology. This is opposed to the *proccount* field in the global machine parameters (section 2.2) where it indicated the number of types of processors in the machine. For each topology a processor type must be defined for every processor, this is discussed in section 2.5.1.

The *linkcount* field defines the number of links in this topology. Each link must be defined separately. This is discussed in section 2.5.2.

2.5.1 Processor Types in the Topology

A processor type description has the following BNF grammar:

procident ::= **p[***integer***] =** *integer nl*

For each processor – as defined by *proccount* in section 2.5 – a processor type must be defined. The processor number (the first *integer*) must be between one and the value for *proccount* (inclusive). The processor type (the second *integer*) must be defined in the same machine definition, though the processor type may be defined after the topology is defined.

2.5.2 Individual Link Descriptions

Each link in the topology is defined separately. They are listed in one block as defined in section 2.5. A link description has the following BNF grammar:

link ::= **begin link** *nl linkelt*+ **end link** *nl*
linkelt ::= **id =** *integer* | **latency =** *timing nl* |
speed = *timing nl* | **connect =** *integer*+ *nl*

For each link an **id** is specified. This is for reference purposes only and must be a number between one and the value of *linkcount* (inclusive) as defined in section 2.5. The links need not necessarily be defined in *id*-order.

The **latency** is a measured time (see section 2.7) and represents the time that passes between the moment of the send call and the moment that the entire message arrives (and can be used) at its destination.

The **speed** field defines the time required to send one byte over this link and is expressed in seconds.

The **connect** field defines connectivity of this link and is specified as follows: the first integer following the assignment defines the number of processors connected to this link. This should be a number greater than one. The integral numbers following the first specify the processor numbers that are connected. These numbers must be between one and the value for *proccount* (section 2.5).

2.6 SPMD Supporting Programming Models

To make a parallel program usable on many different types of computers, it is common practice to write such programs in a standard parallel programming model. Currently, the PVM, Express and MPI platforms are commonly accepted as standards. The Parasol I model defines the behaviour of the functions in these platforms that are needed for SPMD

application programs. This usually involves functions for send and receive, packing and unpacking the data before and after these calls.

The SPMD model functions that will be present in the final model represented by Parasol I will not be exactly what is specified here. The feasibility of representing all three models still needs further research. What is presented here is implemented in the model, but the fields are not yet filled in for any of the SPMD models. Until further research is done, the BNF grammar is as follows:

pvmsection	::=	**begin pvm** *nl pvmspec+* **end pvm** *nl*
pvmspec	::=	*pvmfunc* = *expr nl*
pvmfunc	::=	**sync** | **broadcast** | **multicast** | **send** | **receive** | **packbyte** | **packdouble** | **packfloat** | **packint** | **packlong** | **packstring** | **unpackbyte** | **unpackdouble** | **unpackfloat** | **unpackint** | **unpacklong** | **unpackstring**
expresssection	::=	**begin express** *nl expressspec+* **end express** *nl*
expressspec	::=	*expressfunc* = *expr nl*
expressfunc	::=	**sync** | **broadcast** | **multicast** | **send** **receive** | **exchange** | **combine** | **concat**
mpisection	::=	**begin mpi** *nl mpispec+* **end mpi** *nl*
mpispec	::=	*mpifunc* = *expr nl*
mpifunc	::=	**send** | **receive**

All the SPMD model parameters are specified with expressions (see section 2.8). These expressions will contain many references to other machine parameters since the SPMD functions are implemented using lower level instructions (represented in other sections of the Parasol I model).

2.7 Measured Values

A timing in the Parasol I model is specified by a real number specifying the time in seconds (except when explicitly defined differently). The measurement of the execution time of one instruction or other measurable action can vary due to clock grain size, system dependant optimization, cache hits and missed, and so forth. Therefore multiple samples of the instructions execution time are taken and averaged. The standard deviation of these measurements is also calculated.

So a timing in the Parasol I model consists of an average time and to provide an impression of the accuracy of the measurement, the standard deviation may also be specified.

timing	::=	*float float*?

The first value of the timing is the average value, the second value is the standard deviation of the measured samples.

2.8 Expressions with References

Expressions as machine parameters are used to define how the execution time of the function being described is constructed from other machine parameters. An expression in the Parasol I model has the following BNF grammar:

expr	::=	*expr binaryop expr* | *unaryop expr* | **(** *expr* **)** |
		function **(** *expr* **)** | *reference* | *float* | *integer*
binaryop	::=	**-** | **+** | ***** | **/** | **%** | ****** | **==** | **>=** | **>** | **<=** | **<** | **!=** |
		>> | **<<** | **&** | **|** | **^** | **&&** | **||**
unaryop	::=	**-** | **!** | **~**
function	::=	**abs** | **acos** | **asin** | **atan** | **cos** | **cosh** | **double** |
		exp | **floor** | **int** | **log** | **log10** | **round** | **sgn** |
		sin | **sinh** | **sqrt** | **tan** | **tanh**
float	::=	**-**?*digit*+**.***digit**(**e(-|+)**?*digit*+)?
integer	::=	*digit*+ | **0x***hexdigit*+
digit	::=	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
hexdigit	::=	*digit* | **A** | **B** | **C** | **D** | **E** | **F**

References (*reference*) are explained in section 2.9. These expressions can express a wide range of complexities. The precedence of the operators is the same as is defined in the C programming language.

An example expression with references could look like:
(4*pt(1).ila + 2*pt(1).ils + pt(1).goto) / pr(1).smallloopspeedup

This example could represent the execution time used for one pass through a non-terminating loop, with the code

```
10    a = b + c + d + e
      b = a + a
      goto 10
```

Thus: four integer additions and two assignments, both with local parameters. Added to this is one goto instruction and this loop code is considered to be in cache so the resulting time is divided by the small loop speedup factor.

2.9 References to Machine Parameters

References are basically pointers to other fields in the machine description.

reference	::=	**ma.(id|processors|topologies)** |
		pr(*integer***).(id|smallloopspeedup)** |
		pt(*integer***).***timeparams* |
		pc(*integer***).(size|speed|speedratio)** |
		po(*integer***).***combparams* |
		mc.maxtotalspd |
		to(*integer***).(id|hardware|creation|processors|links)** |
		tp(*integer***)(***integer***)** |
		tl(*integer***)(***integer***).(id|latency|speed|connect(***integer***))** |
		pv.*pvmfunc* | **ex.***expressfunc* | **mp.***mpifunc*

A few BNF terms are not mentioned here, they can be found in Appendix A. The **ma** prefix stands for a global machine parameter (section 2.2). The fields from that section can be referred to using this prefix. The **pr**, **pt**, **pc** and **po** prefixes are references to a processor definition; global (2.3), timing (2.3.1), cache (2.3.2) and combined (2.3.3) respectively. To refer to the correct processor, these four prefixes must be followed by a processor id number.

The **mc** prefix refers to the combined machine parameters. The only field that can be defined in the *mcombsection* section is the **maxtotalspd** field.

The **to**, **tp** and **tl** prefixes are references to the topology definition; global (2.5), processor definitions (2.5.1) and link descriptions (2.5.2) respectively. The integral number following

the prefix defines the topology id number referred to. The **tp** prefix is followed by a second integral number; this is the number of the processor in the topology. It refers to the processors type. For the **tl** prefix the second integral number refers to the link id number being referenced.

Referring to the connectivity of a link needs some extra attention. A connectivity reference with parameter 0 refers to the number of processors connected to the link. Other references to the connectivity, with a number between one and **tl(...)(...).connect(0)**, return what processors in the topology are connected to the link.

The **pv**, **ex** and **mp** prefixes refer to the PVM, Express and MPI programming models respectively; of which only the SPMD supporting functions are modelled. References to expressions return the value represented by the expression.

3 Using the Machine Model

This section describes how the Parasol I model will be filled with data representing a (possibly virtual) machine and how Parasol I will interface with Parasol II.

3.1 Acquiring Accurate Timings

To explain how the correct data is acquired for the Parasol I model, an example is presented in pseudo code; the actual code is in Fortran:

```
// first run

start = gettime()
loop index = 1 to samplesize
    value_A = data[index][1]
    value_B = data[index][2]
    value_C = data[index][3]
    result = value_A + value_B
endloop
end = gettime()
firsttime = end - start

// second run
// determine increment due to extra addition

start = gettime()
loop index = 1 to samplesize
    value_A = data[index][1]
    value_B = data[index][2]
    value_C = data[index][3]
    result = value_A + value_B + value_C
endloop
end = gettime()
secondtime = end - start

// addtime is average time needed for an add instruction
// where one argument is retrieved from memory

addtime = (secondtime - firsttime) / samplesize
```

In the following subsections the problems in measuring the timings are discussed.

3.1.1 The Individual Instruction

In the Parasol I model, the abstraction level makes it difficult to separate the time needed to execute an instruction into all the phases that are passed at the machine level. Instead, the timings presented for the instructions include instruction fetch (from memory) and parameter fetch (if present from data cache).

Whether or not an instruction is in the instruction cache depends on the context in which the instruction is executed; a loop that fits into instruction cache will execute faster than a loop that exceeds said cache. If loops fit into instruction cache depends on different factors; the amount of source code in the loop, what the compiler does with it and of course the size of the instruction cache.

Parasol does not work with compiled code, but with the Fortran source code. Therefore it is very hard to determine how much Fortran code will result in how much machine code. The way Parasol works around this is by defining a speedup factor for 'small loops'. In the example code (at the beginning of section 3.1), the loop size is increased by repeating the lines with `result =` a large number of times, thereby forcing the loop code to become larger than the cache.

The most accurate solution would be providing a context in which the instruction is being executed. A context is easily made available during simulation, but that is not the intention of this project. In Parasol II all instructions are seen without context.

The timings in the model incorporate argument fetch times. These arguments can be in registers, data cache or memory. Where the arguments, needed for an instruction, are, depends on the source code, the compiler, the operating system (are application programs permitted to operate the cache?) and again: the size of the data cache. Because compiler actions are not modelled, there is no way to determine when data is in registers. To determine what arguments are in data cache context is needed.

To get realistic behaviour of the model nonetheless, an average value for the time needed to execute the instruction should be used. To get the perfect average, one would have to know how many cache hits, misses and register optimizations there are in the application being modelled on the machine being modelled. This is very unfortunate since the goal is to model a large group of SPMD application programs. This implies having to run the application program to get enough information about it, to estimate its execution time.

In the Parasol project there is no special solution for this problem. The example at the beginning of section 3.1 shows that it is very likely that all arguments to the instruction are fetched from cache (when the machine being modelled has a data cache).

Where the arguments to an instruction can be found also depends on where the programmer defined the variable being used. In the model, a distinction is made between local and global variables. In the above example, this distinction can be made by defining the variables `value_A`, `value_B`, and `value_C` in or outside the subroutine. In Fortran, outside the subroutine means using a `COMMON` block to share the variables.

For the Fortran intrinsic functions modelled no distinction is made as to the locality of the parameters. Fortran implements these functions with a function call

Instructions may take more or less time, depending on the value of their arguments. With an instruction such as `mul` it is apparent that the evaluation can be done faster when one of the arguments is 1 (one) or 0 (zero). Based on the lack of context attained in the Parasol II project, the assumption is made that it is not necessary to consider this type of exception when measuring the execution time for these instructions. Therefore, when measuring time for low level instructions, only more complex arguments are passed to the instructions.

A stable and usable average instruction time can only be measured using different arguments to the instruction. The average should then be taken over the collective time used by these instructions. This average is not guaranteed to be the same as the average of the corre-

sponding application program. Large arrays containing the precalculated random numbers are used to avoid having non-deterministic behaviour of functions such as `random()` in the loop. The time to index an array has been taken into account in the example (at the start of section 3) by executing the same array references in both loops, so `value_C` is assigned values in the first loop that are never used.

3.1.2 The Timing Call

The system call to retrieve the time also occupies some time. In the example presented above, these delays are eliminated because they are present in the same form in both loops. The difference in execution time of the loops is used as time needed to execute the instructions in the loop thereby ignoring time occupied during time calls.

A problem with the clock is the grain size. The shortest measurable time greater than zero may be so large that it exceeds the execution time of an instruction. For example consider a Transputer (say Inmos T805) with a clock speed of 20 MHz. The high priority timer on the Transputer makes one tick every microsecond, i.e. one tick every 20 processor cycles. This makes it difficult to measure if an instruction (e.g. `add`) takes 1, 2, or many more cycles to execute.

As explained in section 3.1.1 the loop size had to be increased to overflow the instruction cache. The problem of the clock grain size does not need further attention because the instructions needed to fill a cache usually occupy quite a few clock ticks. When no instruction cache is present (as in the Inmos T805) the clock grain size forces more instructions into the loop or more iterations of the loop.

3.1.3 Interference by Parallel Tasks

All machines that are likely to be modelled in the Parasol project have the possibility to run multiple processes on one processor at 'the same' time. More processes running on the same processor will reduce the time given to a process. Task switching may be very cheap, and other processes might be idle but there is still a delay imposed by parallel execution of jobs. It is therefore essential to use time calls that return the number of time used by the measuring process only, or to make sure that there are no other processes running on the machine that wouldn't be running when the application, whose performance is to be estimated, is running.

The model should be filled with realistic figures; how fast would the machine work under a normal workload. On almost al machines there will be an operating system or some other interface layer. This layer is very likely to start parallel jobs, or cause delay in execution speed at unpredictable occasions. Asynchronous communication, for example, is usually implemented by starting a separate process to send or receive the message. Until this process terminates, it will delay other processes on the processor; when communicating, it causes a greater delay than when it is waiting for an acknowledgement or a connection.

When measuring timings for instructions, the machine should be monitored very carefully (but so that no cpu time is used).

3.2 Interaction with the Machine Model

The setup of the Parasol project is to make an interactive performance estimator. Part of the interaction has to come from an interface between the user (the software developer) and the machine description. The other part of the interactive interface is that between the user and Parasol II, which handles and analyses the Fortran code. The user interface that interacts with the machine description is called the Parasol I User Interface (for short PIUI). The PIUI is based on Tk/Tcl combined with the C programming language. Tk/Tcl is a programming system for developing and using graphical user interface (GUI) applications [Ousterhout93]. The Tk/Tcl used for the PIUI runs under X-windows.

3.2.1 The Interface Between PIUI and the User

The PIUI enables the user of the interactive environment to change parameters in the machine and observe what happens to the performance. The user can change the machine parameters at different levels; the time needed to execute one specific instruction can be modified, but the user may also choose to change the overall floating point performance, the overall multiplication performance, or even all the processor timings, which would suggest a higher clock speed. In this section the parameters and groups of parameters that may be changed are discussed. What the graphical interface looks like is not discussed.

3.2.2 Changing Processor Parameters

The processor parameters can be changed individually or in groups.

Changing one processor parameter is possible in the PIUI. It can represent an optimization in the way the processor handles that specific instruction. The processor parameters may also be changed in groups. The effect of changing a group of parameters can be more meaningful than changing just one. In the PIUI several – non disjunct – groups are defined:

- The machine parameters that operate on local instructions. Modifying this group could suggest better register allocation for the variables, or – because the effect on the performance is presented interactively – provide the user with an idea of the gain that would be achieved with more or less variables being defined locally.
- The machine parameters that operate on global instructions. Depending on the implementation this could indicate memory access speeds.
- The machine parameters that operate on one mathematical function such as `add` (for example). This could indicate a change in the performance of the arithmetic unit (ALU or FPU).
- The machine parameters that work on one data type. This could be the effect of changes in the way the data type is handled or represent.
- All the processor timings. An effect typical of changing the clock speed. Changing the cache speed should be done by changing one single parameter: the **smallloop-speedup**.

The explanations of what the groups might represent is not complete; many other meanings may be identified, the user is considered to have considerable knowledge of the system and source code being modelled and may have totally different intentions when changing parameter groups and viewing the results in performance.

The definition of the elements that can be changed with these groups is predefined. The user is not provided with the possibility to specify own groups.

3.2.3 Changing Topology Parameters

The topology parameters can also be changed individually or in groups.

Changing one topology parameter is possible in the PIUI. Increasing the speed of one link could relieve a bottleneck or decreasing the latency might represent an increased efficiency in the routing algorithm. The PIUI provides the following options for changing topology parameters in groups:

- All the link speeds.
- All the link speeds for one type of processor.

- All the link latencies.
- All the link latencies for one type of processor.
- The connectivity can *not* be changed using the PIUI. Topology connectivity must be specified by editing the machine description in memory or in the file. This can not be done graphically. Large topologies with many links can be generated using a small dedicated program.

3.2.4 Changing SPMD Supporting Model Parameters

The description of the SPMD supporting instructions in the Parasol I model is done using expressions (as explained in sections 2.6 and 2.8). Providing a graphical interface to modify these complex expressions is far sought and would probably not be easy to use. Actually changing these parameters would indicate a modification in the implementation of the SPMD supporting model on the machine.

As a result, it is not possible to change the expressions that represent the behaviour of the SPMD model functions. What can be changed is a multiplication factor for a single function in the SPMD model, or all the functions in the entire SPMD model. The latter would indicate a more or less efficient implementation of said model.

3.2.5 The Interface Between PIUI and Parasol II

The interaction with Parasol II will be done using standard procedure calls. This will eventually be implemented using UNIX Remote Procedure Calls (RPC). Parasol II is able to access all the fields defined in the Parasol I model. It can also receive a signal from the PIUI to indicate that a field in the model has changed. Parasol II can then request the new value and evaluate the program that it is analyzing to present a new performance estimate through its own GUI.

4 Current state of Parasol I

The Parasol I machine model is currently in a beta testing phase. Operations on the machine model data structure and file format are supported and functioning.

Several problems in obtaining correct machine timings have been identified. The timings for the arithmetic parameters have been completed for several Sun Sparc stations. The code to measure flow control instructions and the Fortran intrinsic functions is being constructed. When all processor parameters have been measured for the Sun Sparc station, the code can easily be ported to measure the parameters for the Inmos T805 Transputer.

The topology and SPMD supporting models still need to be studied before the parameters can be filled in. The topology for a network of Sun Sparc workstations is easily specified. The throughput and latency will be measured under a normal (and realistic) workload.

The PIUI is still under construction. The user interface concerning the processor parameters is in alpha testing phase. The interaction with the topology parameters and the SPMD supporting models is not yet implemented.

The interaction with Parasol II has been implemented for the processor parameters. This can be extended easily when Parasol II is ready to accept more fields for the machine description. The PIUI is not yet capable of signalling Parasol II that machine parameters have been changed.

A The BNF Grammar

machine ::= **begin machine** *nl machfield+ machsection+*
end machine *nl*
machfield ::= *id* | *name* | *proccount* | *topcount*
id ::= **id =** *integer nl*
name ::= **name =** *string nl*
proccount ::= **processors =** *integer nl*
topcount ::= **topologies =** *integer nl*
machsection ::= *procsection* | *mcombsection* | *topsection* |
pvmsection | *expresssection* | *mpisection*
procsection ::= **begin processor** *nl procspec+* **end processor** *nl*
procspec ::= *id* | *name* | *sloopspdup* |
begin timings *nl timings+* **end timings** *nl* |
begin cache *nl cache+* **end cache** *nl* |
begin combinations *nl combined+* **end combinations** *nl*
sloopspdup ::= **smallloopspeedup =** *timing nl*
timings ::= *timeparams* **=** *timing nl*
timeparams ::= **absc** | **absd** | **absi** | **absr** | **aimagc** | **aintd** | **aintr** | **anintd** | **anintr** |
argld | **arr1** | **arr2** | **arr3** | **arr4** | **arr5** | **arr6** | **arr7** | **asind** | **asinr** |
atand | **atanr** | **cga** | **cgd** | **cge** | **cgm** | **cgs** | **cgt** | **cgx** | **chari** | **cla** |
cld | **cle** | **clm** | **cls** | **clt** | **clx** | **cmplx2d** | **cmplx2i** | **cmplx2r** |
cmplxc | **cmplxd** | **cmplxi** | **cmplxr** | **conjgc** | **dblec** | **dbled** | **dblei** |
dbler | **dga** | **dgd** | **dge** | **dgm** | **dgs** | **dgt** | **dgx** | **dimd** | **dimi** | **dimr** |
dla | **dld** | **dle** | **dlm** | **dls** | **dlt** | **dlx** | **dprodr** | **expc** | **expd** | **expr** |
gcom | **goto** | **iga** | **igd** | **ige** | **igm** | **igs** | **igt** | **igx** | **ila** | **ild** | **ile** | **ilm** |
ils | **ilt** | **ilx** | **intc** | **intd** | **inti** | **intr** | **lga** | **lgcc** | **lgdc** | **lgic** | **lgrc** |
lla | **llcc** | **lldc** | **llic** | **llrc** | **log10d** | **log10r** | **logc** | **logd** | **logr** |
loopi1 | **loopin** | **loopoh1** | **loopohn** | **maxd** | **maxi** | **maxr** | **modd** |
modi | **modr** | **nintd** | **nintr** | **pcall** | **realc** | **reald** | **reali** | **realr** |
rga | **rgd** | **rge** | **rgm** | **rgs** | **rgt** | **rgx** | **rla** | **rld** | **rle** | **rlm** | **rls** | **rlt** |
rlx | **signd** | **signi** | **signr** | **sinc** | **sind** | **sinhd** | **sinhr** | **sinr** | **sqrtc** |
sqrtd | **sqrtr** | **tand** | **tanhd** | **tanhr** | **tanr**
cache ::= **size =** *integer nl* |
speed = *timing nl* |
speedratio = *expr nl*
combined ::= *combparams* **=** *expr nl*
combparams ::= **ia_spd** | **im_spd** |
da_spd | **dm_spd** | **dd_spd** | **ds_spd** | **dp_spd**
mcombsection ::= **begin machinecombinations** *nl mcombspec+*
end machinecombinations *nl*
mcombspec ::= **maxtotalspd =** *expr nl*
topsection ::= **begin topology** *nl topspec+* **end topology** *nl*
topspec ::= *id* | *name* | *hardware* | *create* | *proccount* | *linkcount* |
begin procid *nl procident+* **end procid** *nl* |
begin linklist *nl link+* **end linklist** *nl*
hardware ::= **hardware =** (**0** | **1**) *nl*

create ::= **creation =** *timing nl*
linkcount ::= **links =** *integer nl*
procident ::= **p[***integer***] =** *integer nl*
link ::= **begin link** *nl linkelt+* **end link** *nl*
linkelt ::= *id* |
latency = *timing nl* |
speed = *timing nl* |
connect = *integer+ nl*
pvmsection ::= **begin pvm** *nl pvmspec+* **end pvm** *nl*
pvmspec ::= *pvmfunc* **=** *expr nl*
pvmfunc ::= **sync** | **broadcast** | **multicast** | **send** | **receive** |
packbyte | **packdouble** | **packfloat** | **packint** |
packlong | **packstring** | **unpackbyte** | **unpackdouble** |
unpackfloat | **unpackint** | **unpacklong** | **unpackstring**
expresssection ::= **begin express** *nl expressspec+* **end express** *nl*
expressspec ::= *expressfunc* **=** *expr nl*
expressfunc ::= **sync** | **broadcast** | **multicast** | **send**
receive | **exchange** | **combine** | **concat**
mpisection ::= **begin mpi** *nl mpispec+* **end mpi** *nl*
mpispec ::= *mpifunc* **=** *expr nl*
mpifunc ::= **send** | **receive**
expr ::= *expr binaryop expr* | *unaryop expr* | **(** *expr* **)** |
function **(** *expr* **)** | *reference* | *float* | *integer*
binaryop ::= **-** | **+** | ***** | **/** | **%** | ****** | **==** | **>=** | **>** | **<=** | **<** | **!=** |
>> | **<<** | **&** | **|** | **^** | **&&** | **||**
unaryop ::= **-** | **!** | **~**
function ::= **abs** | **acos** | **asin** | **atan** | **cos** | **cosh** | **double** |
exp | **floor** | **int** | **log** | **log10** | **round** | **sgn** |
sin | **sinh** | **sqrt** | **tan** | **tanh**
timing ::= *float float*?
reference ::= **ma.(id|processors|topologies)** |
pr(*integer***).(id|smallloopspeedup)** |
pt(*integer***).***timeparams* |
pc(*integer***).(size|speed|speedratio)** |
po(*integer***).***combparams* |
mc.maxtotalspd |
to(*integer***).(id|hardware|creation|processors|links)** |
tp(*integer***)(***integer***)** |
tl(*integer***)(***integer***).(id|latency|speed|connect(***integer***))** |
pv.*pvmfunc* | **ex.***expressfunc* | **mp.***mpifunc*
float ::= **-**?*digit+***.***digit****(**e(-|+)**?*digit+*)?
integer ::= *digit+* | **0x***hexdigit+*
digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
hexdigit ::= *digit* | **A** | **B** | **C** | **D** | **E** | **F**
string ::= **"***character+***"**
character ::= any character other than newline
nl ::= newline | vertical tab | **;**

References

[Andrews91] J. B. Andrews and C. D. Polychronopoulos. *An Analytical Approach to Performance / Cost Modeling of Parallel Computers*, Center for Supercomputing Research and Development University of Illinois at Urbana-Champaign, Journal of Parallel and Distributed Computing, 1991.

[Dimpsey91] R. T. Dimpsey and R. K. Iyer. *Performance Prediction and Tuning on a Multiprocessor*, Center for reliable and high-performance computing, University of Illinois at Urbana-Champaign, ACM, 1991.

[Markatos92] E. P. Markatos and T. J. LeBlanc. *Shared-Multiprocessor Trends and the Implications for Parallel Program Performance*, University of Rochester, Computer Science Department, Rochester, New York, May 1992.

[Muller93] H. Muller. *Simulating Computer Architectures*, Faculty of Mathematics and Computer Science, University of Amsterdam, Amsterdam, 1993.

[Ousterhout93] J. K. Ousterhout. *Tcl and the Tk Toolkit*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1993.

[Rosen76] S. Rosen. *Lectures on the Measurement and Evaluation of the Performance of Computing Systems*, Purdue University, Philadelphia, Pennsylvania, SIAM, 1976.

[Saavedra89] R. H. Saavedra-Barrera, A. J. Smith and E. Miya. *Machine Characterization Based on an Abstract High-Level Language Machine*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, IEEE Transactions on Computers, 1989.

[Sarkar89] V. Sarkar. *Determining Average Program Execution Times and Their Variance*, IBM Research, Watson Research Center, Yorktown Heights, New York, ACM, 1989.

[Sunderam89] V. S. Sunderam. *PVM: A Framework for Parallel Distributed Computing*, Department of Math and Computer Science, Emory University, Atlanta, Concurrency: Practice and Experience, 1989.

[Tanenbaum88] A. S. Tanenbaum. *Computer Networks*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.