Abstract Workload Modelling in Computer Architecture Simulation

A.D. Pimentel L.O. Hertzberger

Dept. of Computer Science
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam
The Netherlands
{andy,bob}@wins.uva.nl

Abstract

The Mermaid simulation environment facilitates the performance evaluation of a wide range of design options in MIMD *multicomputer architectures. Because the Mermaid architectural simulators are driven by events that are more abstract than real machine instructions, workloads must explicitly be modelled. In this paper, we provide an overview of the workload modelling techniques used within Mermaid. This includes the tracing of real programs and the stochastic modelling of application behaviour. Moreover, we present several validation and performance results indicating that our simulation methodology obtains good accuracy while being fairly efficient.*

1 Introduction

Simulation is a widely-used technique for evaluating the performance of computer architectures. It facilitates the scenario analysis necessary for gaining insight into the consequences of design decisions. The alternatives to simulation, namely analytical modelling and real hardware prototyping, are often less than adequate for this purpose. When modelling sophisticated architectures the analytical models may quickly become too complex, whereas hardware prototyping is too costly and time-consuming when evaluating a wide range of design options.

In order to provide a workbench for evaluating the performance of MIMD multicomputer architectures, we are developing the Mermaid simulation environment [15]. Originally, Mermaid was targeted for studying future series of Parsytec multicomputers (like the study performed in [14]). However, recent developments are increasingly geared towards generic support for architecture simulation.

Mermaid strives to support the performance evaluation of a wide range of architectural design options. For this purpose, it allows parameterization and configuration of the simulation models at different levels: from processor parameters, such as the instruction set and cache specifics, to switching and routing techniques in a message-passing communication network. Moreover, to make both accurate simulation and fast prototyping of multicomputers feasible, Mermaid features the ability to simulate at different abstractions levels. If the research objective is fast prototyping only, maximum accuracy is not required and therefore simulation can be performed at a high level of abstraction, yielding high simulation efficiency. In the situations where accuracy is required, however, the simulation is performed at a lower and thus more computationally intensive level of abstraction. Unlike many other simulation systems, we do not apply instruction-level simulation [6, 1, 18] at the lowest level of abstraction nor do we use *direct execution* [17, 3, 8, 9, 7, 18, 2] in which parts of the simulated code are directly executed on the host computer. Instead, simulation takes place at a level of abstract machine instructions. Compared to traditional instruction-level simulation, this approach typically results in a higher simulation performance at the cost of a small loss of accuracy.

A consequence of simulating at the level of abstract machine instructions is that the behaviour of applications must explicitly be modelled rather than simply interpreting the machine instructions from the programs. In this paper, we therefore focus on the workload modelling techniques used within Mermaid. Furthermore, we present validation and performance results suggesting that our simulation methodology provides both accuracy and good performance.

The next section gives a brief overview of the Mermaid simulation environment. Section 3 describes the modelling of application behaviour in order to drive the architectural simulators. In Section 4, some validation results are presented. These results were obtained by comparing the performance behaviour of an existing multicomputer to the simulation estimates of our models. Section 5 discusses the simulation performance of Mermaid. Finally, Section 6 presents the summary and mentions possible future work.

2 The Mermaid simulation environment

The multi-layered simulation environment of Mermaid is illustrated in Figure 1. The lowest level, referred to as the architecture level, contains the architecture simulation models. These models are implemented in a highly modular fashion using the object-oriented simulation language Pearl [13], which allows flexible evaluation by means of parameterization. The simulators are driven by abstract instructions, called *operations*, representing processor activity, memory I/O and message-passing communication. Several examples of operations are shown in Table 1.

Simulating at the level of operations has several consequences. As the operations abstract from the processor's instruction sets, the simulators do not have to be adapted each time a processor with a different instruction set is simulated. We simulated, for example, both multicomputers based on Transputers and on PowerPC processors with only little remodelling effort. Furthermore, simulating operations rather than interpreting real machine instructions yields higher simulation performance at the cost of a small loss of accuracy. On the other hand, low-level simulation of, for instance, the processor pipelines is not possible due to the lack of register specifications in the operations.

To correctly model issues like synchronization and load-balancing, each processor within the multicomputer model receives its own trace of operations. In this scheme, the validity of the multiprocessor operation-traces is guaranteed by the fact that the simulator controls the trace generation in an execution-driven manner [15]. In other words, the trace generator is interleaved with the simulator and the traces are generated on-the-fly.

As shown in Figure 1, the simulation environment also provides a suite of tools in order to visualize and analyze the simulation output. Visualization of data can be performed at runtime [10] or post-mortem. Moreover, a tool called RAPID [16] facilitates the statistical analysis of the simulation output. It contains a simple specification language in which the user specifies what statistical methods should be used on which parts of the simulation data. According to this specification, RAPID generates an executable performing the specified types of analysis.

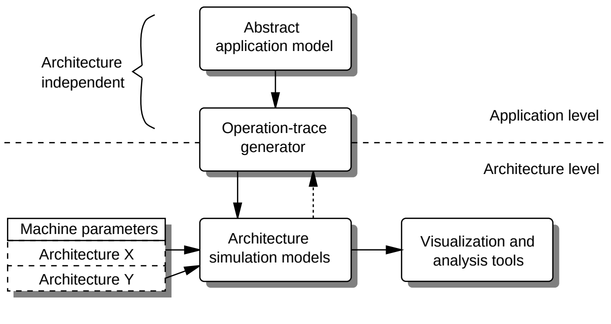

Architecture independent
Abstract application model
Operation-trace generator
Application level
Architecture level
Machine parameters
Architecture X
Architecture Y
Architecture simulation models
Visualization and analysis tools

Figure 1. Simulation environment.

Operation	Meaning
ifetch(a)	Instruction fetch from address a
add(int)	Add two integers residing in registers
ld(w,a)	Load a word from address a
send(n,d)	Send a n-byte message to node d

Table 1. Examples of ***operations*****.**

To provide the architectural simulators with operation-traces representing actual application behaviour, an abstract application model and a trace generator reside on top of the architecture level (see Figure 1). A more comprehensive description of these parts of the simulation environment is presented further on in this paper.

2.1 A hybrid architecture model

Many applications, and especially scientific applications, running on multicomputer platforms contain coarse-grained computations alternated with periods of communication. Because these computation and communication phases typically are distinct, we decided to split the simulation of multicomputers into two different models: a single-node[1] *computational model* and a multi-node *communication model*. Each model operates at a different level of detail, and thus defines its own set of operations. The computational model simulates the application's computational behaviour. It models the incoming computational operations at a level of abstract machine instructions, as was previously discussed. Communication operations are not simulated by this model, but are directly forwarded to the communication model. Subsequently, the communication model accounts for the application's message-passing communication behaviour. To address the issues of synchronization and load-balancing properly, it models the computational delays found in between communication requests at the task level. A parallel workload for this model therefore resembles a graph containing computational tasks and communication operations. The computational tasks are derived from the computational model, which constructs them by measuring the simulated time between two consecutive communication operations.

This approach results in a *hybrid model*, which allows for simulation at different abstraction levels. If accuracy is required, then the complete hybrid model can be used. In this case, the single-node computational model is replicated for each MIMD node taking part in the simulation. Each instance of the single-node model is then assigned to a node within the communication model in order to feed it with the computational tasks and the communication operations. This is illustrated in Figure 2.

[1] It may contain multiple processors sharing a single memory, though.

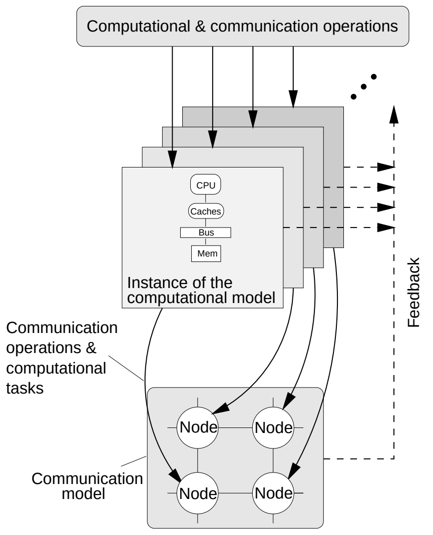

Computational & communication operations
CPU
Caches
Bus
Mem
Instance of the computational model
Feedback
Communication operations & computational tasks
Communication model
Node
Node
Node
Node

Figure 2. Detailed simulation of a multicomputer using both the computational and the communication models.

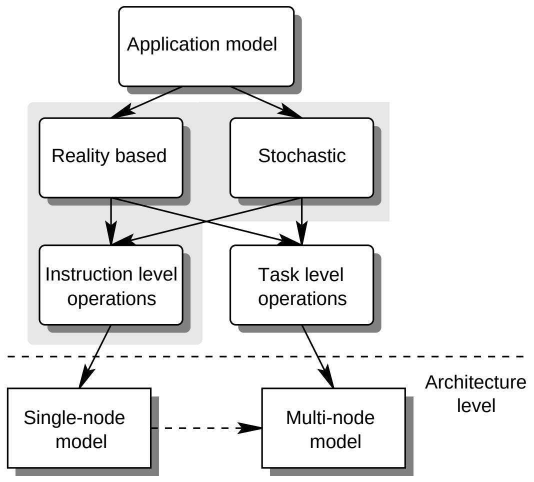


Figure 3. Application modelling within Mermaid. The large shaded area indicates the modelling path currently supported.

However, if there is only the need for fast prototyping, then just using the communication model might be sufficient. In that case, the task-level operation-traces must be directly produced by the trace generator. To some extent, this is similar to what happens in the commonly applied direct execution simulation technique: the performance of the code segments in between two communication events (forming the computational tasks) has to be estimated at compile time.

2.2 Shared memory or hybrid architectures

The multi-node communication model, with its message passing, intrinsically suggests that the system under investigation should be a distributed memory architecture. But, by only using the computational model and configuring it with multiple processors, a shared memory multiprocessor can be simulated. A disadvantage of this approach however, is that simulation can only be performed at the level of computational operations: the highest level of detail.

Hybrid architectures, such as the one proposed in the MIT StarT-ng project [4], can be modelled by both defining multiple processors on a node and using the communication model to interconnect the clusters of shared memory multiprocessors in a message-passing network.

3 Workload modelling

Because the Mermaid architecture models simulate at the level of operations and do not interpret real machine instructions, application behaviour must explicitly be modelled. As application behaviour is best described in terms unrelated to any architecture specifics, we strive to model application workloads in a manner which is independent of both the host computer (on which the simulation runs) and the target architecture (which is being evaluated). As a consequence, Figure 1 shows a clear separation between the application level, at which the application model and part of the trace generator reside, and the architecture level.

Like it is the case in architecture modelling, modelling a workload can also be done at various abstraction levels and with different degrees of accuracy. Figure 3 illustrates the workload modelling framework of Mermaid. A workload is either based on a real application or it is synthetic and produced by some stochastic process. Furthermore, both real and synthetic workloads can model computation either at the level of abstract machine instructions or at the level of tasks. Computation at the instruction level is typically simulated by the single-node (computational) model, whereas task level operations are simulated by the multi-node (communication) model.

Currently, Mermaid supports only the generation of abstract instruction-level operations, as depicted by the large shaded area in Figure 3. We will therefore limit our discussion to this area of workload modelling.

3.1 Reality based workload modelling

To capture a realistic view of application behaviour, it is required to trace a real program. Traditionally, the tracing of applications is mainly performed by augmenting either the assembler code [11, 19] or the executable itself [12]. However, these tracing techniques may introduce dependencies on either the host or the target architectures. To guarantee architecture independence at the application level, we decided to directly augment C source code. So, in this case, the

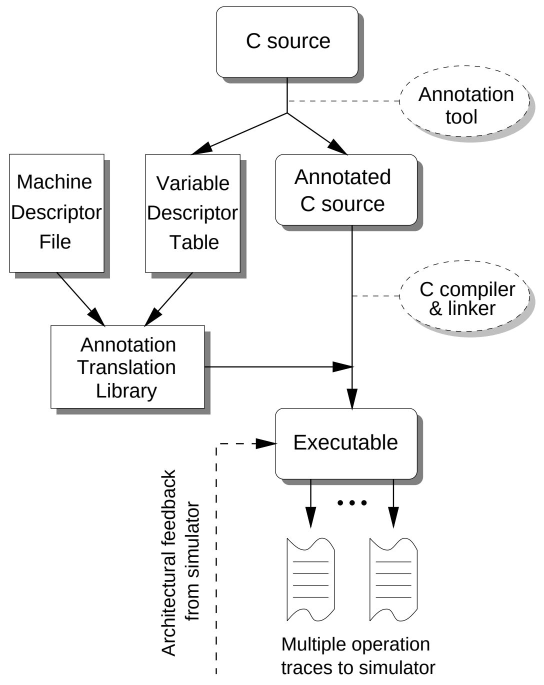

C source
Annotation tool
Machine Descriptor File
Variable Descriptor Table
Annotated C source
C compiler & linker
Annotation Translation Library
Executable
Architectural feedback from simulator
Multiple operation traces to simulator

Figure 4. Generation of operations by augmenting real applications.

abstract application model consists of programs that have been instrumented with annotations following the program's memory, computational and communication behaviour. As the augmented source code is architecture independent, instrumentation has to take place only once, after which the application model can be used to evaluate any architecture.

Figure 4 gives an overview of how Mermaid generates operations by augmenting real applications. A special tool instruments the C source code and constructs a so-called *variable descriptor table*. This table determines whether a variable is global, local, or a function argument. It also contains information on the addresses of variables, whether they are placed in a register or not and the types of the variables.

The Annotation Translation Library (ATL) is the trace generating core, which is linked to the annotated application. As the annotations are simply calls to the ATL, the annotations are dynamically translated into the appropriate trace of operations when the obtained executable is executed. For this purpose, the ATL uses the variable descriptor table and a *machine descriptor table*. The latter contains architecture dependent information necessary for the trace generation (e.g. instruction size, number of registers, etc.). When, for example, an annotation indicates that a variable should be loaded, the generator uses the information from both tables to translate the annotation into the appropriate instruction fetch and memory operations. The ATL can thus be regarded as a kind of generic compiler. It performs the translation of annotations according to the runtime model

```
double a, x[N], y[N];

void foo(...) {

   int i;

   [...]

   for (i = 0; i < N; i++) {
      a += x[i] * y[i];
   }

   [...]
}
```

Figure 5. A double precision inner-product (ddot).

```
double a, x[N], y[N];

void foo(...) {

   int i;

   set_function(foo, (void *)foo);

   [...]

   for (assignIc(local(i), 0), i = 0;
        setPC(c1),
        arithIc(REG, N, local(i), CMP), i < N;
        arithIc(local(i), 1, local(i), ADD), i++) {

      a += x[i] * y[i];

      arithDF(REG, array(&x, local(i), &x[i]),
                   array(&y, local(i), &y[i]), MUL);
      arithDF(&a, REG, &a, ADD);

      setPC(l1);
   }
   setPC(l1);


   [...]

   ret_function();
}
```

Figure 6. The annotated version of ddot.

and addressing capabilities of the target processor. In this approach, only the ATL is architecture dependent. Therefore, only the ATL may need to be updated in order to generate operation-traces for a new (possibly non-existing) architecture.

To illustrate the process of instrumentation, consider Figures 5 and 6. The first figure shows a code fragment calculating a double precision inner-product (ddot), whereas the latter one shows the annotated version of the same code fragment. The annotations *set_function*, *ret_function* and *setPC* describe the control flow behaviour of the program. For example, the *setPC(label)* annotation is some sort of basic block indicator: the first time a *setPC* is triggered, it saves the current (pseudo) program counter together with its label. Succeeding calls to a *setPC* with an identical label will reset the program counter to the saved value. Subsequently, the $assign_{type}$ and $arith_{type}$ annotations describe the computational behaviour of the application. In these annotations, the type must be specified on which is operated. For instance, *Ic* means “Integer constant” and *DF* means “Double Float”. Handling the references to variables is done by annotations such as *local* and *array*. If required, these annotations instruct the ATL to model the fetching of variables according to the variable descriptor table.

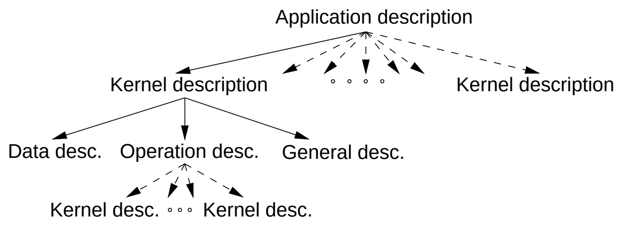
To generate the multiprocessor traces, the ATL models concurrent execution by means of threads. For this purpose, the tool which performs the instrumentation of programs also provides some support for converting (single-threaded) target applications into multi-threaded programs. For SPMD-like applications, this conversion is fully automated. For other classes of applications, manual tuning of the obtained threaded code is still necessary.

3.1.1 Modelling communication

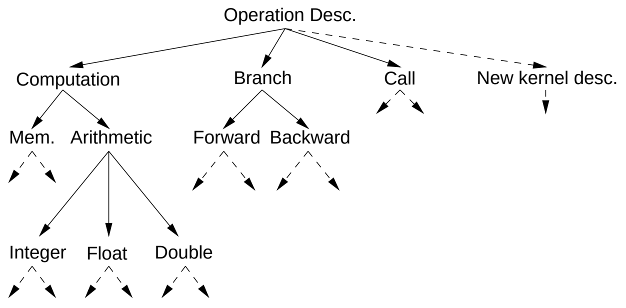
The annotations describing communication behaviour directly map onto operations, such as the *send* listed in Table 1. As the associated parameters of these operations are based on the platform’s physical topology, the modelling of communication still reflects some of the underlying hardware characteristics. Ideally, such architectural details are not visible at the application level. One way to achieve this, is by defining the communication annotations such that they are based on a virtual shared memory model. Consequently, all explicit communication at the application level is removed and substituted by memory references. This implies that the simulation environment issues the communication requests rather than the applications. Naturally, the generation of these communication requests is dependent on the distribution of data over the different processors. By specifying how data is distributed within the simulation environment, different data distributions can easily be evaluated without the need to change the applications. Although this virtual shared memory model has not yet been implemented, it is considered as future work.

3.2 Stochastic workload modelling

Besides the tracing of real applications, we are currently also investigating techniques to generate operation-traces from stochastic application descriptions. In such a scheme, the abstract application model consists of descriptions describing the application behaviour using probabilities. Evidently, this technique represents the behaviour of (a class of) applications only with modest accuracy. It allows, however, for more flexibility than the tracing of real programs. For example, rapidly adjusting the application behaviour is fairly easy using this technique, which can be useful when fast-prototyping new architectures.



(a)



(b)

Figure 7. Structure of a SEA description.

The probabilistic descriptions are written in the language SEA (Stochastically Expressing Applications), which was especially designed for this purpose. A tool, called the Stochastic Trace Generator (STG), interprets the SEA descriptions and subsequently generates operation-traces from them. Figure 7a shows the general structure of a SEA description. From a high-level point of view, an application description is constructed of a sequence of one or more *kernel* descriptions. These kernel descriptions consist of three parts: a data part, an operation part and a general part. The data section describes what kind of variables can be manipulated within a kernel. The operation section specifies the types of operations taking place within a kernel. Since this section may also contain sub-kernel descriptions, the complete application model is represented by a tree of kernel descriptions. Finally, the general section describes issues like the number of operations to be generated within a kernel, addresses of segments, etc.

Figure 7b shows (a part of) the operation description hierarchy. Each arc within the hierarchy is weighted with a certain probability. When an operation is generated, the hierarchy is traversed top-down according to the probabilities of the arcs until a leaf containing a certain operation has been reached. In order to specify the type of data on which the operation is performed (e.g. array, local variable, global variable, etc.), the data section of the kernel description is traversed in a similar manner.

Using kernel descriptions as a building block allows the modelling of an application at different abstraction levels. Basically, the behaviour of a whole program can be described with one kernel description only. In this case, the

```
kdescription[1024, 0x100] {
   general {
      instructions = 8;
   }
   data {
      dfp_data {
         variable[50%] {
            number = 1;
            global = 100%;
            auto   = 0%;
         }
         array[50%] {
            number  = 2;
            av_size = 1024;
            global  = 100%;
         }
      }
      non_fp_data {
         variable[100%] {
            number = 1;
            global = 0%;
            auto   = 100%;
         }
         array[0%] { }
      }
   }
   computation[100%] {
      mem_expr[25%] {
         sfl_point = 0%;
         dfl_point = 50%;
         integer   = 50%;
         constant  = 0%;
      }
      arith_expr[75%] {
         sfl_point = 0%;
         dfl_point = 33%;
         integer   = 67%;
         constant  = 33%;
      }
      fp_expr {
         plain_fp[100%] {
            fadd = 50%; fmul = 50%;
            fdiv = 0%;  fsub = 0%;
         }
         intrinsic_fp[0%] { }
      }
      integer_expr {
         non_bit_op[100%] {
            add = 75%; sub = 25%;
            mul = 0%;  div = 0%;
         }
         bit_op[0%] { }
      }
   }
   calls[0%]         { }
   branches[0%]      { }
   sub_kdesc[0%]     { }
   communication[0%] { }
}
```

Figure 8. SEA description of ddot.

application is described at a high abstraction level, which is flexible but might not be very accurate. Alternatively, multiple kernel descriptions can be used to describe the behaviour of certain parts of a program. This approach may require somewhat more modelling effort but it represents the application behaviour with higher accuracy.

To illustrate the stochastic modelling, consider Figure 8. It shows a SEA description for the *ddot* code fragment of the previous section with N=1024. For the sake of clarity, we omitted a few lines of SEA code which were not relevant to this example. The (single) kernel is iterated 1024 times, generating 8 operations per iteration starting at instruction address 0x100. Of course, SEA would also allow the non-probabilistic values, such as the number of instructions, to be random and to behave according to some distribution.

Currently, the SEA descriptions are produced by hand. Naturally, this will eventually be automated by a framework in which real programs are profiled in order to generate the descriptions. Moreover, stochastic trace generation for multiprocessors is not yet supported. We are still investigating how, for instance, communication can be modelled best within the SEA paradigm. A promising approach, however, is proposed in [5]. As the stochastic modelling of applications is still subject to many (fundamental) changes, we did not yet perform validation experiments. For this reason, the next section will purely focus on the validation of simulations driven by traces from real applications.

4 Validation

To validate our simulation methodology, we instrumented several applications and simulated them for an existing distributed memory multicomputer. This machine, a Parsytec GCel, consists of Inmos T805 transputers connected in a 2D grid network. Validation is performed by comparing the simulation results of several benchmarks with the results of real execution. Initially, the computational and communication models were validated separately. Table 2 shows the overall results of these experiments. It gives the average error, the standard deviation σ of the average error and the worst case error of simulation compared to execution. The workloads for the computational model consisted of a set of well-known numerical kernel functions, including *ddot* (double precision inner-product), *daxpy* (double precision vector update) and some of the Lawrence Livermore kernels.

The communication workloads, which only model message passing and its consequent (computational) delays, can be divided into three categories. The first category, that

Average error (%)	σ	Worst case (%)	Workload
1.5	1.3	3.8	Computational loads
1.5	1.3	5.0	Light communication loads
2.9	3.3	10.9	Uniform congesting communication loads
4.8	9.5	29.7	Non-uniform congesting communication loads

Table 2. Validation of the separate computational and communication models.

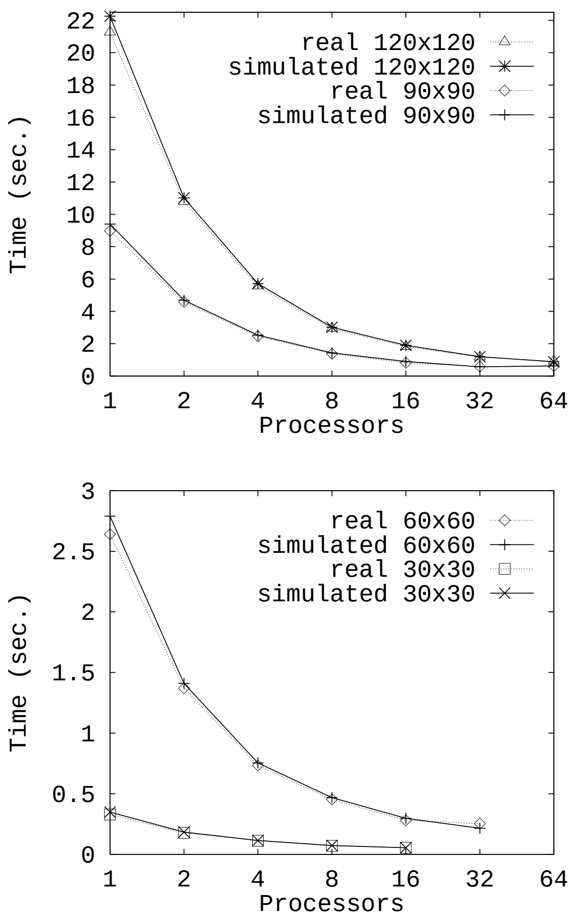


Figure 9. A parallel matrix multiplication.

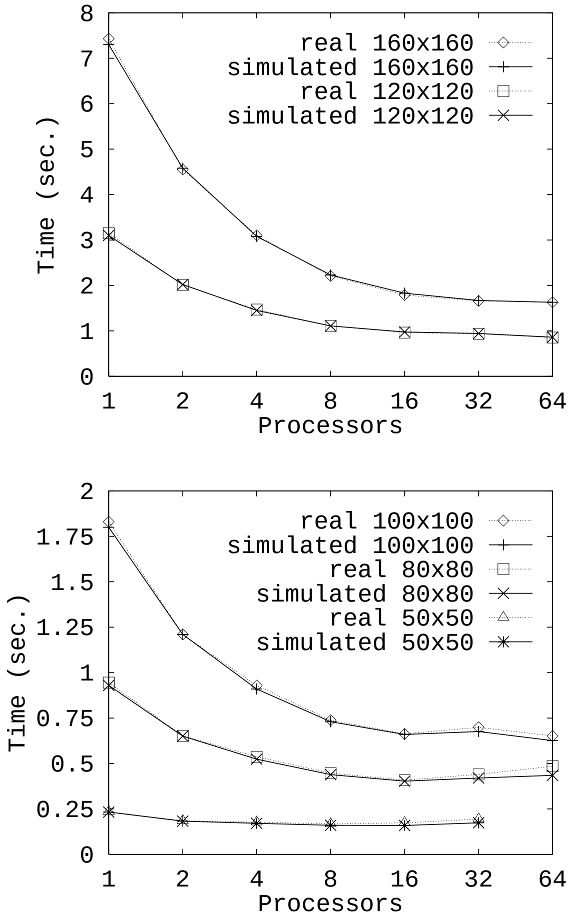


Figure 10. A parallel Gaussian solver.

of light communication loads, uses benchmarks performing message roundtrips. The two remaining types of communication workloads are for so-called "stress-testing" purposes, as they heavily congest the network. They can again be subdivided into uniform and non-uniform loads. The first type of load uniformly distributes communication over the network, while the latter type only stresses some small regions within the network, causing hot-spots to appear.

For computation and light communication loads, the average error does not exceed 1.5%. More importantly, the standard deviation and the worst case error indicate that the performance estimates for these loads are quite accurate in general. The errors of the stress-testing communication benchmarks are higher, especially in the case of the non-uniform loads. This can be explained by the fact that these benchmarks produce workloads exhibiting high contention for network resources, which tends to amplify any existing modelling errors. While a worst case error of nearly 30% was measured for these extreme loads, the standard deviations suggest that the accuracy in general is tolerable.

Validation of the integral system, consisting of both the computational and communication models, has been performed by two parallel benchmarks. The first is a parallel matrix multiplication and the second one concurrently solves a system of linear equations by Gaussian elimination.

Figures 9 and 10 depict the execution times and estimated (simulation) times of both benchmarks for a range of different matrix sizes. With a calculated average error of 1.8% and a standard deviation of 2.6 for the Gaussian solver, high accuracy is obtained. For the simulation of the matrix multiplication, we measured a somewhat larger average error of 3.9% and a standard deviation of 3.2. The worst case errors equal to 15% and 10% for the matrix multiplication and the Gaussian solver respectively.

An important observation is that simulation closely follows the execution trend. This is even the case for the small irregularity in the graph of the Gaussian solver at a problem size of 100×100 using 32 processors. Closer examination showed that this irregularity is caused by a load imbalance.

5 Simulation performance

To give an indication of the simulation performance of Mermaid, we measured the *slowdown* for several simulations. The slowdown is defined by the number of cycles it takes for the host computer to simulate one cycle of the target architecture. An exact value for the slowdown cannot be given since it depends on the type of application and architecture that are being modelled. Therefore, a *typical* value will be used. To determine the slowdown per simulated processor, we examined both the transputer simulator of the previous section and a simulation model of a Motorola PowerPC 604 using two levels of cache.

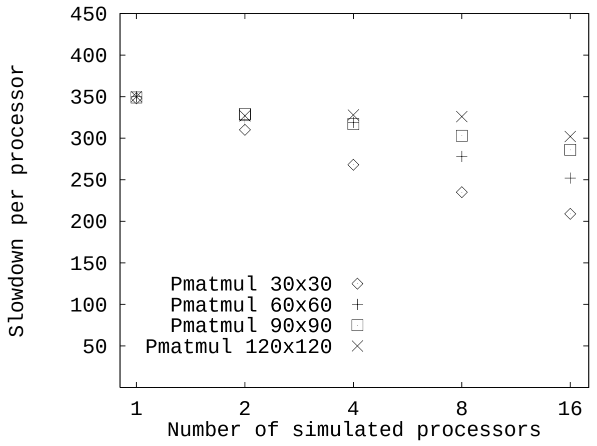
For a mix of application loads, we measured a typical slowdown of about 60 to 750 per processor. So an Ultra Sparc processor running at 143Mhz roughly simulates between 200,000 and 2,400,000 cycles per second. This performance makes Mermaid fairly competitive with, for instance, many direct execution simulators, which typically obtain a slowdown of between 2 and a few hundred [2, 9, 3].

We believe, however, that the simulation efficiency can still be enhanced, making Mermaid even more competitive performance-wise. For instance, the Pearl simulation language, in which the architecture models are written, emphasizes the modularity and easy implementation of architecture models. It generates only moderately efficient code. The choice of another modelling language might therefore improve the simulation performance.

The slowdown of the simulation of an entire multicomputer is a function of the slowdown of a single processor and the overhead caused by simulating the interconnection network. Typically, this slowdown is calculated by multiplying the per-processor slowdown with the number of simulated processors. Thus, a simulated multicomputer containing 64 nodes would then have a slowdown of somewhere between 4,000 and 50,000.

If fast prototyping of a multicomputer is the primary goal, then the communication model can be used directly. The slowdown of this type of simulation depends on the amount of computation and communication present within the application. Computation can be simulated extremely fast since it is modelled at the level of tasks, whereas communication is simulated in more detail and is thus less efficient. Our measurements indicate that simulation at this level of abstraction results in a typical slowdown of between 0.5 and 4 per processor. This means that an entire multicomputer can be simulated with only a minor slowdown.

Figure 11 shows the slowdown per processor as a function of the number of simulated processors for the matrix multiplication and the Gaussian solver simulations of the previous section. Both graphs more or less suggest that the slowdown decreases when simulating more processors. This is caused by the fact that simulation of communication is quite efficient in Mermaid, whereas computation is simulated more low-level and therefore more expensive. In these particular experiments, the simulation of communication is even faster than the communication within the real machine. Therefore, the slowdown decreases as the number of nodes increases.



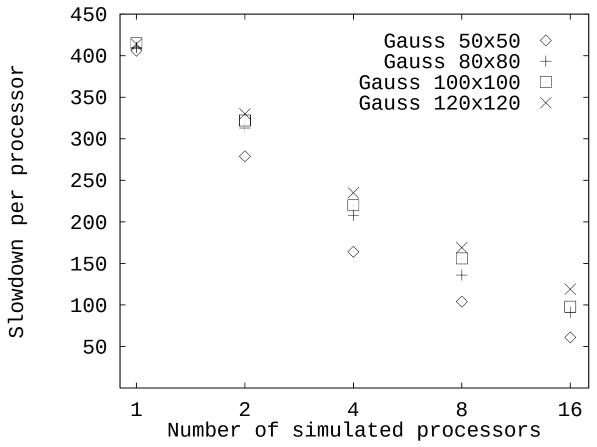


Figure 11. Slowdown of the parallel matrix multiplication (top) and the parallel Gaussian solver (bottom).

Another important aspect of multicomputer simulation is the memory usage. Simulators that consume a lot of memory may encounter problems when scaling the simulation to a large number of nodes. Since instructions are not interpreted in Mermaid, it is not necessary to store large quantities of state information during simulation runs. For example, the contents of the memory does not have to be modelled and simulated caches only need to hold addresses (tags), not data. As a consequence, the simulation of parallel platforms is only constrained by the memory consumption of the (threaded) trace-generating applications.

6 Summary

In this paper, we provided an overview of the Mermaid multicomputer simulation environment, which allows the performance evaluation of a wide range of architectural design options. Mermaid features the ability to simulate at various abstraction levels, each with its own accuracy and performance. It therefore makes both accurate evaluation and fast prototyping of multicomputers feasible. At the lowest level of abstraction, simulation takes place at the level of abstract machine instructions rather than interpreting real machine instructions. As a consequence, the workloads that are driving the architectural simulators should be modelled explicitly.

The workload modelling within Mermaid can be divided into two approaches. The first approach is reality based and applies the tracing of real applications. In this technique, the augmentation of the programs takes place directly at the high-level language level. This guarantees the architecture independence of the application model we strive for. In the second approach, application behaviour is modelled using probabilistic descriptions. Such a scheme allows for more flexibility than the tracing of real programs but its accuracy is only modest. When fast-prototyping new architectures, however, this may be a useful technique.

The Mermaid simulation methodology has been validated by comparing simulation estimates with the performance results of an actual machine. These experiments demonstrate that, despite of the reasonably high abstraction level at which is modelled, good accuracy can be obtained. Typically, errors were measured that do not exceed 10%.

Besides Mermaid's good accuracy, its simulators are fairly efficient as well. For the experiments we performed, we measured a typical slowdown of about 60 to 750 per simulated processor. This performance makes Mermaid quite competitive with many other efficient architecture simulators.

This paper has described ongoing research. Clearly, much work still has to be done in order to provide a broader support for trace generation. This includes techniques for producing realistic, synthetic multiprocessor traces and methods for validating these traces. Moreover, future application modelling will eventually be based on a (virtually) shared memory model. This allows for a more flexible evaluation of architecture and application performance under different data distributions.

Acknowledgements

We would like to thank Hugh McEvoy and Marcel Beemster for their comments on an earlier version of this paper.

References

[1] R. C. Bedichek. Talisman: Fast and accurate multicomputer simulation. In *Proceedings of the 1995 ACM SIGMETRICS Conference*, pages 14–24, May 1995.

[2] B. Boothe. Fast accurate simulation of large shared memory multiprocessors. Tech. Rep. CSD 92/682, Comp. Science Div. (EECS), Univ. of California at Berkeley, June 1993.

[3] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Tech. Rep. MIT/LCS/TR-516, MIT Laboratory for Computer Science, Sept. 1991.

[4] D. Chiou, B. S. Ang, R. Greiner, Arvind, J. C. Hoe, M. J. Beckerle, J. E. Hicks, and A. Boughton. StarT-NG: Delivering seamless parallel computing. In *Proceedings of the First International EURO-PAR Conference*, pages 101–116, Aug. 1995.

[5] S. Chodnekar, V. Srinivasan, A. Vaidya, A. Sivasubramaniam, and C. Das. Towards a communication characterization methodology for parallel applications. In *Proc. of the 3rd International Symposium on High Performance Computer Architecture (HPCA)*, pages 310–319, Feb. 1997.

[6] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proc. of the 1994 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 128–137, May 1994.

[7] R. G. Covington, S. Dwarkadas, J. R. Jump, J. B. Sinclair, and S. Madala. The efficient simulation of parallel computer systems. *Int. Journal in Comp. Simulation*, 1:31–58, 1991.

[8] H. Davis, S. R. Goldschmidt, and J. Hennessy. Multiprocessor simulation and tracing using Tango. In *Proc. of the 1993 Int. Conf. in Parallel Processing*, pages 99–107, Aug. 1991.

[9] A. G. P. Joubert. SPAM: A multiprocessor execution driven simulation kernel. Tech. Rep. 708, IRISA research laboratory, Mar. 1993.

[10] H. C. Kok, A. D. Pimentel, and L. O. Hertzberger. Runtime visualization of computer architecture simulations. To appear in *Proc. of the Workshop on Performance Analysis and its Impact on Design (in conjunction with the 24th Int. Symp. on Computer Architecture)*, June 1997.

[11] J. R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software Practice & Experience*, 20(12):1241–1258, Dec. 1990.

[12] J. R. Larus and T. Ball. Rewriting executable files to measure program behaviour. *Software Practice & Experience*, 24(2):197–218, Feb. 1994.

[13] H. L. Muller. *Simulating computer architectures*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, Feb. 1993.

[14] A. D. Pimentel and L. O. Hertzberger. Evaluation of a mesh of clos wormhole network. In *Proc. of the 3rd International Conference on High Performance Computing*, pages 158–164. IEEE Computer Society Press, Dec. 1996.

[15] A. D. Pimentel and L. O. Hertzberger. An architecture workbench for multicomputers. In *Proc. of the 11th International Parallel Processing Symposium*, pages 94–99. IEEE Computer Society Press, April 1997.

[16] A. D. Pimentel and L. O. Hertzberger. RAPID: RAPid Interpretation of Data. Tech. Rep. CS-97-01, Dept. of Comp. Sys, Univ. of Amsterdam, Jan. 1997.

[17] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proc. of the 1993 ACM SIGMETRICS Conference*, pages 48–60, May 1993.

[18] M. Rosenblum, A. S. Herrod, and A. Gupta. Complete computer system simulation: The simOS approach. *IEEE Parallel & Distributed Technology*, 03(04):34, 1995.

[19] C. B. Stunkel, B. Janssens, and W. K. Fuchs. Address tracing for parallel machines. *IEEE Computer*, 24(1):31–38, Jan. 1991.