Technical Report
MECSE-12-2007

Petri Net modelling of computer parallelism

T. Ramdas, G. Egan and D. Abramson

Petri Net modelling of computer parallelism

Tirath Ramdas, Gregory Egan and David Abramson

Abstract— **Petri Net modelling may be applied to parallel computer systems. For the purpose of projecting the performance of a model, there is flexibility: depending on the availability of computational resources and the desired level of precision, one may choose between a rigorous state-space based estimation or one may fall back upon less precise discrete-event simulation approaches. In this paper a simple Petri Net model of a computer is evolved, highlighting how concepts such as resource contention and latency hiding may be incorporated.**

I. Introduction

Petri Nets are a formalism based on Markov processes that have found applicability in disciplines as diverse as interactive computer systems to business and industrial process modelling. Within the computer architecture domain Petri Net modelling has been applied to analysis of network applications on the IXP network processor [1], analysis of superscalar processors [2][3] and many others. One specific task which Petri Nets are adept at handling is performance modelling.

The goal of this paper is not to present any new/novel computer architecture findings, nor is it to provide a tutorial on Petri Net modelling. Our goal is merely to demonstrate that simple Petri Nets are capable of capturing pertinent structural information of computer systems, and are therefore capable of modelling the performance of computer systems. Furthermore, we favour simple models that exhibit simple behaviour so that the correctness of their behaviour is obvious. An ancillary contribution of this paper is that some basic ideas are presented on how to model common computational concepts such as spatial parallelism (i.e. "replication"), pipelining, shared busses, etc.

A good "vade mecum" style treatment of Petri Nets is provided in [4]. A comprehensive coverage of the subject from the perspective of manufacturing systems is presented in [5]. Murata presents a comprehensive overview of Petri Net capabilities in [6]. We will present only a quick "refresher" style overview of Petri Nets. However, we have attempted to present the models in the latter sections with adequate explanation; we hope to leverage one of the greatest strengths of Petri Nets, i.e. that it is quite an intuitive graphical expression, and therefore requires little explanation.

We agree with the opinion of Govind and Govindarajan [1] that models of computer systems ought to take into account not only the architectural characteristics and structure of the hardware, but also the specific characteristics and requirements of the program to be executed. We believe that Petri Net modelling is especially applicable to application specific architecture studies.

This paper starts with an overview of Petri Net formalism. This overview is accompanied by some introductory examples, which are accompanied by state-space analysis. We then compare Petri Nets to other techniques for modelling computer systems. Subsequently we present increasingly complex Petri Net models of computer systems: starting with a basic sequential model, followed by an embarrassingly parallel model, followed by a parallel model with bus contention, and finally a simple two-stage pipeline.

II. Petri Net Overview

Petri Nets are a graphical formalism for expression of discrete event systems, which may exhibit some or all of the following characteristics [7]:

- *Event-driven*, i.e. changes in system state are triggered by event occurrences.
- May be *asynchronous*.
- There may be some sequential relationship (or *precedence relations*) between a subset of events.
- There may be *concurrency* between a subset of events.
- *Conflict* may occur (e.g. with shared access to a common resource).
- The system may exhibit *mutual exclusion* (e.g. a subset of events cannot occur at the same time).
- There may be *non-determinism* in the sequence of event occurrences.
- The system may suffer *deadlock* states.

Some of the advantages offered by Petri Net modelling of discrete event systems include [7]:

- Ease of use; the graphical representation allows human eyes to quickly observe system dependencies and to focus on regions of interest in the model.
- It is possible to synthesise supervisory control code for the target system.
- Ability to perform state-space analysis in order to detect system deadlock and un-boundedness.
- Performance analysis (e.g. throughput vs. number of machines) may be conducted through state-space analysis or simulation.

A Petri Net consists of places, tokens, transitions, and arcs, which have the following physical significance [7]:

- A *place* may represent resources (e.g. a source data structure) or an operation status (e.g. when the machine is idle).
- A *token* in a place is interpreted based on the interpretation of the place. When the place represent a resource, then the token within the place represents the availability of that resource. When the place is an operation status,

T. Ramdas {tirath@int19h.com} is with the Center for Telecommunications and Information Engineering, Monash University.

G. Egan {greg.egan@eng.monash.edu.au} is with the Center for Telecommunications and Information Engineering, Monash University.

D. Abramson {david.abramson@infotech.monash.edu.au} is with the Center for Distributed Systems and Software Engineering, Monash University.

then the token within the place indicates that the system is currently in that state and performing that activity.

- A *transition* represents events; the firing of a transition moves tokens from the source place(s) to the destination place(s). In timed Petri Net models, transitions may have associated delays/rates. Specifically regarding timed Petri Nets, Montano et al. [8] describe three roles that transitions may play:
  - Transitions may represent operations; when a transition is enabled, it commences the operation, and when it fires the operation is terminated. Time in this context refers to the amount of time needed to perform the operation.
  - Transitions may represent events; when a transition fires, the corresponding event is said to have occurred. Time in this context refers to the time between event occurrences.
  - Transitions may be used to express synchronisation and control tasks, and exist only to facilitate state-changes in the model – they therefore lack a tangible interpretation in the context of the physical system. Such transitions have no time associated with them.
- An *arc* is unidirectional, and links a place to a transition or a transition to a place. Both transitions and places may have more than one terminating or originating arc.

The basic methodology for constructing a Petri Net model of a system may be summarised as follows [7]:

1) Operations and resources required for the system to function are identified.
2) Operations are ordered according to precedence relations.
3) A place is created to represent each operation.
4) Transitions are placed before and after each place indicating the start and end of the operation. A single transition may indicate the end of one operation and the start of another.
5) A place is created for each resource required to perform the various operations.
6) The initial marking of the system (i.e. the initial allocation of tokens) is specified.

III. Applied Petri Net Modelling

The top-level modelling space is rich in tools and methodologies. In this section we attempt to argue that Petri Nets are a good fit for modelling computer systems. Much of our reasoning is influenced by Zhou and DiCesare [7], Desrochers and Al-Jaar [5], Cassandras and Lafortune [9], Donatelli et al. [10] and Montano et al. [8].

We are particularly interested in performance analysis of the various computer architecture models. We will begin this section with an example analysis of a simple model. This presentation assumes basic knowledge of Petri Net modelling, including an understanding of the basic concepts of *places* and *transitions*. The goal of this section is to solidify introductory-level Petri Net knowledge, and also to define certain terms that we will use for the rest of the paper. Experienced Petri Net practitioners can safely skim through this section.

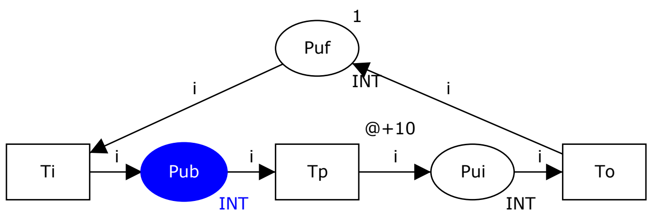


Fig. 1. Petri Net model of an reliable machine.

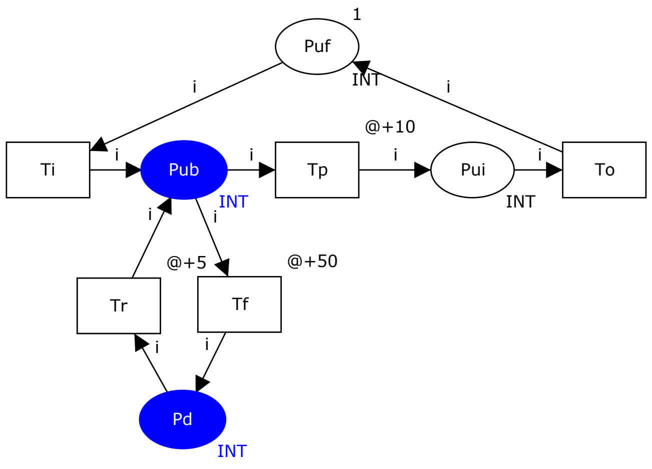


Fig. 2. Petri Net model of an unreliable machine.

Once an illuminating example has been presented, we will provide a brief (and not at all exhaustive) comparison of Petri Nets vs other modelling approaches, with the intention of providing some insight into the relative merits and capabilities of Petri Net modelling.

A. Analysis Example

In this section, we present an example analysis of a simple model. Consider the Petri Net model presented in Fig. 1 (for now disregard the fact that some places are shaded and some are not). This model corresponds to what we will call a *reliable machine*. When the machine is "free" (i.e. when place *Puf* is marked – this is the initial state of the machine, where there is 1 token in place *Puf*, and no tokens in all other places), transition *Ti* fires, which indicates that an unprocessed part enters the machine input. In terms of the Petri Net model, the firing of this transition moves the token in place *Puf* to place *Pub*. Now the part undergoes processing (i.e. place *Pub* is marked – the machine is "busy"). Once the part has been processed, transition *Tp* fires, and the token in place *Pub* is moved to place *Pui*, indicating that the machine is "idle". Next transition *To* fires, which signifies that a processed part is output from the machine, and the token in place *Pui* is moved to place *Puf*, indicating that the machine is once again "free" and a new unprocessed part may be input into the machine – i.e. transition *Ti* is enabled.

The reliable machine model is not very realistic, and not very interesting. We shall instead consider analysis of an "unreliable machine".

Identifier	Interpretation
Puf	Machine up and free
Pub	Machine up and busy
Pui	Machine up and idle
Pd	Machine down
Ti	Part enters machine
Tp	Part is processed
To	Part leaves machine
Tf	Machine breaks down
Tr	Machine is repaired

TABLE I
PLACES AND TRANSITIONS OF FIG. 2

*T*ransition	Rate	Interpretation
Tp	t_p	Peak part processing rate when up and busy (parts/hour)
Tr	t_r	Machine repair rate (machines/hour)
Tf	t_f	Machine failure rate (failures/hour)

TABLE II
TRANSITION FIRING RATES OF FIG. 2

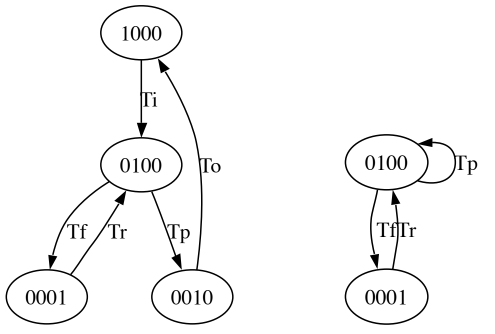
Fig. 2 depicts the Petri Net model of an *unreliable machine* (once again, for now disregard that some places are shaded and some are not). This machine occasionally breaks down and needs to be repaired. Of course, while it is broken (and being repaired) it is not processing anything and therefore is not productive. The places and transitions of this model have the interpretations listed in Table I.

Note that transitions *Tp*, *Tf*, and *Tr* have *firing rates* associated with them. In contrast, transitions *Ti* and *To* do not – *Ti* and *To* are called *immediate transitions*. Some application-specific and intuition-reliant thought must go into the design of such models. With this example, the reasoning behind the omission of firing rates for *To* and *Ti* is that the time it takes to load an unprocessed part into the machine and unload a processed part from the machine is negligible, particularly compared against the time it takes to process the part. For the timed transitions, our interpretation is explained in Table II. [1]

Next we extract the *embedded Markov chain* implicitly modelled by our Petri Net. Before we begin, we need to establish a state-naming convention. The Markov chain states represent Petri Net place-token markings, therefore a common state-naming convention is to use a N_p-tuple (where N_p is the number of places in the Petri Net) where each each element is the number of tokens in the corresponding place. We will define our tuple as $\{$*Puf*, *Pub*, *Pui*, *Pd*$\}$, therefore the initial marking (where we have a token in *Puf* and in no other places) would be $\{1000\}$. In other words, the machine is in state 1000 when the machine is up and free.

In this state, only transition *Ti* is enabled (in contrast, in state 0100 both transition *Tf* and *Tp* are enabled). *T*ransition *Ti* has no firing rate associated with it, so it occurs immediately, placing the machine in state 0100 – i.e. the machine is up and

[1]Something worth noting at this point is this: as a *token* travels through the model, it's interpretation can change. More precisely, the interpretation of a *token* is determined by the *place* which it occupies – since *tokens* move from *place* to *place*, clearly the significance/meaning of a *token* can change as well.



(a) Embedded Markov Chain. (b) Reduced Embedded Markov Chain.

Fig. 3. Embedded Markov chain for Fig. 2

busy processing a part. By tracing the possible transitions and subsequent states, we are generating a *reachability tree* of the model, which closely corresponds to the embedded Markov chain. Fig. 3(a) depicts the embedded Markov chain derived from our Petri Net model in Fig. 2.

The next step is to identify *tangible* and *vanishing* states. Vanishing states are states in which the system spends zero time in – i.e. the system transitions out of the state as soon as it enters it. Consider the state 0010, corresponding to there being a finished part in the machine, or a token in *Pui*. The only transition that may fire is *To*, which is an immediate transition. Therefore the machine spends zero time in state 0010; upon entering state 0010, the system immediately transitions to state 1000. Therefore, we call state 0010 a *vanishing state*. Incidentally, state 1000 is also a vanishing state, since the system immediately transitions to state 0100. We may then *reduce* the Markov chain by eliminating these vanishing states; we are left with tangible states, and what we call the *reduced* embedded Markov chain, depicted in Fig. 3(b). Note that the tangible states correspond to a token being present in place *Pub* or *Pd*; the two places that are shaded in Fig. 2.

Based on Fig. 3(b), we may now construct the *transition rate matrix*, which is defined as follows:

$$Q = \begin{bmatrix} -\sum \text{rates leaving state 0} & \text{Rate entering state 0 from state 1} & \cdots \\ \text{Rate entering state 1 from state 0} & -\sum \text{rates leaving state 1} & \cdots \\ \vdots & \vdots & \ddots \\ \text{Rate entering state n from state 0} & \text{Rate entering state n from state 1} & \cdots \end{bmatrix} \quad (1)$$

Therefore, the transition rate matrix for our model in Fig. 2 is:

$$Q = \begin{bmatrix} -t_f & t_r \\ t_f & -t_r \end{bmatrix} \quad (2)$$

Let the probability that the machine is in state "0100" (i.e. machine is "up and busy") be represented by π_1, and the probability that the machine is in state "0001" (i.e. machine is "down") be represented by π_2. These are the only two tangible states in the system, therefore $\pi_1 + \pi_2 = 1$. To determine the

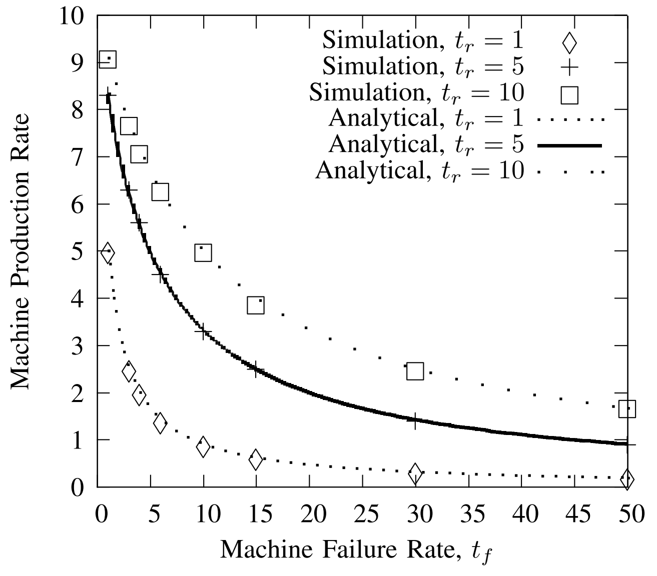


Fig. 4. Machine Production Rate vs. Machine Reliability, for $t_p = 10$.

probabilities π_1 and π_2 we solve:

$$Q \begin{bmatrix} \pi_1 \\ \pi_2 \end{bmatrix} = 0 \tag{3}$$

We now know the proportion of time that the machine is "up and busy", i.e. productive.

Thus far t_p has played no part in our analysis, and in fact it has been revealed that the peak throughput of the machine, i.e. the *rate* at which it processes parts when it is up and running, has no impact on the *proportion* of time that the machine is up and running. We can deduce that the machine production rate is given by $T = \pi_1 * t_p$. Several machine configurations, in terms of failure rate and ease-of-repair, may be compared, as we have plotted in Fig. 4 (the analytical curves).

To conclude this analysis, we summarise the procedure undertaken:

1) Extract embedded Markov chain.
2) Determine reduced embedded Markov chain.
3) Construct transition rate matrix, equation 1,
4) Obtain state probabilities by solving equation equivalent to example in equation 3.

An alternative procedure would be to employ simulation instead of rigourous analysis. This can be carried out by observing the occupancy of places (i.e. the number of tokens occupying a place) over a long period of simulated time. In this way we may obtain the ratio of time that the machine is up and busy. Using this approach yields results that are approximately the same as the analytically calculated results, although as we shall see later there are some potential pitfalls to be aware of, specifically to do with the dynamics of *timed-PN*. These simulation results are also plotted in Fig. 4. Simulation based approaches may be more appropriate for highly complex Petri Net models which can be quite laborious to process rigourously; due to state-space explosion the task can become intractable even with automated computer analysis.

B. Petri Nets vs. Queuing Networks

Queuing Networks (QN) have some difficulty in modelling system features such as blocking, synchronisation, complex prioritisation, accurate resource contention, etc. Petri Nets are inherently capable of incorporating these features, i.e. Petri Nets have a richer immediate[2] expressive range than QN.

On the other hand, one disadvantage of Petri Nets vs. QN is in terms of performance analysis. QN have a more intuitive and direct significance in terms of performance; metrics such as throughput and utilisation are directly observable with queues. Nevertheless, such metrics are obtainable with Petri Nets, albeit with a bit more work. Petri Nets fall back upon their underlying Markov chain foundations for performance modelling. Once a Petri Net model has been developed, complete with initial state marking, a "reachability tree" is generated from which an equivalent Markov chain can be obtained and analyzed, yielding the desired performance metrics. This will be illustrated through an example later.

C. Petri Nets vs. Process Algebra

Donatelli et al. present a comparison of Generalised Stochastic Petri Nets (GSPN) vs. *Process Evaluation Process Algebra* (PEPA) [10]: their findings suggest that both methods are roughly equal in terms of capability and required effort. However, in our opinion one significant disadvantage of PEPA and process algebras in general is that the mathematical expression is not intuitive for non-practitioners. Petri Nets, on the other hand, have a graphical expression that is arguably intuitive for humans. Although some basic explanation of the significance and meanings of the various artifacts (such as tokens, places, and transitions) is required, on the whole Petri Nets may be used to convey structural information about a system to an audience from diverse disciplines.

D. Petri Nets vs. Finite State Machines

The key advantage Petri Nets have over Finite State Machines (FSM) is in terms of ease of evolution. By allowing a variable number of tokens within each *place*, a Petri Net can represent many different states within the same structure. In contrast, a FSM represents a fixed number of states and must be completely modified every time there is a change to state information. This means Petri Nets have a longer useful life-span, are more easily evolved, and are more readily parameterised.

IV. Petri Net Modelling of Computer Systems

In this section we elaborate upon the application of Petri Net (PN) modelling specifically to computer systems, particularly for the purpose of performance modelling. We will employ an evolutionary approach – we will start with a very primitive model, and we will introduce elaborations until we have developed a reasonably sophisticated model.

[2]We use the qualifying term "immediate" here to concede the fact that a complex queuing network may be perfectly capable of capturing all these features, however such a model would certainly not be intuitive to design (or read). With Petri Nets, such features are more immediately expressible, and therefore more intuitive.

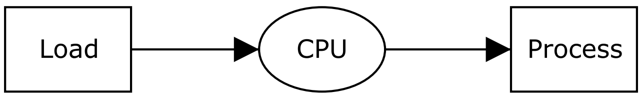


Fig. 5. Simple PN model of a computer.

For the rest of this paper, the following conventions apply: 1) (**X**) – X is a *place*, 2) [**Y**] – Y is a *transition*.

A. Flawed Basic Timed Model

Consider Fig. 5, which we shall mull as potentially the most primitive and simplified PN model of a computer system. [**Load**] represents the loading of data into the *CPU*. Subsequently, once all required data has been loaded into the CPU (indicated by a single token in (**CPU**)) the transition [**Process**] may fire.

Unfortunately, from a timed-PN simulation standpoint, this model is flawed. There are delays associated with each transition; specifically t_{Load} with [**Load**] and $t_{Process}$ with [**Process**], which broadly correspond to the memory access rate and the CPU processing rate. Once we associate these timing parameters to the model, the simulation fails – [**Load**] fires repeatedly, [**Process**] never fires, and tokens pile up in (**CPU**). To understand this behaviour, we must understand the dynamics of timed-PN.

A timed model has a *global clock*. When a timed transition fires, it places tokens in it's target places, and these tokens have an associated *timestamp*, but the global clock is *not incremented*. Recall that a transition is fire-able when all it's source places have tokens. With timed-PN models there is an additional constraint: the global clock time must be $\geq$ the timestamp of the token for the presence of that token in the corresponding place to enable the transition. Only when no transitions are fire-able does the global-clock increment.

Back to our example problem, since [**Load**] is always fire-able, the global clock never increments. Since [**Load**] is timed, each token it places in (**CPU**) is timestamped with t_{Load}, which means [**Process**] is only fire-able when the global clock $= t_{Load}$, but of course the global clock is always zero, hence the observed behaviour. A physical interpretation of this phenomenon is difficult to imagine, but one possibility is that the CPU loads an infinite amount of data through a bus of infinite bandwidth, with rate t_{Load}. Processing this infinite volume of data occurs at a rate of $t_{Process}$, however by virtue of the data-set being infinite, the full data set is never available in the CPU, and therefore processing cannot occur.

This unbounded behaviour is clearly undesirable, therefore we must limit the number of tokens that may be placed in (**CPU**) in any given time. To do this we must associate (**CPU**) with an *inhibitor arc*, or alternatively we may introduce an *anti-place*, which is simply a place with a particular configuration. We have in fact already seen examples of anti-places; recall figures 1 and 2 of the Reliable Machine and Unreliable Machine. The number of parts processed by the machine in any given time was limited to one; this was achieved with (**Puf**), which indicates that the machine is "up and free". When (**Pub**) has one token, (**Puf**) has none and therefore [**Ti**] cannot fire.

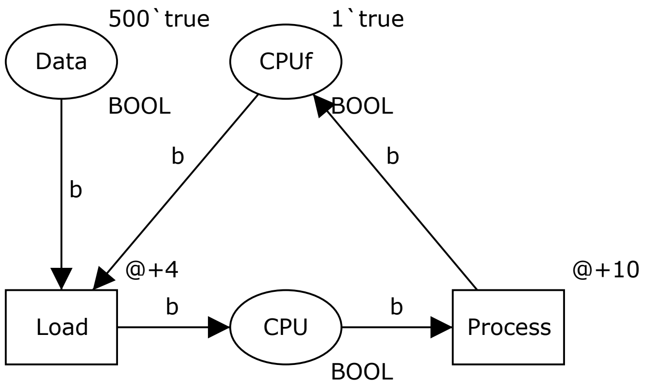


Fig. 6. Improved simple PN model of a computer.

Conversely, when (**Puf**) has one token, (**Pub**) and (**Pui**) are empty, indicating that the machine is empty and ready for a new part, and hence [**Ti**] is fire-able.

B. Improved Basic Timed Model

The solution to the flaw described in the previous section now appears obvious; we must introduce an *anti-place* to limit the number of tokens that can be placed in (**CPU**).

In addition, we will make one more modification. All previous models assumed an infinite amount of work to be done; this is certainly a reasonable assumption to make, however we will now introduce a finite amount of work to be done so that we can directly compare the model time to assess different models. We can implement this by introducing a place with an initial number of tokens corresponding to the amount of work to be done. This is all illustrated in Fig. 6. (**CPUf**) has an initial marking of 1; this means (**CPU**) can only have one token at most in any given time, effectively limiting our processor to working on one workload at a time. (**Data**) has an initial marking of 500, which means at the start of the simulation there are 500 tokens in (**Data**), which are depleted by [**Load**]; once all the tokens are exhausted, [**Load**] can no longer fire, and eventually no other transition will be fire-able either, thus ending the simulation. This effectively means that the computer halts after all the work has been done.

With this model, memory access and processing occurs sequentially, therefore the performance behaviour is very simple, as is illustrated in Fig. 7. Note, however, that our performance metrics in this case are quite different from that discussed in section III-A; in fact while we associated *rates* with the transitions in the unreliable machine example, in this primitive computer model we have associated *delays* with the transitions. This is acceptable: the important thing is to remain consistent so that meaningful analyses may be conducted.

C. Multiple CPUs

We can easily modify our model of a computer system as presented in Fig. 6; all we have to do is change the initial marking associated with (**CPUf**). By setting the initial marking (i.e. initial number of tokens) of (**CPUf**) to 2, we are effectively saying that there are 2 CPUs available.

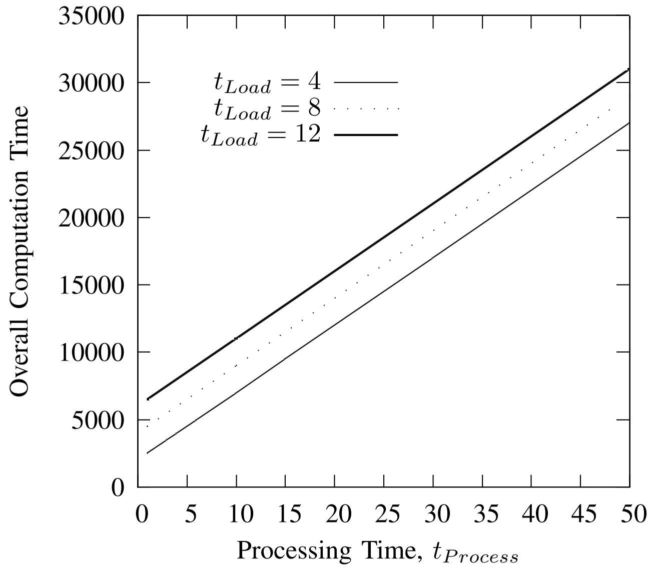


Fig. 7. Primitive computer model performance.

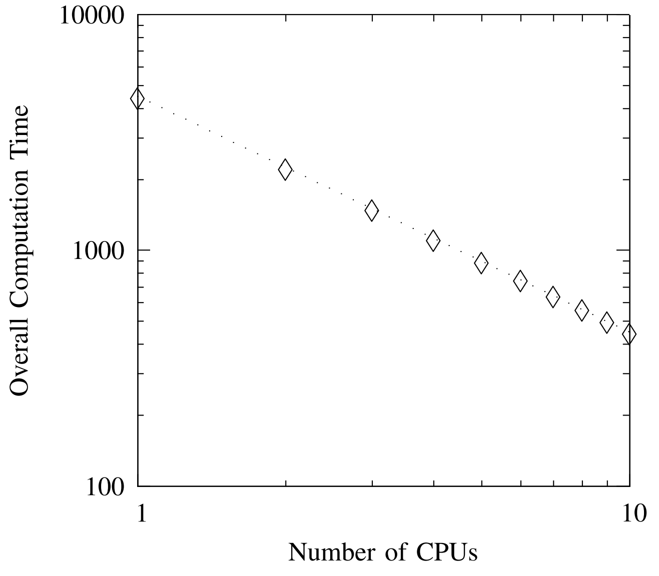


Fig. 8. Primitive parallel computing model performance.

In doing so, we are modelling an embarrassingly parallel computation, and the results are embarrassingly simple and predictable, as illustrated in Fig. 8.

D. Constrained Bus

We now attempt to incorporate a shared constrained memory bus into our model, and our proposal is illustrated in Fig. 9.

The places (**CPU**), (**CPUf**) and (**Data**) should be familiar, as should the transitions [**Process**] and [**Load**]. The place (**BUS**) represents the data bus shared between all the CPUs, while the place (**BUSf**) has a role similar to that of (**CPUf**); it effectively constrains the capacity of the bus. If we decide that a marking of 1 in (**CPUf**) corresponds to 1 CPU in the system, then a

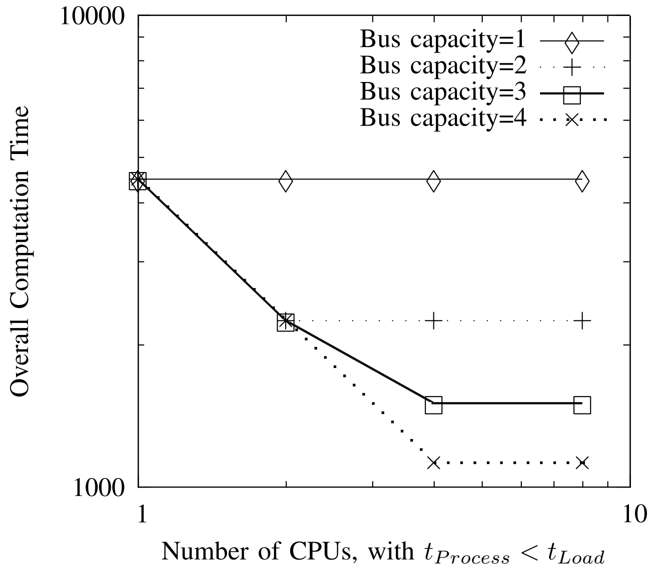


Fig. 10. Constrained bus parallel computing model performance, with dominating load time.

marking of 1 in (**BUSf**) corresponds to the bus capacity to feed 1 CPU. Data is placed onto the bus through the firing of [**Access**], which is fire-able only when there is a token in (**Data**) – which means we have data to process – and a token in (**BUSf**) – which means the bus can sustain the transfer.

Now the behaviour of our model is sensitive to quite a few parameters. Obviously the number of CPUs and the capacity of the bus are critical factors, however the processing delay $t_{Process}$ and loading delay t_{Load} are also significant.

We first model the system with $t_{Load} = 8$ and $t_{Process} = 1$, i.e. $t_{Process} < t_{Load}$; this implies that the program is such that very little computation is performed with each piece of data. The performance of the model under these parameters is illustrated in Fig. 10.

When we set $t_{Process} = 20$, then $t_{Process} > t_{Load}$ and we observe different behaviour, as illustrated in Fig. 11. In this case, the system spends more time crunching data rather than fetching data, therefore there is less pressure placed on the bus. One interesting and potentially valuable finding revealed here is that with workloads that exhibit this 20:8 ratio of computation to communication, we can achieve near-linear speedup with 8 CPUs with a bus with sufficient bandwidth to sustain 3 CPUs; any additional bandwidth provisioned would be wasted.

E. Pipelined Memory-Processor Model

Pipelining is a means of latency hiding, and we will now attempt to model this phenomena. In our model we will assume two discrete pipeline stages; stage 1 performs memory reads, and stage 2 performs computation. This model is conceptually similar to the parallel DMA access method employed by the Cell BE SPE Memory Flow Controller [11].

We will start from the basic timed model presented in section IV-B. Our elaborated model is presented in Fig. 12

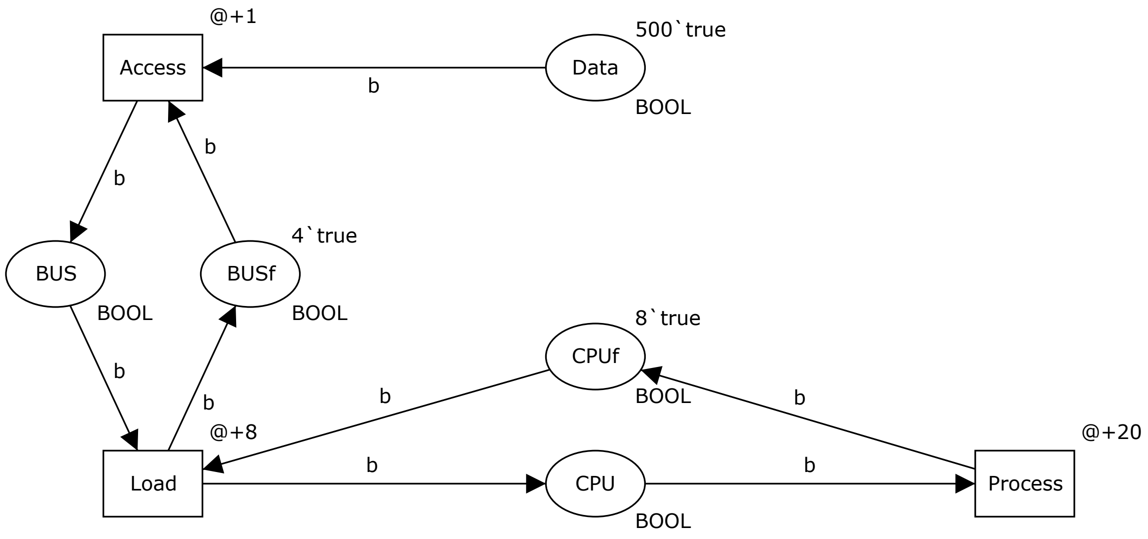


Fig. 9. Constrained bus parallel computer model.

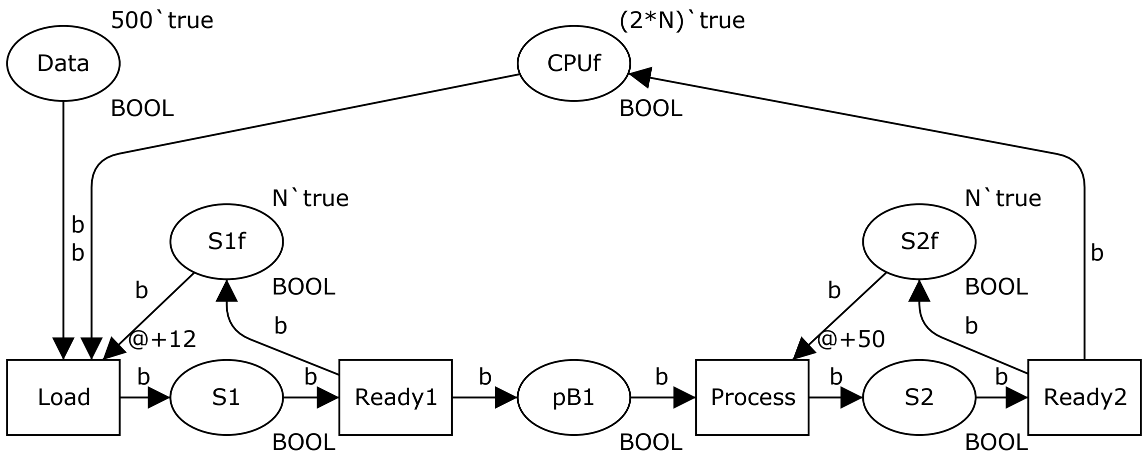


Fig. 12. Simple two-stage pipelined processor model.

The places (**S1**) and (**S2**) are the two stages in the processor, and (**pB1**) represents a buffer between the two stages. (**S1f**) and (**S2f**) are anti-places. N represents the number of cores in the processor – for our discussion we will assume that $N = 1$. The two stages perform two distinct operations in parallel, i.e. [**Load**] and [**Process**]. The transitions [**Ready1**] and [**Ready2**] are merely control transitions, which fire to indicate when the stage is free for new work.

The performance curves estimated from this model are presented in Fig. 13. From the Fig. we can see that once the processor time $t_{Process}$ exceeds the memory load time t_{Load}, the cost of the loads is effectively hidden.

V. Conclusion

Petri Net modelling is an appropriate formalism for the modelling of computer systems. It is a flexible approach, providing rigourous Markov chain modelling when required, and also simulation-based techniques for situations which suffer state-space explosion.

One aspect not explicitly handled was incorporation of application characteristics. Doing so may be as simple as tuning t_{Load} and $t_{Process}$ appropriately.

We have shown how to model several basic concepts and architectural features in high-performance computer systems, such as pipelining, shared busses, and replication. These primitives may be extended and used to model larger, more realistic, and more complex systems.

References

[1] S. Govind and R. Govindarajan, “Performance modeling and architecture exploration of network processors,” in *QEST’05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, 2005.

[2] F. P. Burns, A. M. Koelmans, and A. V. Yakolev, “Analysing superscalar processor architectures with coloured Petri nets,” *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 2, pp. 182–191, 1998.

[3] F. Burns, A. Koelmans, and A. Yakovlev, “WCET Analysis of Superscalar Processors Using Simulation With Coloured Petri Nets,” *The International Journal of Time-Critical Computing Systems*, vol. 18, pp. 275–288, 2000.

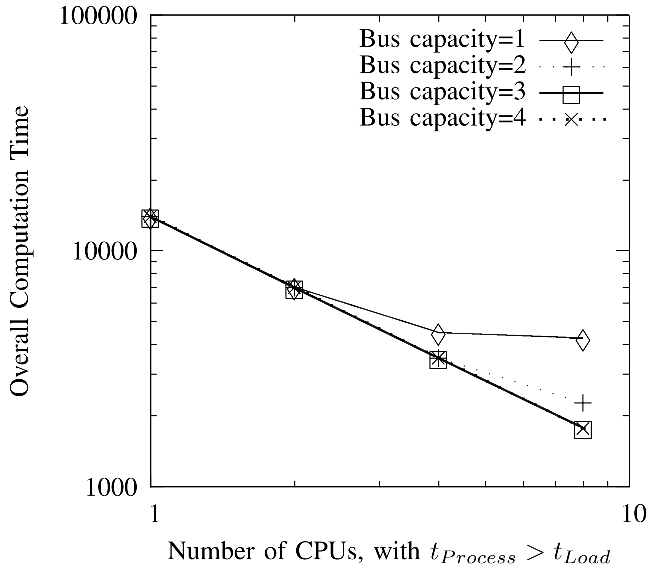


Fig. 11. Constrained bus parallel computing model performance, with dominating processing time.

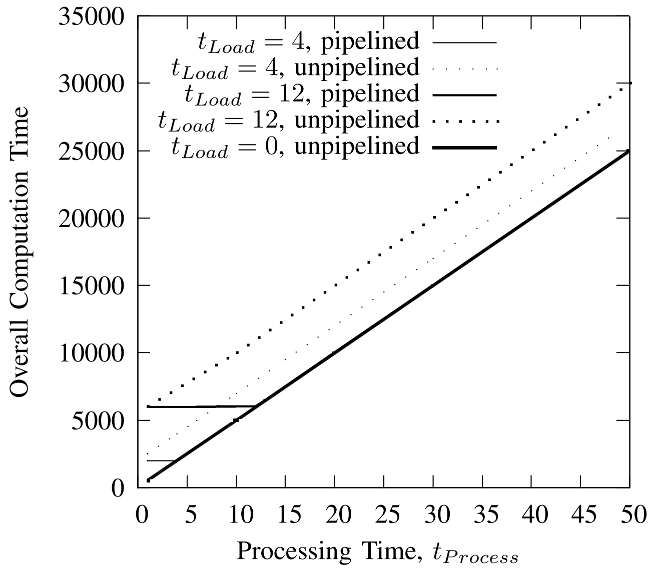


Fig. 13. Pipelined processor model performance.

[4] L. M. Kristensen, S. Christensen, and K. Jensen, "The practitioner's guide to coloured Petri nets," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 2, pp. 98–132, 1998. [Online]. Available: citeseer.ist.psu.edu/kristensen98practitioners.html

[5] A. A. Desrochers and R. Y. Al-Jaar, *Applications of Petri Nets in Manufacturing Systems: Modeling, Control, and Performance Analysis.* IEEE Press, 1995.

[6] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[7] M. Zhou and F. DiCesare, *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems.* Kluwer Academic Publishers, 1993.

[8] L. Montano, F. J. Garcia, and J. L. Villarroel, "Using Time Petri Net Formalisn for Specification, Validation, and Code Generation in Robot-Control Applications," *International Journal of Robotics Research*, vol. 19, no. 1, pp. 59–76, 2000.

[9] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems.* Kluwer Academic Publishers, 1999.

[10] S. Donatelli, J. Hillston, and M. Ribaudo, "A comparison of Performance Evaluation Process Algebra and Generalized Stochastic Petri Nets," in *Proceedings of the 6th International Workshop on Petri Nets and Performance Models*, 1995.

[11] M. Gschwind, "Chip multiprocessing and the cell broadband engine," in *CF'06: Proceedings of ACM Computing Frontiers 2006*, 2006, pp. 1–7.