Methodology and Tools for System Analysis of Parallel Pipelines

M. Fleury, N. Sarvan, A. C. Downton and A. F. Clark

Department of Electronic Systems Engineering,

University of Essex, Wivenhoe Park,

Colchester, CO4 4SQ, U.K

tel: +44 - 1206 - 872911

fax: +44 - 1206 - 872900

e-mail fleum@essex.ac.uk

Summary

The pipelines of processor farms (PPF) design pattern, intended for continuous-flow embedded systems, has been augmented by a software toolkit at the system analysis level. Other relevant approaches to system support employing tools are reviewed. The PPF structure supports incrementally-scalable systems which can meet real-time specifications. An outline of the design and development cycle of PPF systems follows. The paper considers in detail the prediction component of the cycle. A graphical simulation tool for modelling asynchronous pipeline behaviour uses a Java-based visual display. An extended example showing how the performance tool supports PPF design principles concludes the paper.

keywords: toolkit, pipelined system, processor farms, performance prediction, graphical simulation

1 Introduction

This paper considers a methodology and support tools for the design of embedded systems utilising parallelism as one way to gain real-time performance. We have designed a programmer's template which aids the rapid construction of data-farms connected in a prototype pipeline. Existing tools for the analysis of sequential code, prior to parallelisation, have been selected for use within the methodology. We have developed a graphical tool which displays predicted parallel activity through a discrete-event simulation. A complementary performance analysis tool fed by an event trace visualises the performance of a prototype pipeline. Target applications are formed into parallel pipelines, that is a set of independent stages each one of which may incorporate either task or data parallelism. Pipelines may have feedback loops, though currently restricted to one per stage. In order to mask synchronization points, it is permitted to foldback a pipeline. That is to say one stage can perform the work of a later stage rather than wait for feedback from that later stage. Otherwise, pipelines are restricted to a single forward data flow.

Parallel embedded systems research has in the past tended to concentrate on small-scale problems giving rise to numerous one-off solutions, perhaps because there appears to be a dearth of system design schemes. In a competitive research environment this is understandable but it may explain the poor take-up of parallel solutions. Limited guidance is offered to the application engineer in a production environment, where resources are available to tackle larger systems.

Recently, programming models have appeared, such as Bulk Synchronous Processing (BSP) [1], derived from long-term research into PRAM-like algorithmic models. A welcome feature of BSP is a performance model, in this case a parameterized linear model of communication. This performance model, which motivated us, reflects the efficiency of some second-generation multicomputer interconnects in allowing congestion and other non-linear effects to be neglected. However, BSP, actors [2], UNITY [3] and others are programming models, one level below a system development scheme. In fact, the specification language

CSP and adaptations of it already provide a programming model for soft [4] and hard [5] real-time embedded systems. The level at which a toolkit might profitably be aimed is in support of system analysis.

2   Analysis tools for parallel embedded systems

Analysis of larger parallel embedded systems may rely on pre-packaged schemes such as the Yourdon dataflow method [6]. The dataflow approach is bottom-up and avoids fixing the design prematurely. Dataflow is intended to approximate the way people see systems. The advantage to the designer is that the system might be applied to a sequential programming environment, a pseudo-parallel or concurrent system, or a parallel environment. The raw dataflow system does not capture timing information, however, raising questions about its suitability for hard real-time systems.

The PARET environment [7] for multicomputers has similarities with dataflow schemes. The parallel application is represented by a network of nodes connected by arcs. Tokens, representing data or control flow, pass between nodes. Buffers can store tokens. State transitions occur according to node output and input policy, which bears a resemblance to modelling by timed Petri-nets, usually applied to small-scale systems. The user can zoom in at successive resolutions. The PARET system is suitable for a variety of purposes, examples given are a parallel simulator of MOS circuit timing, and a hypercuboid computer interconnect. Aspects of the PARET modelling technique were attractive to us but the displays appeared to lack structure.

The aim of our work has been to produce a machine-neutral system description, which will reflect linguistically-based thought processes in developing a design. Computer environments frequently model an aspect of the world. In our toolkit it is important to present such concepts as 'pipeline', 'data farm', 'flow' and 'hotspot', which do not necessarily exist in other visualizers.[1] ParaGraph [8] is a well-known example of a tool which aims at the widest

---

[1]Earlier transputer tools, Transim and Grail, are an exception.

generality. ParaGraph provides twenty-five displays interpreting the same event-trace data in different ways. In retrospect, ParaGraph's displays lean overly towards the hypercube, which at the time the tool was developed was regarded as the most likely general topology, in part because other topologies could be embedded within a hypercube. By its nature, ParaGraph does not specifically support pipelined systems.

3   Requirements of tools

In our preliminary survey of existing tools, the following user requirements were identified:

Correctness checking Soft real-time systems merely require verification of program behaviour. Safety-critical and hard real-time systems require something more than verification, either comprehensive traces or formal methods of proof should be considered.

Performance debugging Slow downs, bottlenecks or hold-ups should be identified.

Processor utilization is important when costing a solution.

Cross-architectural comparisons can also guide purchasing decisions.

Prediction models can be used in conjunction with analyses of event traces to satisfy these requirements. A prediction model aids the identification of untoward behaviour which emerges from analysis of the event trace. Changing the communication and computation parameters within the prediction model is a method that has been successfully applied [9] in order to project performance to other machine types from timings at the basic-block level on a development machine.

A decision was needed on whether an integrated tools environment was desirable. The term 'integrated' can be used in the sense that all tools have a similar feel. Integration can also imply that the output from one tool feeds into another, a toolset as opposed to a toolkit. In the MAD environment [10], there are: EMU, which instruments and monitors the application; ATEMPT, which provides performance analysis and error detection through visualization; and

`PARASIT`, which simulates the application in order to predict race conditions. We decided to avoid the attempt to lock the user into a toolset and possibly lock ourselves into a restrictive toolset environment. On the other hand, the user is guided through a set of actions by the development methodology and the core prediction and analysis tools are integrated, Section 5.

The form of the user interface needed to be considered. A number of display types were influential:

- The space-time diagram is common to a number of event-trace systems [11]. Its advantage may stem from the persistence of information displayed, allowing the mind to build up a pattern of activity.

- The diagram-based display, such as SPV for the Paragon [12], is a way of showing process meters indicating parameters such as the instantaneous arrival and departure rates of messages, the activity status of a process, and link activity. A disadvantage of a diagram-based display may be confusion if large numbers of processes are involved, to counter which zooming is possible.

- The state-change display is well-suited to showing pipeline activity. In the visual programming tool `HeNCE` [13] the nodes of a graph change their shading according to activity in the associated process.

- `xpvm` [14] has a run-time display system for networked computation using a space-time diagram. Bias introduced by the extra messages relaying the event trace make 'on-the-fly' displays unsuitable for real-time systems.

4 PPF systems

Our toolkit supports the pipelined processor farms (PPF) [15] stylized system design and development methodology, which is envisaged as a parallel extension to existing software-engineering practice. PPF is intended for continuous-flow, embedded systems though not regular, deterministic systems, which are already catered for by a partitioning methodology

5

[16] aimed at systolic pipelines. PPF applications will need to meet constraints such as minimum pipeline throughput, and maximum traversal latency. An output-ordering constraint can also degrade pipeline traversal latency.

Recently, a number of medium-scale, data-dominated, vision and image-processing applications [17, 18, 19, 20, 21, 22], have been parallelized along PPF lines. We tentatively define 'medium-sized' systems as containing multiple algorithmic components, with approximately 3,000 to 50,000 lines of code. Examples of applicable irregular, continuous-flow systems can be found in vision, radar [23], speech processing [24], and data compression [25]. These systems all use data farming and partitioned pipelines in one form or another, but their development took place in isolation from each other.

PPF promotes the notion of a software pipeline, which can subsequently be transferred onto available hardware according to client need, analogously to the way relational database systems are mapped to differing parallel hardware [26]. It has been observed [27] that many parallel algorithms merely form a sub-system of a vision-processing system. A way of combining algorithmic components in a coherent whole is required, though any subsequent changes should be self-contained. Notice that this strategy provides managers with a way of work partitioning. In PPF, a pipeline is a convenient organising entity. Each stage of the pipeline can independently provide the differing requirements of a system's algorithmic components, either through centralized processing, data farming, or some other form of algorithmic parallelism.

By adjusting the system parameters, differing performance goals can be achieved. Previous application studies have established that designs may be 'crafted quantitatively' for a desired set of performance characteristics. By timing the algorithmic components of a system in a sequential setting, those components which form a significant proportion of the whole are identified. The ratio of processing power required at each stage of a pipeline can be initially assessed. For greater confidence (rather than heuristics), second-order statistics or even identification of the statistical distribution formed by the processing times is necessary. The distribution can be estimated by fitting a distribution to the timings' histogram [28].

As timings on sequential code give static requirements, the system designer, aiming for a

6

reduction in hardware costs, will also want to know dynamic effects such as

- the mean, variance and maximum work flow, i.e. the throughput and traversal latency metrics of a particular configuration of a parallel pipeline in the long run;

- the likely steady-state per-stage scheduling regime behaviour;

- what memory requirements will be needed if there are temporary holdups in the workflow requiring buffering;

- the effect of varying compute and communicate parameters; and

- will a particular processor employed at any one stage spend a significant time idling while a less-powerful processor would suit the granularity of the tasks more closely.

The PPF performance tools progress the designer to a resolution of these issues.

5 Development framework

Previous work on the PPF toolkit has entailed identifying semi-manual methods of partitioning existing sequential code, i.e. identifying likely partition points, and transferring the partitioned code to pre-written high-level software templates which will enact individual processing stages. The integrity of sequential code sections is preserved as it is simpler to modify code sections, possibly containing complex algorithms, in a constrained sequential programming environment. This approach is an echo of the CSP model of parallelism, though in the PPF model one aims, in a heinous manner, to preserve as much sequentiality as possible so long as performance goals are met, so as to minimize parallel design effort.

The PPF design cycle is shown in Fig. 1. The performance predictor works by simulation and analytic means. Because it is desirable to compare predicted with actual performance the simulation should set-up a visual impression of activity within a pipeline segment which the designer can compare with recorded activity on the prototyping machine, using a similar display format. The display format of the simulator is deceptively simple as:

7

- the PPF methodology restricts, in terms of the types of pipeline and the use of processor farms, the degrees of freedom in the parallel system development path; and

- extraneous detail is avoided so that the mapping between prototype design and target system(s) is not prematurely fixed.

A pipeline is split into asynchronous and synchronous segments. An asynchronous segment is one in which individual jobs may flow from stage to stage without waiting for the completion of other work. A segment is a set of adjacent stages. A segment can be a sub-set of a pipeline or indeed be the complete pipeline. A synchronous segment delays the flow of jobs through: the completion of all work within a stage (algorithmic constraint), for example until after a row-column Fast Fourier Transform completes; or the arrival of feedback (feedback constraint), for example a quantization level. Synchronous segments can be treated analytically, in contrast to the performance of asynchronous segments of more than one stage. The issue of analytic results is pursued elsewhere [29]. Asynchronous segments require adequate buffering to smooth the pipeline flow. Synchronous constraints create problems in balancing a pipeline.

Because the analysis tool models the physical pipeline, physical connections need to be shown, including feedback loops. For the purposes of the predictor tool which shows a logical pipeline, a feedback loop can be replaced by a nominal pipeline stage which models the delay distribution experienced by the succeeding stage (Fig. 2). A folded-back pipeline can also be unwrapped again to form a linear pipeline by means of first replacing the folding and then providing a nominal stage. A nominal stage, which can be synchronous or asynchronous, will not correspond to a stage in the implemented software pipeline but will correspond to a feedback loop or a foldback stage. The point is that nominal stages allow a normalized representation of pipelines as a linear chain of stages. Fig. 2 (B) is the result of reducing Fig. 2 (A) to normal form, and Fig. 2 (E) is the result of reducing Fig. 2 (C) & (D) to normal form. Once in normal form, the pipeline is amenable to systematic performance analysis.

By alternating simulation and analytic predictions the performance of a complete pipeline

can be built up in a piecewise fashion. Fig. 3 shows three examples of pipelines containing both synchronized and asynchronous segments. Fig 3 (B) & (C) require normalization by removal of feedback lines, whereupon the pipeline can be systematically analysed. For example, in the pipeline of Fig. 3(C), after first replacing the feedback loop by a nominal stage, the performance metrics for the first segment are found. The throughput characteristics act as inputs to the first asynchronous segment proceeding in the direction of data flow, which is simulated. The output metrics of the first asynchronous segment are available for subsequent segments. The maximum latency is additive between pipeline segments. Notice that the ubiquitous central-limit theory in various forms suggests that the result of concatenating random variables, the service times, across a number of stages is a Gaussian distribution, which is described solely by mean and variance. Synchronous and asynchronous pipeline segment results are then combined. Single-stage scheduling regimes using either fixed or variable task sizes are also open to simulation, and analytic prediction.

A form of pipeline algebra is in the process of development. A synchronous segment may contain any finite number of complete asynchronous segments, Fig. 3 (B) & (C). The inclusion of an asynchronous segment within a larger synchronous segment is explained by the effect of a later synchronous constraint on output from a portion of a pipeline which has no constraints. An asynchronous segment is the largest set of adjacent stages, other than the trivial case, that can be formed at any one site in which no synchronous constraints apply. Synchronous segments must contain all synchronous constraints to ensure that the segment is self-contained but cannot include any constraints that are not needed to make the segment self-contained. Notice that a synchronous segment may contain another synchronous segment as a subset, Fig. 3 (B) & (C).

A configuration file is output from the predictor tool. Unlike event-trace files [30], there is no common format for such files. Therefore, a set of format drivers is to be provided. In addition, as configuration files merely provide a topology in the mathematical sense; a high-level description of the pipeline layout is also output, which can be input at the analysis stage.

9

The design cycle can be further automated by allowing timing data files to be loaded from the predictor tool and the data subsequently analysed using either chi-squared tests for binned data or the Kolmogorov-Smirnov test for continuous data or both [31]. The nature of 'real' statistics is that classical distribution fits are not always found. A distribution may be 'almost deterministic' but with some variance. A histogram can be output to allow formation of a judgement but for further assistance a statistics package and specialist knowledge would be necessary.

6 Simulating asynchronous pipeline segments

Analytic results for asynchronous segments require waiting time distributions which are available for exponential distributions [32] but require queueing approximations for other distributions. A discrete-event simulation was constructed without difficulty as an alternative to analytic prediction. A simulation also enabled start-up and wind-down behaviour to be found. The simulation is updated whenever a new job completes. The update time is determined by modelling the job service distribution.

Previous models of performance and scheduling [33] have supposed that work tasks are divided into jobs. The number of jobs within a task can be varied to reach a benign scheduling regime, depending on the statistical nature of the job service time distribution. There may also be savings in message size by grouping jobs into a logical task. In image processing, there is a reduction in border size when spatial filtering of several image rows (jobs) is combined into one task. Conversely, where a task represents a physical unit then splitting it into its constituent parts can reduce latency. This is because the latency of individual jobs can now be experienced in parallel. An example of a physical task is a postcode to be recognized in a vision pipeline. Splitting the postcode into characters (jobs) reduces the postcode latency because then character latency can be experienced in parallel.

Presently, the task size in the simulator may vary between stages of the pipeline but not within a stage. Single jobs are passed between stages in a pipeline. A task is assembled

from jobs held in an inter-stage buffer. The latency shown in a running display is the per-job pipeline traversal latency. Job latencies are grouped into their output tasks, enabling the spread of latencies to be seen. The per-task latency metric is found at the final stage by selecting the highest latency within the set of jobs making up a task. Ordering constraints when grouping jobs within the pipeline are not simulated as they are application dependent but such degradation can be estimated by the use of order statistics [34]. For example, in the postcode example there is an output ordering constraint as the postcodes must leave the pipeline in the order they entered but breaking the postcode into characters may cause one character with extra processing time to hold up others.

6.1   The simulation implementation

The simulation tool was written in the Java^{TM} 1.1 programming language (version 1.2 appeared towards the end of this work) for reasons of portability. Java enables the development of applications without revealing the source code through the intermediate medium of byte-code. Java comes with a comprehensive graphical library. Using a somewhat slower semi-interpreted language with additional byte-code integrity checking in the virtual machine has caused us to rethink approaches to display. Improvements to Java's speed are on-going including Just-in-Time (JIT) compilers and now hot-spot compilers. It is also possible to use 'final' methods to aid the compiler to optimise, and to judiciously arrange when objects are instantiated.

In theoretical work [34] to verify analytical results against simulation, the simulation update order was chosen to suit the calculation algorithm. The simulation loop was subsequently modified to perform updates in the order that the display is updated. An array indexed by pipeline stage and worker process records the outstanding work left at each worker process. A further array records the latency to date experienced by each job.

Advantage was taken of multi-dimensional arrays each dimension of which could have a variable number of elements — a feature of Java. If a job passes to an inter-stage buffer the latency experience is passed with it. Because inter-stage buffers can be unbounded they would

11

normally be implemented as a linked list. However, Java avoids pointers for security reasons and destructor methods are replaced by background garbage collection, preventing memory leakage. However, the utility vector class methods allow the storage of object references and, hence, dynamic data structures. On locating the object, its contents are unpacked. The capacity of the vector is automatically doubled when necessary.

At each step of the simulation the global minimum remaining work time is located. All other work times are decremented on each worker process and similarly all jobs waiting in buffers have their latency incremented. If the global minimum is found on a worker processor at the output stage, the latency and throughput characteristics of the pipeline are updated on the display. Before updating, if bounded inter-stage buffering is set a check is made to ensure that there are empty buffer slots at the next stage. If not, a new global minimum must be found, though the output stage will always remain unblocked.

The simulator is implemented as a single thread, so as to allow the incorporation of other user threads at a future date.

6.2   Display features

A prototype version of the simulator animated the passing of messages by tokens crossing communication arcs. In tests by knowledgeable users on a Pentium-based PC, the display was reported to be slow, not conveying an impression of parallelism. Preliminary work did not use just-in-time compilation but as the AWT is written in native code no great improvement can be expected in that direction. Multiple threads are employed in Java for animation and no doubt the burden of interpreting byte-code leads to some reduction in speed. Screen flicker was also observed, but when double-buffering was deployed to reduce the effect not only was there residual flicker but a further reduction in speed occurred. Java's MediaTracker is another method to prevent flicker, by coordinating image loading.

Animation using individual tokens for each message makes it difficult to distinguish between levels of activity. The display is not persistent, after the passage of a token the display reverts to its previous state. However, an alternative display method which did convey an

12

impression of parallel communication activity and did identify communication 'hotspots' was found.

The communication arcs were changed in colour by analogy with metal heating or cooling. An update is made where necessary at each step of the simulation loop, but the heated state of the arc changes slowly. In addition, the width of the arc is increased or decreased to reinforce the impression. Changing size is a technique adopted from static statistical diagrams [35], which have the same need to meaningfully convey information.

Because the passing of just one message is normally not enough to change the arc colour state, screen flicker is reduced. The end result is a slowly changing display, which as desired establishes a pattern of activity in the user's mind because the display is persistent. The most difficult aspect of using a graduated colour display is arranging step size and choice of colour, and conveying when an arc's communication bandwidth has been saturated. Another indicator might be used to show saturation.

The worker and farmer states are also indicated by colour, either active, idle, or blocked through full buffers. The internal state of any worker is accessed by a pop-up window. However, notice that true pop-ups are not part of the current Java AWT. Arrangement of graphical objects on a display panel is time-consuming, though to maintain portability object placement should be relative. The grid-bag layout was found to be the most convenient pre-supplied format. The tool was tested on a Unix^{TM}-based workstation as well as a PC to confirm portability.

Features of the display (Figure 4) now included:

- fine control of each stage's communication parameters and work distribution;

- zoom windowing on individual farms;

- communication 'hotspot' indication through arc colour and size;

- colour state change of farmer and worker activity (busy, idle and blocked);

- access to individual idling times and work history;

13

- inter-stage and local buffering monitoring;

- running indication of simulation time and performance metrics;

- controls over simulation display speed and state;

- selection of computation rate and interconnect bandwidth scaling (relative to a base setting);

- help through WWW pages.

6.3 Ongoing work

A further review of the frontend concluded that the parameter-entry window display was too stark for novice users. Therefore, a 'wizard'-like set of windows has been added to guide the user through data entry. The communication rate entry window is shown in Fig. 5. The main pipeline window has been augmented to include summary information of pipeline activity, Fig. 6, without the need to go to a zoom window. The latency display graph, which includes automatic rescaling, allows the user to see a persistent record of activity. The bar-chart display of activity is less successful in that respect and may need to be re-thought. Wherever possible, an intuitive graphical display is used with quantitative information available in reserve. Therefore, the pop-up in the zoom window now is a bar display.

Input to the pipeline is either batch, jobs instantaneously available, or exponential arrival rate, which can result in latency at the first stage. Care has to be taken that if input is batch, that the latency is correctly calculated when there is blocking at the first stage of the pipeline. Otherwise, jobs entering the pipeline because of the provision of buffering for blocked jobs experience more latency than those outside the pipeline when there is blocking and no buffering. Provision of serialisation of the simulation, i.e. a reloadable display, and semi-automatic job service distribution analysis is in hand.

The analysis tool development which is linked to the configuration tool is underway. Unlike displays arranged by communication network topology, in which display of differently-routed

messages is not a problem, the analyser display is arranged by communication route function within the pipeline. Feedback loops, which should account for all connection possibilities, may be shown by a set of linear communication arrows placed in some order above the pipeline.

7   Example

To check our work, simulated results were compared to previously implemented applications, an example of which is presented as an illustration. In principle, settings for the simulation could be made prior to parallelisation.

Recognition of handwritten postcodes [20] has been parallelised into three PPF stages: identification of features within each character of a postcode, classification of those features to form a ranked list of candidate characters, and a search to match candidate postcodes against a dictionary of available postcodes. This moderately-sized system, 4.5k lines of code, employs multiple algorithms with data-dependency introduced by the bimodal task service time in the final stage (UK postcodes tested could have 6 or 7 characters). For this application, if the maximum latency constraint were to be exceeded then the computer processing would not keep up with a mechanical conveyor belt. The throughput should be at least ten postcodes/s.

The run characteristics are for an eight module Transtech Paramid with i860 super-scalar/vector processor running at 50 MHz and T805 transputer communications coprocessor with per-link raw bandwidth of 20 MHz. The transputers provided a service layer which included the farmer processes and buffering. Timing a set of 300 (1945) postcodes (characters) and then applying statistical tests, established that the distributions of processing times were approximately deterministic (and not Gaussian as had been supposed before tests) with per character constants of 0.028s and 0.036s for an i860 respectively in stages one and two. The final stage, assuming random ordering in the input file set-up to test recognition accuracy, was matched by a Bernoulli distribution, assigning postcodes with service times of 0.027s and 0.13s in the ratio 155:145. In this instance, all jobs within one task in the dictionary stage were set with the same time. Outward message sizes were 2119 bytes in stage one, less than

40 bytes in stage two, and 112 bytes maximum in stage three. Interstage buffering had been set by trial-and-error at 20 slots, while the local input buffer sizes were 10 slots. The aim had been to find the best throughput if jobs were instantaneously available in the worst case scenario.

Each application has some special features. In the postcode application, differing postcode (task) sizes in the final stage occur which we bracketed by worst (all size seven) and (all size six) best cases. In Table 1, the worst-case estimates are quoted. The error for run-time estimates is 10% or less.

The implementation characteristics may also affect the model's accuracy. Timings had revealed that latency is adversely affected by unfair multiplexing of returning work which in extreme cases can lead to service starvation. On the T805, the multiplex is supported by microcode. A software shuffle of input channels was implemented to ameliorate the effect. To reduce overhead, the shuffle simply swaps the last-serviced channel with the last channel in lexicographical order, which still may not be 'fair' [36]. Latencies are increased if a data-farm, other than the first-stage farm, has its buffers statically loaded at start-up, an effect which the simulation highlighted, resulting in corrective action. Notice that measuring latencies from the moment of entering the first stage does not include latency arising from a blocked first stage. The exponential arrival rate option does model this effect. Other changes to latency measurement are ongoing.

8   Conclusion

We have identified support for system analysis as an appropriate role for a toolkit. Our toolkit is based upon experience in parallelizing soft real-time systems. Systems of interest are irregular, multi-algorithm, and of significant size. A generalised methodology, PPF, guides the user towards a design which can be tailored to match performance specifications. PPF designs form pipelines with parallelised stages. PPF also enables management of the development process. Appropriate tools are selected or have been constructed to support the methodology.

The paper has considered performance modelling and its graphical representation in detail. Synchronization constraints are calculated by analytic means, but freely-flowing pipelines are usually simulated.

Representing a simulation in graphical form and moreover a simulation of parallel activity in a sequential setting is difficult. Token-passing simulation displays are inappropriate for these conditions. Our solution makes use of colour and shape giving a persistent display of communication behaviour. Key performance parameters, such as throughput and latency, are metered and process state changes as the display progresses. The display shows activity at the pipeline and data-farm level. Calibration tests show at most 10% inaccuracy for run-time estimates.

Acknowledgement

References

[1] W. F. McColl. Scalable computing. *Lecture Notes in Computer Science*, 1000:46–61, 1995.

[2] A. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

[4] M. Fleury, H. Sava, A. C. Downton, and A. F. Clark. A real-time parallel image-processing model. In 6th *International Conference on Image Processing and its Applications*, volume 1, pages 174–178, August 1997. IEE Publication No. 443.

[5] E. Verhulst. Non-sequential processing: bridging the semantic gap left by the von Neumann architecture. In *1997 IEEE Workshop on Signal Processing Systems*, pages 35–49. IEEE, Piscataway, NJ, 1997.

[6] S. Goldsmith. *A Practical Guide to Real-Time Systems Development*. Prentice Hall, New York, 1993.

[7] K. M. Nichols and J. T. Edmark. Modeling multicomputer systems with PARET. *IEEE Computer*, pages 39–47, May 1988.

[8] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, 1991.

[9] V. Sarkar. Determining average program execution times and their variance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 298–312, 1989.

[10] S. Grabner, D. Kranzlmüller, and J. Volkert. Debugging parallel programs using ATEMPT. In B. Hertzberger and G. Serazzi, editors, *High-Performance Computing and*

Networking International Conference, pages 235–240. Springer, Berlin, 1995. Lecture Notes in Computer Science Volume 919.

[11] J. C. Yan. Performance tuning with AIMS—an automated instrumentation and monitoring system for multicomputers. In 27th *Annual Hawaii International Conference on System Sciences*, 1994.

[12] *System Performance Visualization Tool User's Guide*, 1993. Intel Corporation, Ref. No 312889-001.

[13] A. Beguelin, J. J. Dongarra, A. Geist, R. Manchek, K. Moore, and V. Sunderam. PVM and HeNCE: Tools for heterogeneous network computing. In J. J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*. North-Holland, Amsterdam, 1993.

[14] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing*. MIT, Cambridge, MA, 1994.

[15] A. C. Downton, R. W. S. Tregidgo, and A. Çuhadar. Top-down structured parallelisation of embedded image processing applications. *IEE Proceedings, Part I (Vision, Image and Signal Processing)*, 141(6):431–437, December 1994.

[16] J. G. McWhirter. Algorithmic engineering in adaptive signal processing. *IEE Proceedings Part F (Radar and Signal Processing)*, 139(3):226–232, 1992.

[17] A. C. Downton. Speed-up trend analysis for H.261 and model-based image coding algorithms using a parallel-pipeline model. *Signal Processing: Image Communications*, 7:489–502, 1995.

[18] H. P. Sava, M. Fleury, A. C. Downton, and A. F. Clark. A case study in pipeline processor farming: Parallelising the H.263 encoder. In *UK Parallel '96*, pages 196–205. Springer, London, 1996.

[19] A. Çuhadar, D. Sampson, and A. Downton. A scalable parallel approach to vector quantization. *Real-Time Imaging*, 2:241–247, 1996.

[20] A. Çuhadar, A. C. Downton, and M. Fleury. A structured parallel design for embedded vision systems: A case study. *Microprocessors and Microsystems*, 21:131–141, 1997.

[21] M. Fleury, A. C. Downton, and A. F. Clark. Pipelined parallelization of face recognition. *Machine Vision and Applications*, 1998. submitted for publication.

[22] M. Fleury, A. C. Downton, and A. F. Clark. Co-design by parallel prototyping: optical-flow case study. In *High Performance Architectures for Real-Time Image Processing*, pages 8/1–8/13, 1998. IEE Colloquium Ref. No. 1998/197.

[23] M. N. Edward. Radar signal processing on a fault tolerant transputer array. In T.S Durrani, W.A. Sandham, J.J. Soraghan, and S.M. Forbes, editors, *Applications of Transputers 3*. IOS, Amsterdam, 1991.

[24] S. Glinski and D. Roe. Spoken language recognition on a DSP array processor. *IEEE Transactions on Parallel and Distributed Systems*, 5(7):697–703, 1994.

[25] A-M Cheng. High speed video compression testbed. *IEEE Transactions on Consumer Electronics*, 40(3):538–548, 1994.

[26] J. Spiers. Database management systems for parallel computers. Technical report, Oracle Corporation, 1990.

[27] S-Y. Lee and J. K. Aggarwal. A system design/scheduling strategy for parallel image processing. *IEEE Trans. on PAMI*, 12(2):194–204, February 1990.

[28] D. Durand, T. Montaut, L. Kervella, and W. Jalby. Impact of memory contention on dynamic scheduling on NUMA multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1201–1214, November 1996.

[29] M. Fleury, A. C. Downton, and A. F. Clark. Performance metrics for pipelined processor farms. *IEEE Transactions on Parallel and Distributed Systems*, 1998. Submitted for publication.

[30] P. H. Worley. A new PICL trace file format. Technical report, Oak Ridge National Laboratory, Oak Ridge, TN, USA, September 1992. Report ORNL/TM-12125.

[31] D. E. Knuth. *The Art of Computer Programming*, volume 2/ Seminumerical Algorithms. Addison-Wesley, Reading, MA, 2 edition, 1981.

[32] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of Scheduling*. Addison-Wesley, Reading, MA, 1967.

[33] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Trans. on SW. Eng.*, 11(10):1001–1016, October 1985.

[34] M. Fleury, A. C. Downton, and A. F. Clark. Modelling pipelines for embedded parallel processor system design. *Electronic Letters*, 33(22):1852–1853, 1997.

[35] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CO, 1983.

[36] D. M. Ho. Variations of ALT implementation on a transputer. In D. May and K. L. Kunii, editors, *Proceedings of the 3rd Transputer—Occam International Conference*, pages 195–207. IOS, Amsterdam, 1990.

simulated				
pipeline worker ratio	run-time (s)	throughput postcodes/s	mean (s) latency	max. (s) latency
3:3:1	24.2	12.39	1.2	1.6
2:3:2	27.7	10.83	0.6	0.7
1:4:2	54.6	5.49	0.3	0.4
2:2:3	35.3	8.50	1.0	1.0
3:2:2	35.1	8.56	1.0	1.0
implemented				
3:3:1	26.20	11.43	1.02	1.27
2:3:2	28.65	10.47	0.57	0.86
1:4:2	56.90	5.33	0.78	0.79
2:2:3	35.90	8.36	1.13	1.28
3:2:2	35.90	8.35	1.24	1.60

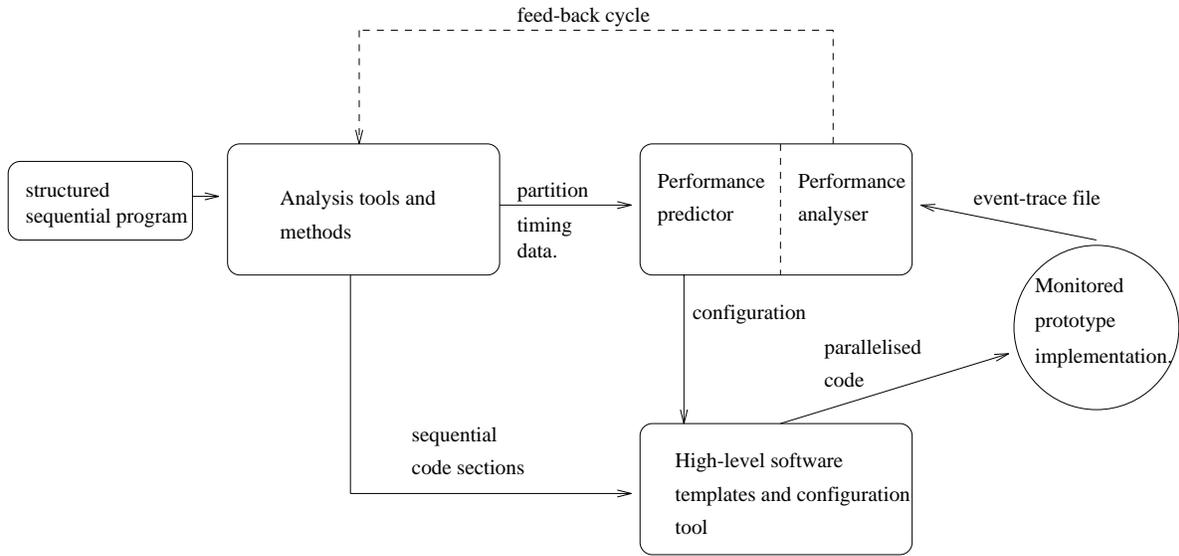Table 1: Comparison of simulated with timed results
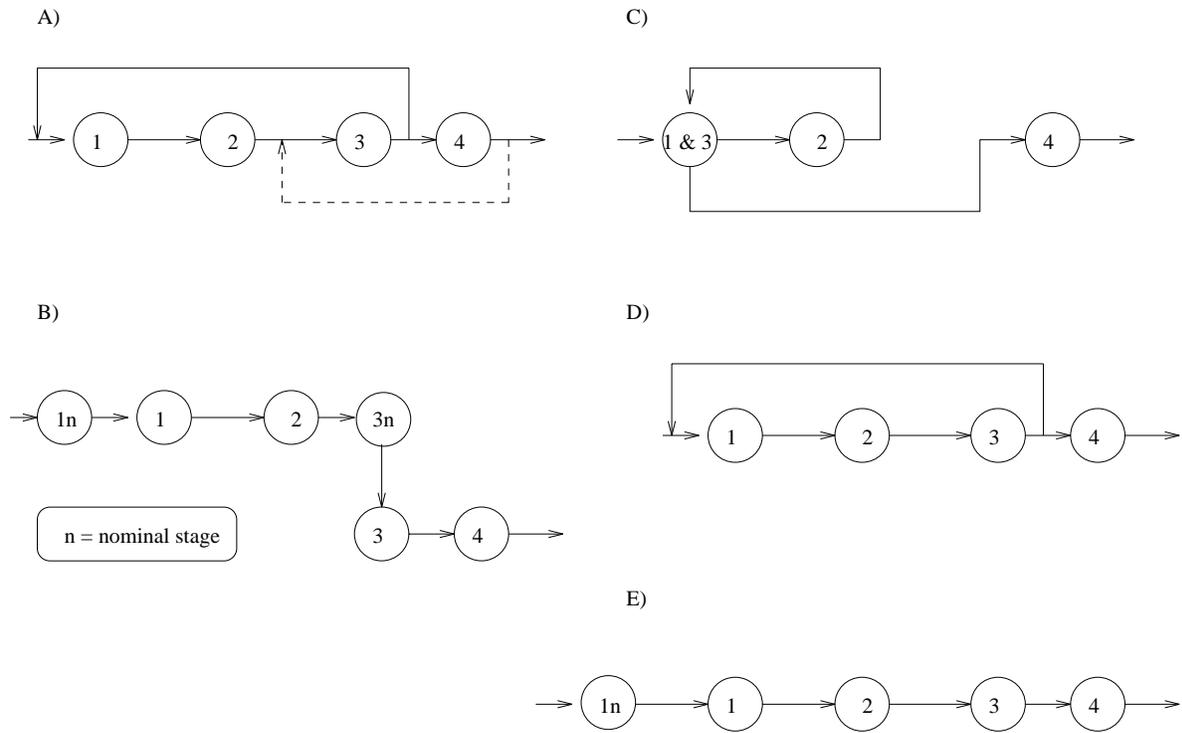
Figure 1: PPF design cycle

Figure 2: A) two simple feedback loops B) replacement by nominal stages C) folded-back pipeline D) after unwrapping E) after substituting a nominal stage.
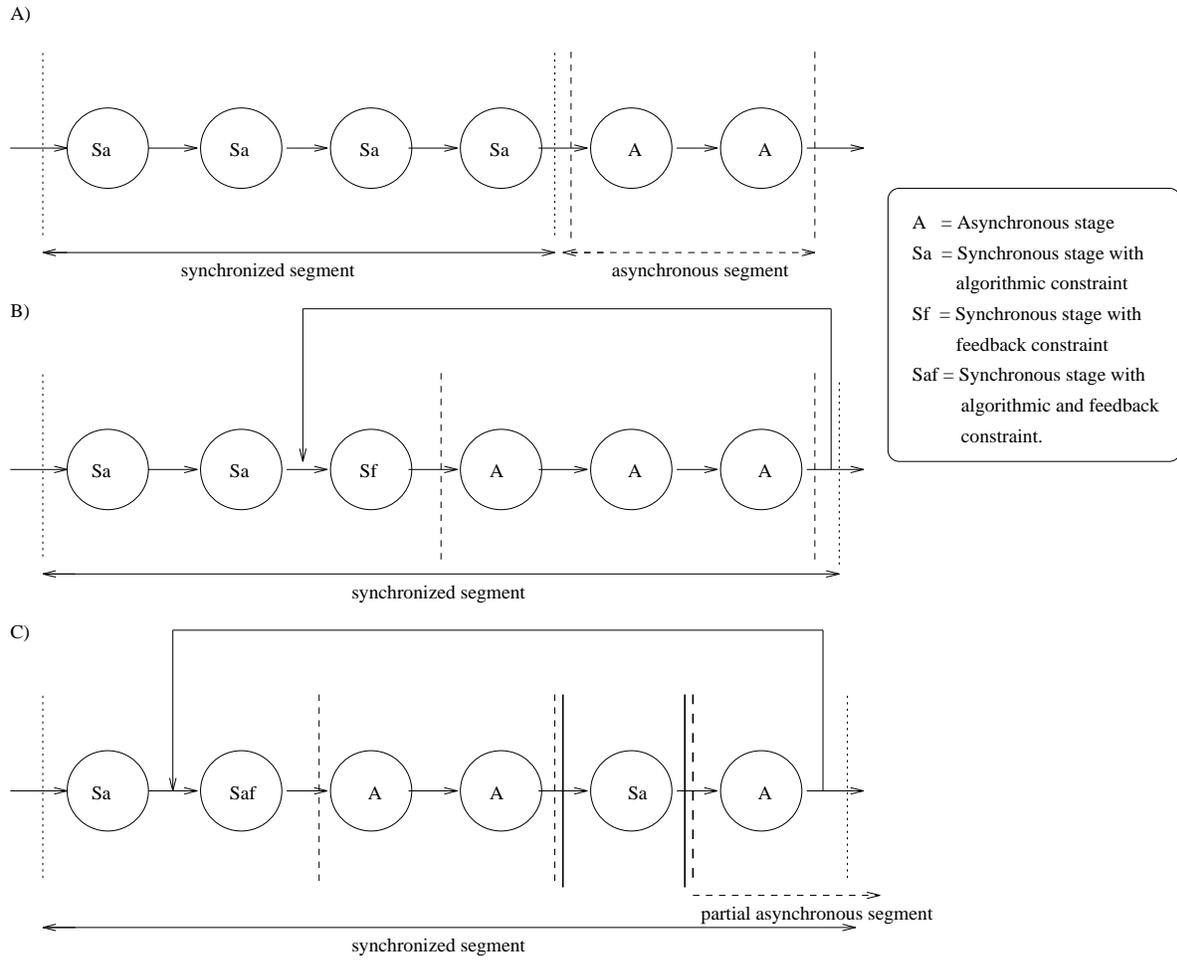
A)

Sa → Sa → Sa → Sa → A → A

synchronized segment   asynchronous segment

A   = Asynchronous stage
Sa = Synchronous stage with
        algorithmic constraint
Sf = Synchronous stage with
        feedback constraint
Saf = Synchronous stage with
        algorithmic and feedback
        constraint.

B)

Sa → Sa → Sf → A → A → A

synchronized segment

C)

Sa → Saf → A → A → Sa → A

synchronized segment   partial asynchronous segment

Figure 3: Pipeline splittings: A) disjoint segments B) singly nested segments C) multiply nested segments
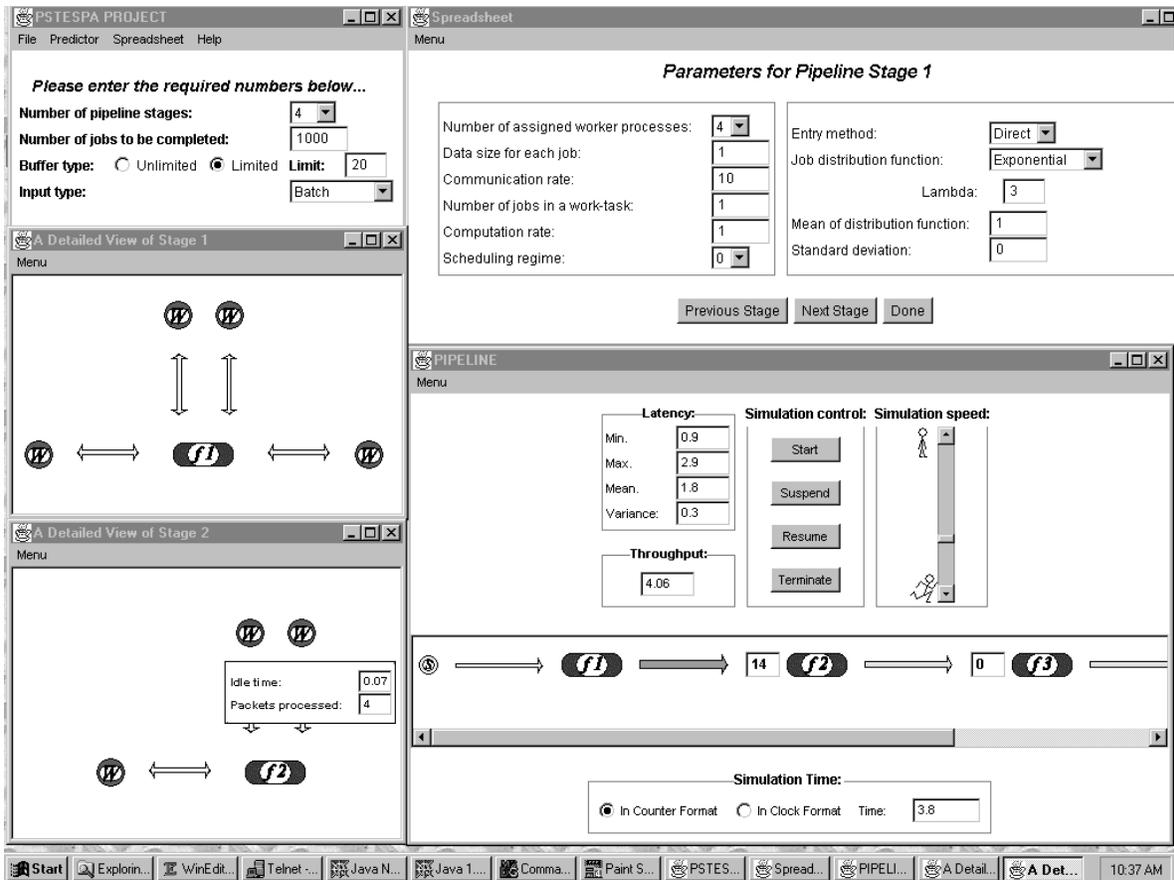
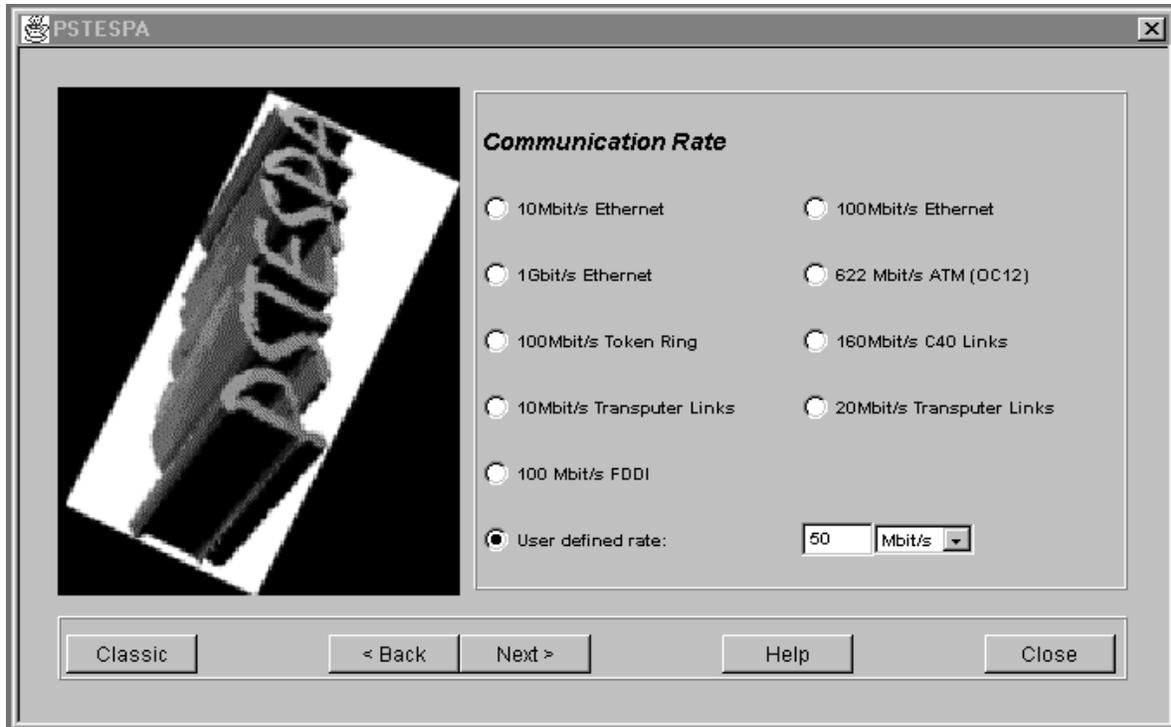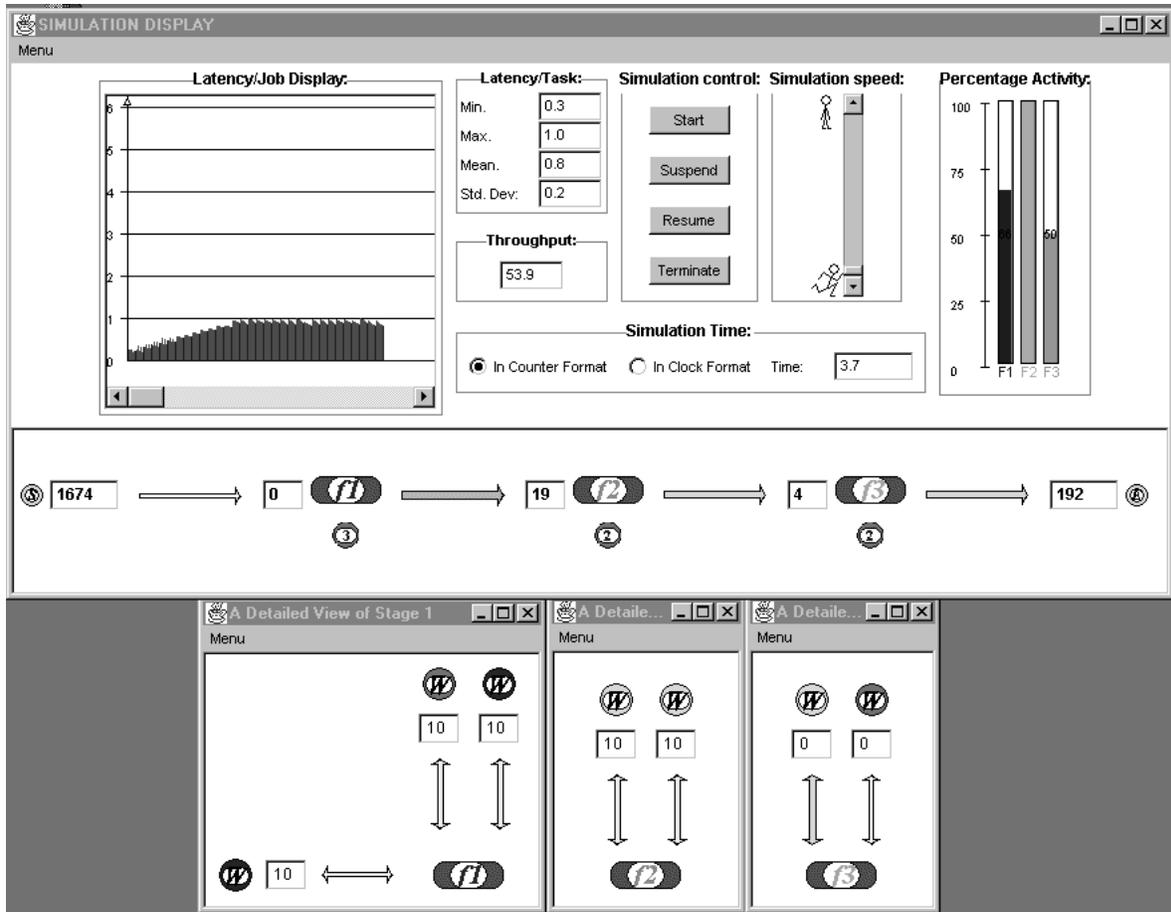Figure 4: Screen shot of the simulation predictor tool

Figure 5: Sample window from data-entry wizard

Figure 6: Revised simulation window