Analysis Prediction Template Toolkit (APTT) for Object-based Computation

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

Dept. of Electronic Systems Engineering,

Wivenhoe Park, Colchester, CO4 3SQ, U.K

tel: +44 - 1206 - 872817

fax: +44 - 1206 - 872900

e-mail fleum@essex.ac.uk

Abstract

APTT (the Analysis, Prediction, Template Toolkit) is an integrated set of visualization tools aimed at the design of continuous-flow, multi-algorithm embedded applications in the multimedia, signal-processing domain. APTT is constrained by a pipelined design pattern, with each stage of the pipeline capable of supporting internal parallelism. APTT includes three tools. The graphical simulation tool predicts pipeline metrics: memory; interconnect bandwidth; throughput; and latency (both mean and maxima); before parallel partitioning is carried out. A post-mortem trace analysis tool with the same format as the predictor tool enables performance of parallelised applications to be checked and optimised. A semi-manual code generator tool has been designed to support APTT templates, a means of rapidly prototyping processing pipelines. The results for performance prediction with the predictor tool on a machine-vision application are compared with actual execution times for a realistic application, and show a prediction accuracy within 10%. The worked example also includes details of a form of high-level codesign whereby cross-architectural comparisons of performance can be made. The paper discusses a template for an actor software object, a semi-dynamic structure with object-loading and reflection based around Java Remote Method Invocation (RMI).

1 Introduction

The Analysis Prediction Template Toolkit (APTT) is a portable visualisation package implemented in the Java programming language, and aimed at constructing medium-grained distributed, and parallel applications. APTT is intended as a prototyping environment with an iterative design cycle, and facilities to project performance to alternative hardware architectures. As a result, an application design is not fixed prematurely. The class of applications of interest can be characterised as soft real-time with a continuous flow of data, for which pipelines are a natural structure. Soft real-time systems do not generally have to meet critical deadlines though they do have to meet performance targets such as a given throughput or traversal latency. Such systems are data-centric, not control-centric. Candidate applications exist in video coding [1], vision [2], and image processing [3]. Recent industrial interest in networked embedded systems [4] has led us to consider the modifications needed to cater for processing pipelines across mobile networks.

APTT is built around a design pattern consisting of a pipeline of data farms each one of which can incorporate internal parallelism, Fig. 1. A pipeline with a single backplane [5] has the advantage that it can map onto a variety of architectures. A farmer has one set of connections to a high-bandwidth bus making the data-farm the minimum replaceable unit (MRU) in a dependable pipeline [6]. Sequential bottlenecks are masked by single processor stages, possibly accelerated. Each parallel stage of the pipeline has a farmer responsible for distributing work to a set of workers which may be on a subsidiary network. The deployment of worker tasks is determined by the relative per-stage workload, the desired global throughput, and the desired pipeline traversal latency. Feedback loops introduce synchronous constraints. For practicality, only one such loop is permitted at each input site. The design pattern is an extension and generalisation of a number of data-farming schemes: some NOW-based [7]; some based on dedicated multicomputers [8]; some constructed as algorithmic skeletons [9]; and some implicitly invoked in parallel extensions of C++ [10].

APTT includes (in order of their use):

- a visual prediction tool for simulating PPF systems, based upon data extracted by top-down profiling of the original sequential application;
- an instrumented processor farm template, which provides a communication library for implementing message passing for parallel pipelines (or indeed a single processor farm), and instrumentation to enable post- mortem analysis of test runs of the parallelised application;

and

- a visual trace tool for post-mortem analysis of parallel execution, with a similar graphical user interface (GUI) to the simulation tool.

Parallel development proceeds from a sequential simulation on a single workstation, through a 'parallel logic' design phase using a network of workstations to a 'parallel performance' phase of execution optimisation on the target parallel processor system. Depending upon the application, the 'parallel logic' and 'parallel performance' phases could take place in the same environment, but the APTT software tools have been implemented for a number of architectures. These include: a Transtech Paramid distributed-memory multicomputer with i860 computation engines connected by means of transputer communication coprocessors [11]; a network of workstations connected by the socket API combined with SunOs 4.1 lightweight processes [12]; and a network of 68030-based VME systems running VxWorks real-time executive [3].

Our current templating experience has been with the RMI object request broker (ORB), which supports dynamic object loading. Standardisation allows Java components written by others to be incorporated as computational units into a data-farm pipeline, and for other systems to employ a data-farm pipeline as a computational resource. The data-farm pipeline becomes a container structure into which are inserted differing compute-intensive applications. The ability to put aside low-level communication issues allows more attention to be paid to fashioning a dynamic system of object creation and configuration, leading towards a ubiquitous processing paradigm for distributed computation.

The rest of the paper is organised as follows. Section 2 outlines the design cycle that APTT is intended to support. The two main graphical elements of the APTT toolkit are described in Section 3. The APTT programming template has taken a number of forms, but the most recent embodiment is described in Section 4 which accords with a prevailing interest in middleware and higher-level software objects linked by that middleware (in a pipeline). A code generator, Section 5, adapting ideas from GUI builders seemed an appropriate way to tie in the template to the graphical environment. Section 6 is a worked example for a handwritten postcode recognition pipeline, demonstrating the utility of the predictor tool. An extension of the prediction tool to cross-architectural comparisons is the subject of the further results in Section 7, a form of high-level codesign. Since we already run pipelines on NOWs it is not difficult to see that the next step will be a processing pipeline on mobile networks, which Section 8 proposes. Finally, Section 9

summarizes and draws some conclusions.

2 Design cycle

APTT underpins the design cycle shown in Fig. 2. Profiling the development code enables a tentative allocation of functions to different stages of the pipeline based on mean timing ratios. We originally employed `gprof` [13] which samples the program counter at quantised intervals but switched to `Quantify`[14] not only because a graphical analysis of the function call graph was available but also because object-code instrumentation overcame some of the deficiencies of sampling [15]. Quantify helped to identify the compute-intensive strand of Gaussian mixture probability calculation from within a speech-recognition application run [16] in which 1188 different functions were called. The identification was by visual inspection of the call graph but it cannot be discounted that the identification could be done by automatically scanning the call graph guided by an expert system. Normally, the potential for loop parallelism will determine the per-stage weighting of processors. A set of inner-loop timings enable distribution fitting by statistical or other means. Identifying a job as one iteration of the inner loop, jobs can then be grouped into tasks. Note that the grouping results in a change in distribution as the addition of distributions is a convolution. Optimal task groupings by means of order statistics are explored in [17], and adapted for data-farms in [48], thus this issue is not pursued further in the present paper. Two classes of problem are catered for: vision applications where the semantic content pre-determines the extent to which an optimal grouping can be made; and low-level image processing where there is generally more latitude in the way image segments can be grouped.

To arrive at an optimal arrangement also requires consideration of dynamic effects such as communication bandwidth both across the pipeline backplane and in subsidiary farms. Buffering can be added, at a cost in memory, to enhance the bandwidth and smooth work flow. Memory costs are now generally less than communication bandwidth. Though synchronous pipelines are best dealt with analytically [18], asynchronous pipelines are difficult in the general case, requiring the statistics of large deviations [19], which makes a discrete event simulation an attractive and malleable tool.

Integrated with APTT is a parallel-application generator (an automated template tool), which takes as input high-level descriptions of shared data structures and inner-loop sequential code sections. Boiler-plated parallel code can be output. Built-in trace instrumentation and communi-

cation calls are intended for two generic targets: a network or cluster of workstations, or a modestly parallel machine. The former is suitable for verification of correct working of the application and the latter for performance testing or tuning. In fact, we have employed a logical clock [11] to timestamp traces for the parallel logic stage while a global clock system which amalgamated a number of techniques was designed for the parallel multicomputer. By means of the trace file, the simulation prediction can be compared to actual execution on the target hardware. Trace files are held in a standard format [20]. As with the configurer tool, an intermediate form is utilised for the application code so that communication primitives can be generated for a variety of machines.

Both simulation (prediction) and trace (analysis) are presented in a similar graphical format, though the former is intended as an abstract model while the latter is a physical model. A key difference is that feedback is explicitly represented in the analysis tool while the predictor reduces feedback to a flat representation.

There is nothing in the APTT design model which precludes heterogeneous processing, either between stages of the pipeline or within a pipeline stage. In the latter case, the most likely division is between farmer and data-farm. The data-farm acts in effect as a coprocessor or accelerator of software. At present, partitioning between farmer and data-farm is at a high-level through the predictor tool. In this, APTT is similar to the software partition methodologies which are one side of hardware/software codesign [21]. It is also possible that detailed partitioning between farmer and data-farm coprocessor could take the form of hardware-in-the-loop, which is the other side to codesign [22].

3 Graphical representation

A graphical interface is the user's view of a toolkit, and in terms of person hours of design effort has been the most expensive part of the APTT development. Our design aims to exploit familiar user interface paradigms in terms of navigating data entry screens and utilising simulation and trace tools; thus reducing the user's learning time. The interface was written in the Java programming language, which has enabled a trivial port between Windows NT and Unix operating systems which would not have been possible with X-window software.

A problem with previous analysis visualisation tools, such as ParaGraph [23], is that a highly animated display occurs, rather like a cartoon film. The user may find it difficult to establish a pattern. Moreover, in seeking generality, with twenty-four ways of presenting data, no structure

to the ParaGraph tool's usage was provided. An over-animated display also reinforces the sequentiality of the simulation whereas the pipeline represented in reality has both local and general parallelism.

Fig. 5, showing a summary of statistics entered, is taken from the APTT data-entry 'wizard' which has a familiar look-and-feel to ease user adaptation. Fig. 6 shows a snapshot of the predictor running the postcode simulation. The pipeline backplane occupies the main window with details of the stage activity such as buffer and processor usage available from subsidiary windows. Processor activity is shown using colour by analogy with stop/go displays. Again using the linguistic associations of colour, the communication arrows change colour from black, through red to white to highlight 'hotspots'. Provided the shades range in tone then this type of display is suitable for monochrome as well as colour display. The arrows also widen and contract, which is a format well-known in static displays [24]. Mean bandwidth, rather than instantaneous bandwidth is displayed so that the rate of display change is smoothed out allowing the viewer to establish a pattern. The colour scaling is adjustable to centre on and bracket critical data rates as the variation across the full bandwidth range would be too low to show up, Fig 7. Latency is also indicated in a persistent display. Jobs are marked off at task boundaries, with the task latency determined by the slowest job. Though persistent displays convey more information, they need to be balanced with features marking progress, which is why the processor activity diagram and message motion arrows are included. Qualitative, graphical information is balanced by the quantitative instantaneous information in the centre of the display. The user can control the speed of the display. There are facilities to customise the display and select the random-number generator.

Also included in Fig. 6 are pop-up windows (activated by double-clicking on the farmer icons) which allow the activity of individual farms to be viewed. The pop-up windows themselves have a further pop-up (not shown) which gives the number of jobs processed and the activity level of each worker process. In general, qualitative information is given priority but quantitative information is always available.

Fig. 8 shows the analyser window with the postcode trace running and the configuration set-up from an intermediary format file. As instrumentation is inserted transparently to the application the display does not try to infer performance statistics other than run-times.

4 Templates

The parallel pipeline structure has been captured in a number of templates[1]. In this paper, we describe the Java RMI-based template which we have prototyped on PC networks. Earlier versions of the template are described in [11, 25]; they were closer to a relaxed version of CSP. The RMI template has a worker module which is designed as a pragmatic adaptation of the actor paradigm [26, 27], Fig. 9, and the farmer module acts as a work scheduler.

An actor is a form of dynamic object. The key problem when combining parallel and distributed systems with an object-based system is encapsulating the parallel structure within each object [28]. Implicit methods of encapsulating parallelism such as path expressions [29] are possible. In opting for an explicit means – an object manager – APTT is similar to a number of distributed-object programming environments for example SR [30], ARGUS [31], and ALPS [32]. In addition, the object manager, or in our case data-farmer, can perform task scheduling.

The template design decouples the (farmer) client from worker (server) by polling, moving away from a synchronous model of computation. Parallel slackness and buffering are user-level techniques employed by us to reduce communication latency thus widening the class of applications suitable for RMI. Though we have called our design a template in fact a more apt description would be a software component, in the sense that a component encapsulates dynamic behaviour in contrast to objects which are passive. The correct features of any component design might be arrived at through categorical data theory [33].

The extent to which this model of computation will be possible depends on the facilities supported by a particular ORB, and the performance, which will depend on factors such as choice of protocol (HTTP/IIOP/DCOM), levels of indirection, class loading latency, and security if reflectance is employed.

4.1 The APTT template system

RMI has a three-layered software architecture with transport level protocol (TCP/IP circuit-oriented) handling at the lowest level, and choice of communication model, (unicast point-to-point), at the intermediate level below the application layer. Because RMI is specialised for Java applications, remote object loading, dynamic class and stub loading are supported in APTT.[2]

[1]The term 'template' refers to a way of guiding the construction of parallel pipelines and not in the C++ sense as a generic class.

[2]Dynamic class loading is also a feature of mobile agent systems constructed under RMI.

Polymorphic loading can occur, whereby a sub-type of a declared type is loaded. On each processor, a registry must be run to act as a nameserver for remote object method invocation. The registry is not persistent as remote object garbage collection takes place by means of reference counting.

In the APTT template, the Java vector class, which can be made to transparently grow and shrink as work requests are serviced, removes the need to provide concurrent access management. To avoid synchronous delay the implemented design requires the data farmer to poll the remote worker processes acting as servers. Conveniently, this fits a polling queueing model of performance estimation. Java's pre-emptive priority-based thread scheduling can be adapted to provide a responsive structure.

4.2 Worker module

The worker module is designed to offer a range of facilities which at infrequently spaced intervals can be changed by object loading. The intention is to overcome the limited resources likely to be available at the mobile station.

In Fig. 10, the threads (ovals) have high priority given to communication facilitating threads while worker tasks threads, each with a local message queue, run at reduced priority. In fact, each module contains a number of worker task instances which can either provide different functionality or multiple instances of the same functionality. Parallel slackness [34], whereby if one instance is blocked another instance of the same worker task can take over, is a method of masking communication latency originally introduced to allow one architecture to simulate another. There are limits to the number of threads that can be gainfully employed [35]. Therefore additionally, buffering of messages containing work requests can mask latency. Allocation of work is arranged by a local work manager, which provides an ordering structure. Scheduling of threads is divorced from ordering through a highest priority user-level scheduling thread. The scheduling thread suspends itself for a given time interval. When it is rescheduled by the Java operating system, the scheduling thread adjusts the priority of the worker task threads before suspending again. The RMI server includes the skeleton stub created by `rmic`.

On inception, worker modules multicast their names and associated addresses. Associated with each name can be the computational services offered. Multicasting is provided through the Java socket API. The intended multicast recipient is the farmer module. If and when a worker module becomes available then the farmer module sends out requests for work processing. Multicast names and services are also collected by worker modules, allowing co-operative, intra-farm, processing to

take place. A rather limited example of co-operative processing in work farms is the exchange of border regions during image processing. Multicasts are also employed for distribution of start-up parameters. In some applications, such as the H.263 hybrid video encoder [36], global data may need to be distributed after every round of processing, e.g. the current quantisation level.

4.3 Farmer module

Once sufficient work messages have been sent out, the farmer module polls the worker processes for processed work. There are a variety of servicing schemes [37] for arranging polling such as waiting for work before proceeding, or waiting for a set number of processed items, but these are beyond the scope of this paper. On distributed-memory multicomputers, a farmer would not normally poll for work. Instead, worker processes request work on completion. The advantage of casting the relationship in this way is that the worker processes can balance their own workload. However, polling has been applied to bandwidth sharing by dynamic link-swapping in an otherwise fixed connection system [38].

As an alternative to work requests, an object not designated as remote can be serialised or reified and sent as a standard application parameter from the farmer module to the remote server, the worker module. This allows the remote worker after suitable casting to execute or revert the methods of that object. Due to polymorphism it is not necessary for the receiving worker to be aware of the implementation of a method. In this way, the worker module acts as a meta-object acting transparently to the methods embedded in the worker tasks which are the object itself. At the moment, this is only possible because RMI is confined to Java and hence the internal representation of objects is fixed. In a similar way, stubs, which are designated as remote, can be dynamically sent to a farmer module, the client, thus aiding in configuration of the data-farm pipeline. These reflection facilities have previously been available only in actor systems, e.g. ABCL [39] and Merle IV [40] designed before the onset of mobile stations. In effect, reflectance is a form of code reuse or inheritance for dynamic systems.

5 Code generation

Code generators (CG) [41] enable application code to be entered in a schematic manner in a graphical format. [42] for Linda applications is an example of a code generator which is an enhanced text editor. The model for construction of the APTT code generator was SpecTcl

[43], which is a GUI constructor written as an application of the scripting language Tcl. The APTT code generator design allows a graphical pipeline to be constructed in a guided fashion from a menu of farmers and workers, Fig. 11. Each module has a set of properties associated with it, which appear in a pop-up menu. An example property would be a label identifying which application code extract is to be executed. Snippets of code are entered through a text editor allowing subsequent changes. The CG end-user is presented with a dataflow model. A set of triggers are needed to start processing. The concept of triggering stems from graphical programming environments [44]. The layout of messages is entered in a separate part of the CG editor. Once the construction process is complete there are commands to compile and run. Since the entry process is high-level, the end-user is not confined to RMI as the Java snippets could be embedded in (say) MPI implemented in Java. In fact, there is nothing about the structure that confines data-entry to Java, as 'C' could be entered into the editor.

6 Worked example

To check our work, simulated results were correlated with an application [2], previously implemented on an eight module message-passing parallel machine, the Transtech Paramid. Recognition of handwritten postcodes with appropriate weightings can be split into three 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 in the final stage (UK postcodes tested could have 6 or 7 characters in the ratio 145:155), achieving 80% recognition accuracy. If the throughput constraint of 10 postcode/s or the maximum latency constraint of 8s were to be exceeded then the computer processing would not keep up with the mechanical conveyor belt which transports the mail items.

The static timings are set out in Table 1. Timing a set of 300 (1945) postcodes (characters) and then applying separately Kolmogorov-Smirnov and chi-squared tests [45], established that the distributions of processing times were approximately deterministic (and not Gaussian as had been supposed before tests), while the final stage, assuming random ordering in the input file set-up to test recognition accuracy, was matched by a Bernoulli distribution. In the original implementation, 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 best (all size six) cases. In Table 2, the worst-case estimates are compared with the implemented result with favourable accuracy. The 3:3:1 pipeline simulation is optimal, as had been suggested by preliminary static analysis. Note that one of the eight processor modules is reserved for feeding the test file to remove I/O dependency. Throughput is critical, while latency is well below the 8s requirement on the Paramid, though on earlier transputer-based machines latency was an issue. Though one might seek to apply a simulation capturing more of the computer system detail, experience has shown [46] that no greater accuracy necessarily results.

In the APTT simulation, varying the number of processors in the pipeline above and below the number in the Paramid, Table 4, established possible cost/performance tradeoffs and highlights the advantages of the simulation tool in allowing rapid and complete exploration of the design space of possible parallel solutions. The original buffer slot sizes were probably set too high as internal buffering was not found to be critical while interstage buffering could be reduced throughout to the postcode character size (seven slots).

When testing a real-time application it may be difficult to remove the effect of system-dependent I/O except by pre-loading a file. However, latency also arises if jobs are blocked on requesting entry to the pipeline. In order to judge the value of the simulation in that respect, after loading the file, assuming Poisson statistics for job arrival rates, a delay was artificially generated in the implemented version. In some operating conditions, latencies longer than 8s may arise due to the additional possibility of multiple arrivals while the pipeline is blocked.

7 Cross-architectural comparison

To allow the performance within APTT on one machine to be extrapolated to another we sought a simple but widely-recognised characterisation. A two parameter model of performance has now been applied to a variety of parallel architectures [47], though not apparently previously in a predictor tool. For example, in Fig. 3, which is a log-log plot, the Paramid reaches half its maximum bandwidth with messages of about 60 bytes (first parameter, established by linear regression) before reaching steady state (second parameter). In this case, the user need only know the message length and the target processor to project results.

Measurements on an individual Paramid processor, an i860, showed that a two parameter characterisation might be insufficient for computation as there was dependency on the computation kernel being performed, with additional cache effects evident. Fig. 4, showing results for four out of seventeen test kernels at full compiler optimisation, indicates two linear phases for some kernels where the vector length being computed stays within and steps outside the cache. However, it is not a difficult matter to store in a look-up-table the results for each machine and for each kernel. The user then selects a kernel, vector size, and processor to enable the performance tool to give a first-order approximation by means of scaling the computation times. This is likely to be more helpful for regular computations such as orthogonal transforms. An alternative characterisation is to use the computational intensity of the code, f, in units of flops/memory reference. Table 3 records steady-state performance (which is well below theoretically optimal performance), r_α^{ic} and r_α^{oc} being respectively in-cache and out-of-cache performance in units of Mflop/s.[3] The settings for compiler optimisation level three show a reverse trend to the expected increase in performance as intensity increases; compare level two in Table 3. However, we use the higher figures in order to make a fair comparison between the two processors.[4]

Applying the out-of-cache computational intensity test (level three setting for the Paramid), a Dec Alpha (21064 at 175 MHz) server was found to scale over the i860 by a factor of 3.0 for $f = 5$ with a `-fast` compiler setting. As this is a load-dependent measurement, the arithmetic mean of five selected results was taken. Table 4 records the projected timings if 21064s were to be substituted for i860s, otherwise keeping the system the same.[5]The longer out-of-cache test figures were chosen because the lower resolution clock would otherwise affect the accuracy[6] though in-cache timings indicated a larger scaling particularly for higher values of f, indicating an efficient memory hierarchy. Low-resolution software clocks may be a deterrent to the use of a processor in some hard real-time system, though not perhaps for soft real-time systems as herein.

Table 4 is intended as an enquiry into the projected performance of a processor, the DEC Alpha, which though general-purpose has also has been used for embedded applications such as BBN's 50 Gbps IP router. The projection should be treated with care as in itself this does not

[3]The out-of-cache measurements arise by using vector lengths designed to exceed the cache size and by causing a cache flush between tests.

[4]Note that at $f = 5$ out-of-cache performance is similar for both compiler options.

[5]This is not a practical possibility as the choice of processor and coprocessor is dictated by cost and compatibility, the i860 and transputer both being little-endian.

[6]Clock resolution was ~ 0.01s as opposed to ~ 0.001s on the Paramid, mean timing error $\pm 6.1\%$.

validate the model. However, when the postcode recognition application was previously ported from running on 32 transputers on the Meiko CS-2, to the Paramid [2] using i860s with transputer co-processors, the mean pipeline traversal latency reduced from 3.20 s to 2.23 s, with throughput increasing from 2 to 11.39 postcode/s. This matched expectations before the port was made that the transfer would meet required specification goals based on the relative computational performance of the processors.

Once the characteristics within APTT have been determined for one machine then an interesting possibility is to build on the results to make run-time decisions in the event of the need for dynamic reconfiguration of resources. For example, in mobile settings bandwidth (see below, Section 8) may alter or mobile stations may be disconnected. As heterogeneous processing devices will no doubt be brought together the need for a simple performance database is heightened. Demand-based data farming is a scheduling method particularly suited to allocation of work when resources are dynamically changing, with heterogeneous hardware forming the farm, though there may be a need to cut off allocated work not processed within a given time threshold. Therefore, future work will be to adapt the insights gained from APTT to this purpose.

8 Adapting to mobile networked computation

We have given some thought to inception and synchronisation of a networked computation. One plan is to start with one master farmer module which is always present and a pool of farmer and worker modules which can be chained together. As currently implemented, the first farm is formed dynamically to save time. When processed work is ready to go on to another pipeline stage, the first farmer inspects for farmer multicasts and selects a ready farmer. A more flexible arrangement is shown in Fig. 12. The pipeline manager (PM) is contacted from a pool of RMI elements on a known port/multicast address. The PM then ascribes the 'gender' (either farmer or worker) to an RMI element. Having set-up the pipeline, Fig 13, the PM reflects its behaviour to become a monitor. Each instance of a farmer is characterised by the type of data manager: at the start and end of a pipeline a file data manager is assigned, while in intermediate stages a message reception and relay manager is needed.

RMI can pass objects as parameters to a server provided that the objects are serialisable, allowing in effect PPF to dynamically load code onto a worker, though there are issues of casting. For the purposes of automatic configuration, objects designated as remote can be passed from

server to client though in this case only the stub object arrives. [7] In principal, an application can be assembled as a pipeline of data-farms from a central source, or the application can be extended by passing stubs (having first sent naming information). A further issue to address is security, as code distribution is much like virus distribution.

The Jini model for distributed computation [49] would now appear to provide a software technology to support the mobile computation proposed in this paper. Jini is layered on RMI, but enhances the RMI nameserver with an automatic and generic way of discovering and forming an *ad hoc* computation network. Through leasing, Jini also enables such background federations of devices to be dissolved in due course. Jini provides an implementation repository which can be accessed via an `http` server, maybe through a WAP portal but not directly by RMI. The BlueTooth radio standard [50] is a low-level way of forming device federations, especially by short-range connections to the fixed network.

9 Conclusion

APTT was constructed in the Java programming language with the intention of avoiding the portability trap between PCs and workstations. In fact, the port is a trivial exercise. An initial concern was the slowness of the graphical display but these concerns have largely evaporated with the advent of: Just-in-Time (JIT) and 'hotspot' compilers; the increase in clock speed of the Pentium family of microprocessors; and our experience of the Java graphical class libraries [51]. The predictor and analysis tool share a balanced layout, with facilities to control the speed and format of the display. The qualitative indicators make appropriate use of the graphical medium, while second tier quantitative information is also available. Though the predictor is intended as a first level selector of pipeline structure and processor configuration the worked example showed agreement to 10% with live runs on a distributed-memory multicomputer. The third element of APTT is the most experimental, given that a series of programmers' templates have already been implemented. Serendipitously, RMI appeared along with Java graphical libraries with flexible facilities which are quite suitable for the likely future processing environment.

[7] As the stub object is strictly constrained, it becomes necessary to provide a customised security manager.

Acknowledgement

Software construction of APTT by N. Sarvan, R. Durrant, and G. Sweeney is gratefully acknowledged.

We also acknowledge helpful comments by the anonymous referees.

This work was carried out under EPSRC research contract GR/K40277 'Parallel software tools for embedded signal-processing applications' as part of the EPSRC Portable Software Tools for Parallel Architectures directed programme.

References

[1] A. C. Downton. Generalised approach to parallelising image sequence coding algorithms. *IEE Proceedings Part I (Vision, Image and Signal Processing)*, 141(6):438–445, 1994.

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

[3] M. Fleury, A. C. Downton, and A. F. Clark. Karhünen-Loève transform: An exercise in simple image-processing parallel pipelines. In C. Lengauer and M. Griebl, editors, *EuroPar'97*, pages 815–819. Springer, Berlin, 1997. Lecture Notes in Computer Science 1300.

[4] J. Fleischmann and K. Buchenreider. Integrated engineering. *IEEE Computer*, 32(2):116–119, 1999.

[5] S-Y Lee and J. K. Aggarwal. A system design scheduling strategy for parallel image processing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2):194–204, 1990.

[6] 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.

[7] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology*, pages 85–95, August 1993.

[8] A. S. Wagner, H. V. Skreekantaswamy, and S. T. Chanson. Performance models for the processor farm paradigm. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):475–489, May 1997.

[9] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, London, 1998.

[10] A. S. Grimshaw, W. T. Strayer, and P. Narayan. Dynamic, object-oriented parallel processing. *IEEE Computer*, pages 33–46, May 1993.

[11] M. Fleury, H. P. Sava, A. C. Downton, and A. F. Clark. Designing and instrumenting a software template for embedded parallel systems. In *UK PARALLEL '96*, pages 163–180. Springer, London, 1996.

[12] M. Fleury, N. Sarvan, A. C. Downton, and A. F. Clark. Methodology and tools for system analysis of parallel pipelines. *Concurrency: Practice and Experience*, 11(8):1–16, 1999.

[13] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. *ACM SIGPLAN notices*, 17(6):120–126, June 1982.

[14] Pure Software Inc., 1309 South Mary Ave., Sunnyval, CA. *Quantify User's Guide*, 1992.

[15] C. Ponder and R. Fateman. Inaccuracies in program profilers. *Software–Practice and Experience*, 18(5):459–467, May 1988.

[16] M. Fleury, A. C. Downton, and A. F. Clark. Parallel structure in an integrated speech-recognition network. In *Euro-Par'99 Parallel Processing*, pages 995–1004. Springer, Berlin, 1999.

[17] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, October 1985.

[18] S. Madala and J. B. Sinclair. Performance of synchronous parallel algorithms with regular structures. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):105–116, January 1991.

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

[20] P. Worley. A new PICL trace file format. Technical report, Oak Ridge National Laboratory, 1992. Report no. ORNL/TM-12125.

[21] J. Hou and W. Wolf. Presynthesis partitioning for hardware/software cosynthesis. *IEE Proceedings Part E (Computers and Digital Techniques)*, 145(3):197–202, May 1998.

[22] M. Edwards and J. Forrest. A practical hardware architecture to support software acceleration. *Microprocessors and Microsystems*, 20:167–174, 1997.

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

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

[25] M. Fleury, A. C. Downton, and A. F. Clark. Constructing generic data-farm templates. *Concurrency: Practice and Experience*, 11(9):1–20, 1999.

[26] G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT, Cambridge, MA, 1986.

[27] G. Agha, S. Frolund, Kim W. Y., R. Panwar, A. Patterson, and D. Sturnman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Computer*, pages 3–13, May 1993.

[28] Y. Wu and T. G. Lewis. Parallelism encapsulation in C++. In *International Conference on Parallel Processing*, volume II, pages 35–42. Pennsylvamia State University, 1990.

[29] P. E. Lauer, P. R. Torrigiani, and M. W. Shields. COSY – a system specification language based on paths and expressions. *Acta Informatica*, 12:109–158, 1979.

[30] G. R. Andrews, R. A. Olsson, M. Coggin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[31] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, 1988.

[32] P. Vishnubhotia. Synchronization and scheduling in ALPS objects. In $8^{\underline{th}}$ *International Conference on Distributed Computing Systems*, pages 256–264, June 1988.

[33] D. B. Skillicorn. Structured parallel computation using categorical data types. In A. Y. Zomaya, editor, *Parallel Computing: Paradigms and Applications*, pages 145–161. International Thomson Computer Press, 1996.

[34] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[35] A. Agarwal. Performance tradoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[36] H. P. Sava, M. Fleury, A. C. Downton, and A. F. Clark. A case study in pipeline processing farming: Parallelising the H.263 encoder. In *UK PARALLEL '96*, pages 196–206. Springer, London, 1996.

[37] H. Takagi. Queueing analysis of polling models. *ACM Computing Surveys*, 20:5–28, 1988.

[38] M. Fleury and A. F. Clark. Performance prediction for parallel reconfigurable low-level image processing. In *International Conference on Pattern Recognition*, volume 3, pages 349–351. IEEE, 1994.

[39] A. Yonezawa, editor. *ABCL: An Object-oriented Concurrent System*. MIT, Cambridge, MA, 1990.

[40] J. Ferber and P. Carle. Actors and agents as reflective concurrent objects: A Mering IV perspective. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4):1420–1436, 1991.

[41] T. G. Lewis. Code generators. *IEEE Software*, pages 67–70, May 1990.

[42] S. Ahmed, N. Carriero, and D. Gelernter. The Linda program builder. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 71–87. Pitman, London, 1991.

[43] Scriptics Corporation, 2590 Coast Ave., Mountain View, CA. *SpecTcl Manual*, 1999. Available from http://www.scriptics/spectcl.

[44] K. Konstandides and J. R. Rasure. The Khoros software developement environment for image and signal processing. *IEEE Transactions on Image Processing*, 3(3):243–252, 1994.

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

[46] T. Delaitre, M. J. Zemerly, P. Vekariya, G. R. Justo, J. Bourgeois, F. Schinkmann, F. Spies, S. Randoux, and S. C. Winter. EDPEPPS: A toolset for the design and performance evaluation of parallel applications. In D. Pritchard and J. Reeve, editors, *Euro-Par'98 Parallel Processing*, pages 113–135. Springer, Berlin, 1998. Lecture Notes in Computer Science 1470.

[47] R. W. Hockney. *The Science of Computer Benchmarking*. SIAM, Philadelphia, 1996.

[48] M. Fleury, A. C. Downton, and A. F. Clark. Scheduling schemes for data farming. *IEE Proceedings Part E (Computers and Digital Techniques)*, 146(5):227–234, 1999.

[49] W. K. Edwards. *Core Jini*. Prentice Hall, Upper Saddle River, NJ, 1999.

[50] J. C. Haartsen. The Bluetooth radio system. *IEEE Personal Communications*, 7(1):28–36, 2000.

[51] M. Sulieman, C. H. Ooi, and M. Fleury. Parallel pipeline to ATM: Graphical simulation techniques. In 15th *UK Performance Engineering Workshop, UKPEW'99*, 1999.

Table 1: -Input parameters timed on target processor, an i860, (a) 6- (b) 7-character postcode

stage	job time(s)	distribution	max. data transfer (bytes)
1	0.028	constant	2119
2	0.036	constant	40
3(a)	0.0045	bi-modal	112
3(b)	0.019	bi-modal	112

Table 2: - Comparison of simulated with timed results

worker ratio	run-time (s)	throughput postcodes/s	mean (s) latency	max. (s) latency
		simulated		
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
		implemented		
3:3:1	24.2	12.4	1.2	1.6
2:3:2	27.7	10.8	0.6	0.7
1:4:2	54.6	5.5	0.3	0.4
2:2:3	35.3	8.5	1.0	1.0
3:2:2	35.1	8.6	1.0	1.0

Table 3: - Paramid computational performance

compiler option 2								
f	1	2	4	5	6	8	9	10
r_{α}^{ic}	10.0	12.5	14.3	14.7	15.3	15.3	15.5	15.6
r_{alpha}^{oc}	9.2	12.0	14.2	14.5	15.1	15.2	15.2	15.3
compiler option 3								
f	1	2	4	5	6	8	9	10
r_{α}^{ic}	25.6	18.3	15.9	15.6	15.4	15.1	13.2	13.3
r_{α}^{oc}	18.1	16.1	15.7	14.8	14.7	14.9	15.0	13.2

Table 4: - Simulated results for a variety of pipelines

	Paramid		Dec Alpha (21064)	
pipeline worker ratio	run-time (s)	thruput pcodes/s	run-time (s)	thruput pcode/s
3:2:1	35.1	8.6	11.9	23.3
2:3:1	27.6	11.0	9.4	29.8
4:3:1	24.3	12.4	8.1	34.5
3:4:1	23.9	12.6	8.3	33.3
3:3:2	23.5	12.8	8.0	34.7
3:4:2	18.6	16.1	6.3	43.9
4:4:2	17.7	17.0	6.0	46.3
2:5:1	27.3	11.0	9.1	30.0

List of Figures

1 PPF design pattern 23

2 Design cycle 23

3 Point-to-point communication performance for the Paramid 24

4 Selected computation performance for the Paramid 24

5 APTT data-entry 'wizard' 25

6 APTT predictor window 25

7 APTT scaling window 26

8 APTT analyser window 26

9 The Actor model 27

10 Java RMI Worker Module 27

11 APTT template window 28

12 Inchoate system 28

13 Dual data-farm pipeline after configuration 29

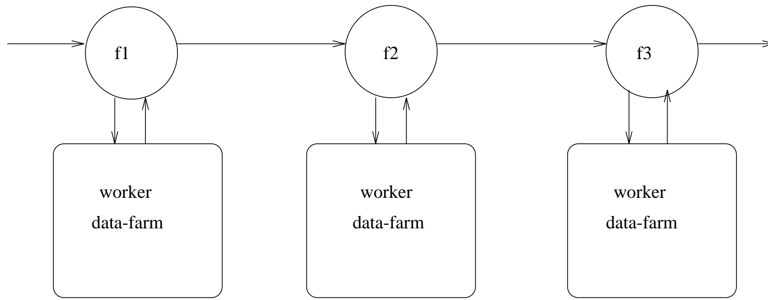

Figure 1: PPF design pattern

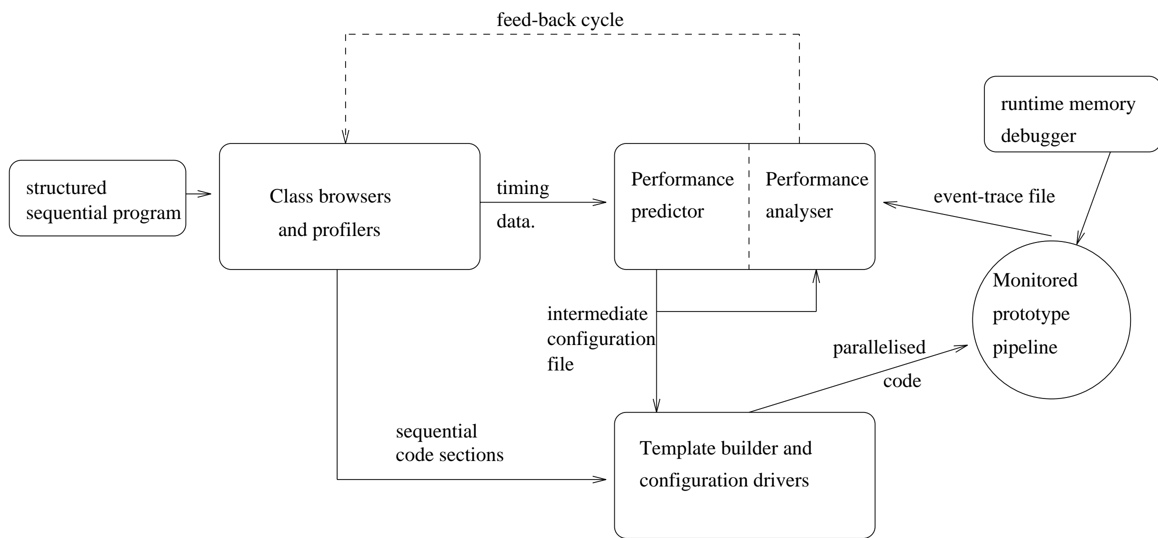

Figure 2: Design cycle

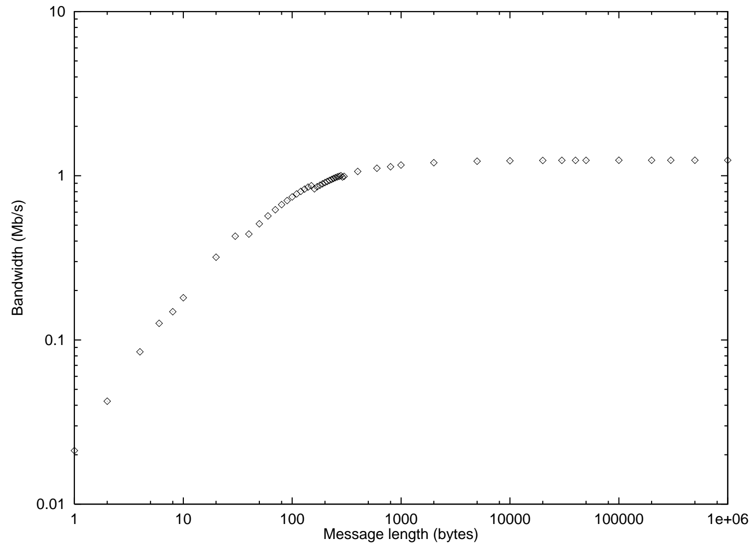

Figure 3: Point-to-point communication performance for the Paramid

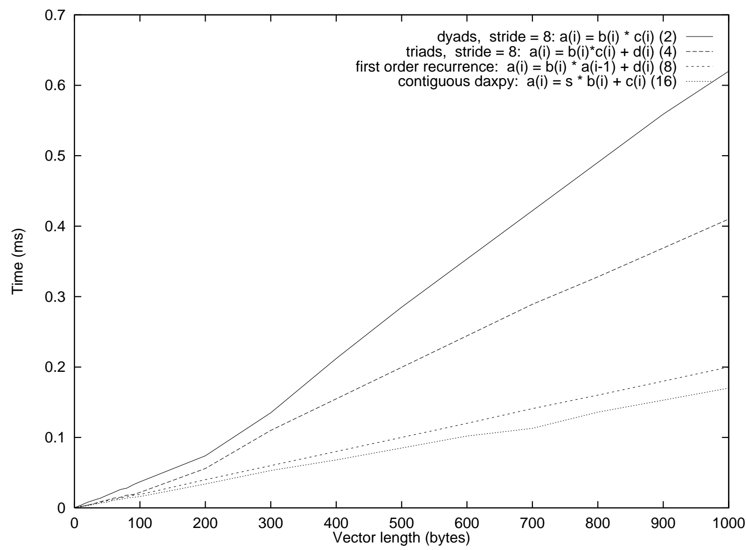

Figure 4: Selected computation performance for the Paramid

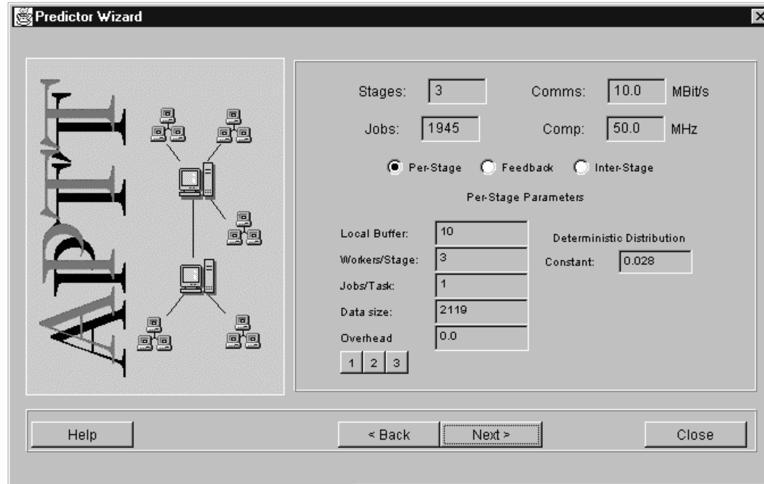

Figure 5: APTT data-entry 'wizard'

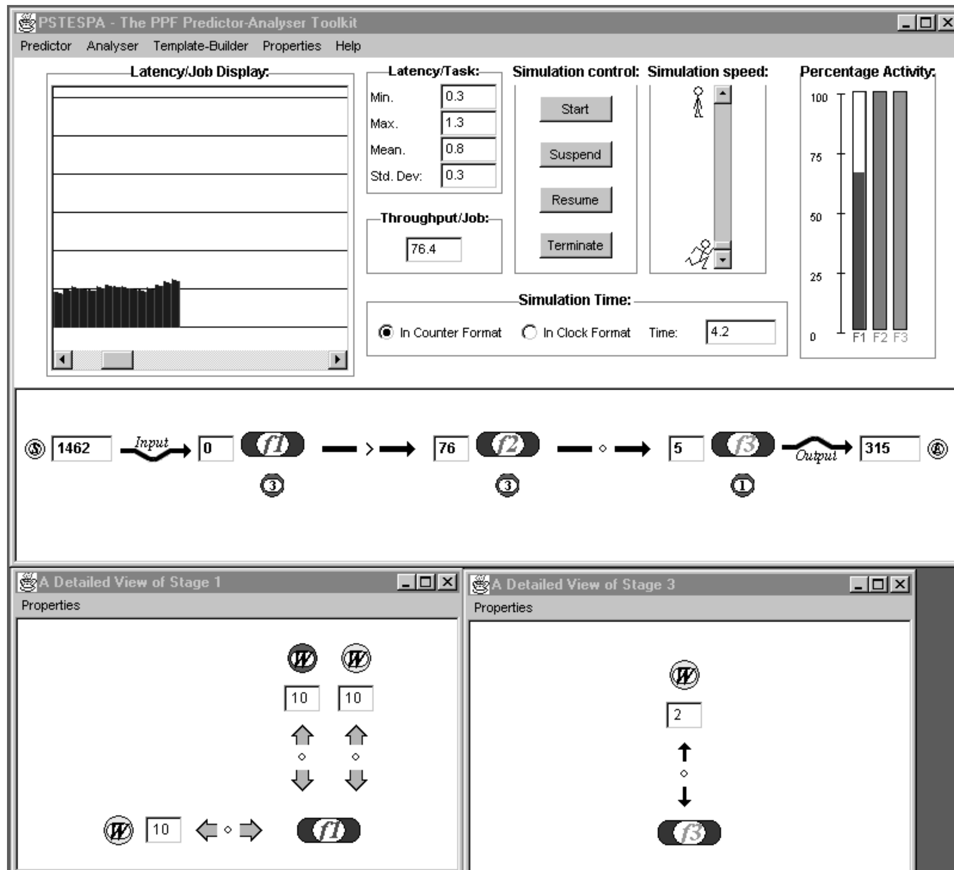

Figure 6: APTT predictor window

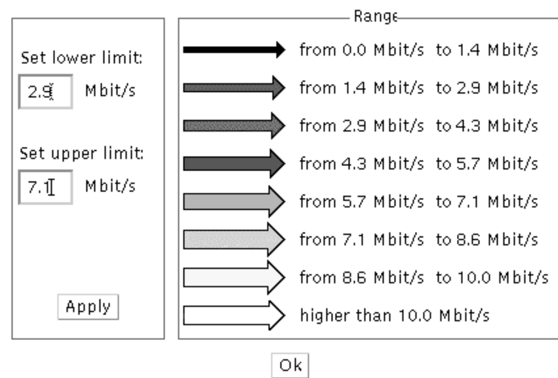

Figure 7: APTT scaling window

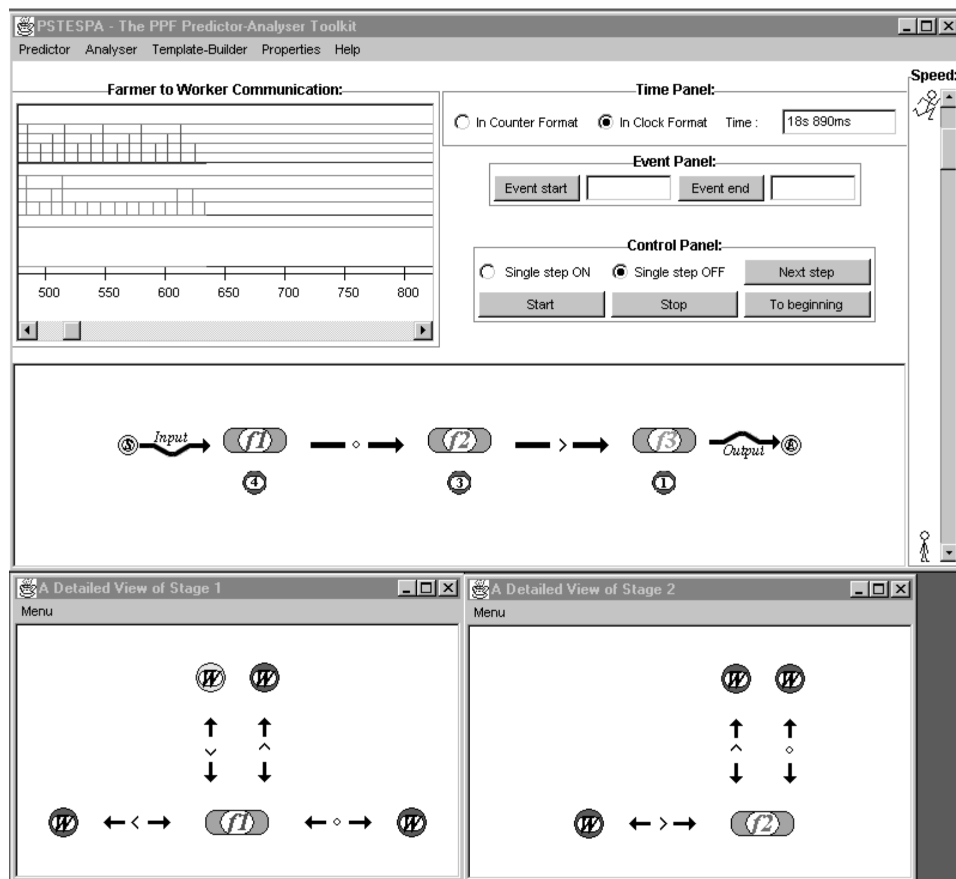

Figure 8: APTT analyser window

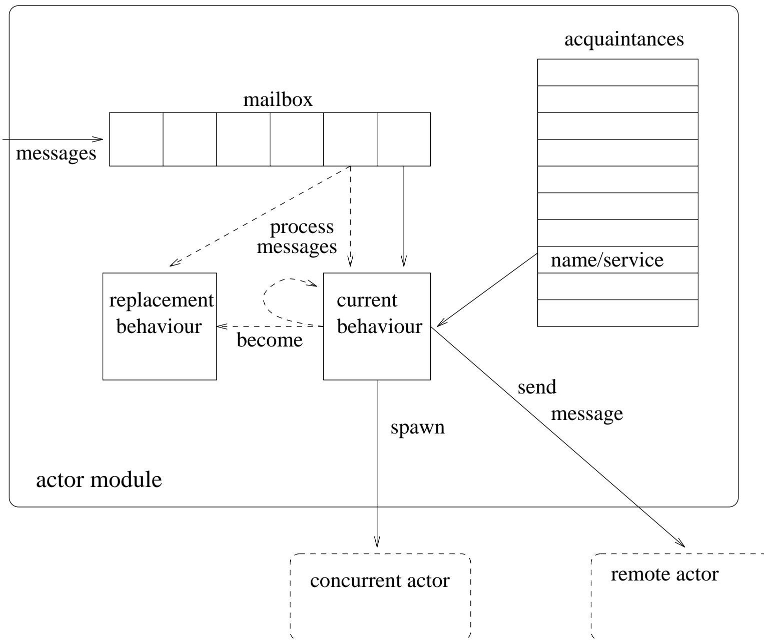

Figure 9: The Actor model

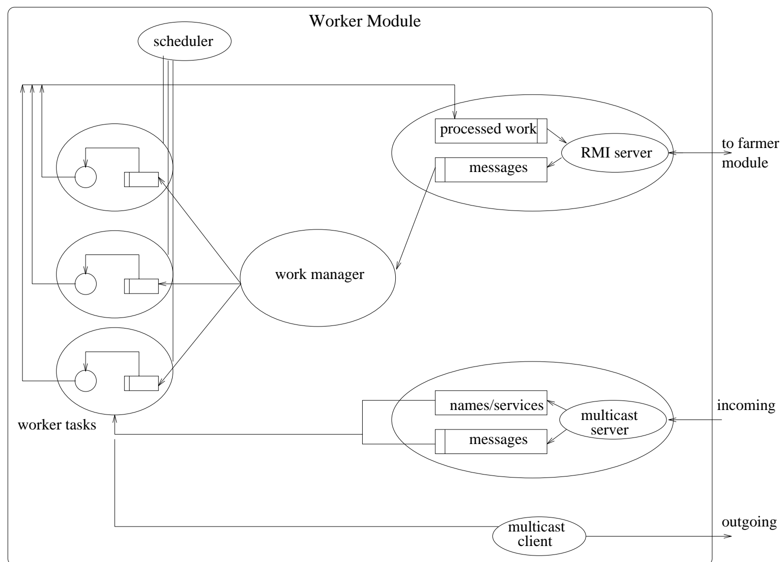

Figure 10: Java RMI Worker Module

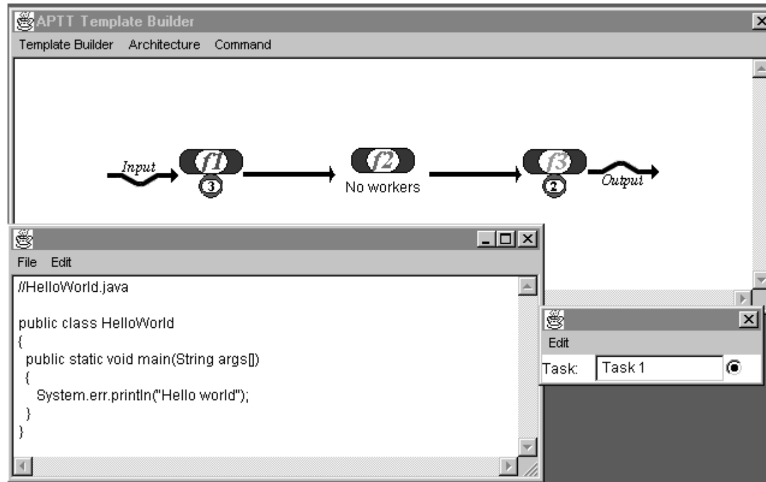

Figure 11: APTT template window

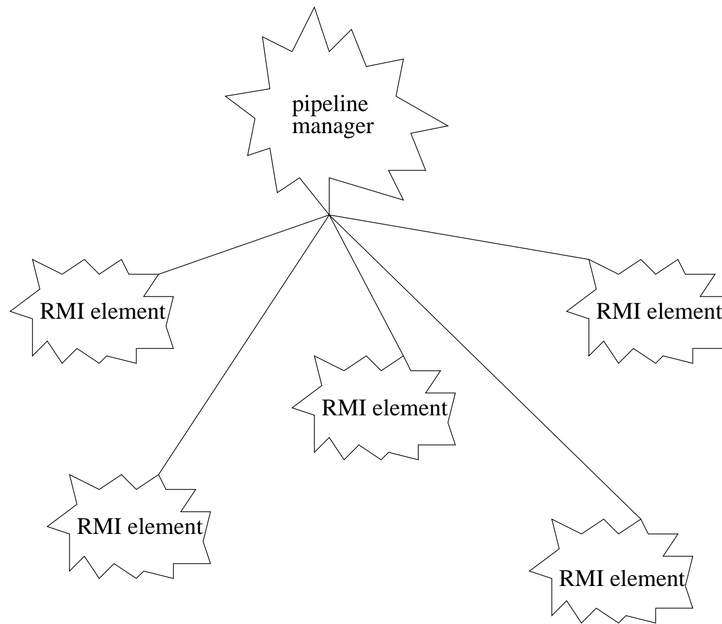

Figure 12: Inchoate system

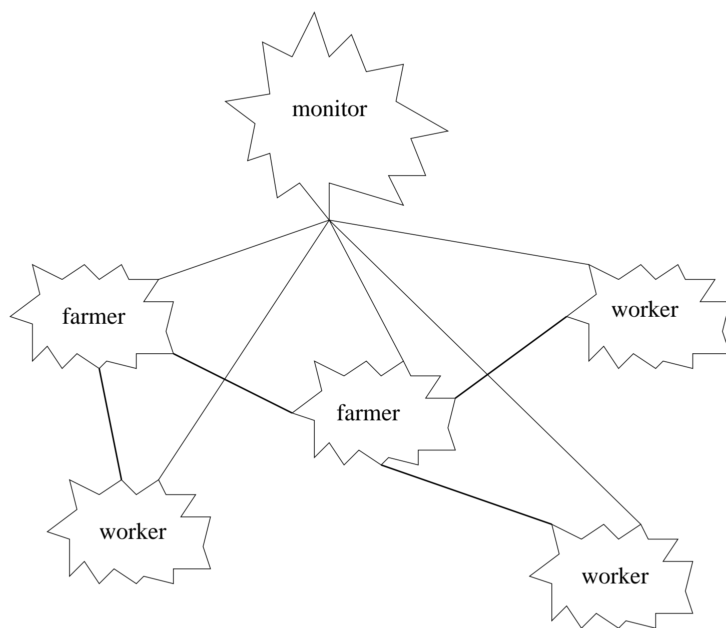

Figure 13: Dual data-farm pipeline after configuration