Communicating Process Architectures – 2002
James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)
IOS Press, 2002

Performance Analysis and Behaviour Tuning for Optimisation of Communicating Systems

Mark GREEN and Ali E. ABDALLAH
Centre for Applied Formal Methods, South Bank Technopark, 103 Borough Road, London, SE1 0AA, UK.

Abstract. Improving performance is the main driving force behind the use of parallel systems. Models for performance evaluation and techniques for performance optimisation are crucial for effectively exploiting the computational power of parallel systems. This paper focuses on methods for evaluating the performance of parallel applications built from components using the software architecture methodology. Minor differences in the low-level behaviour of functionally equivalent processing elements can have a dramatic effect upon the performance of the overall system; this confounds attempts to predict performance at any step prior to implementation. VisualNets is a tool supporting the construction and graphical manipulation of interacting systems built from components. It makes use of specifications in the formal method *CSP*, which enables the relevant component behaviours and the linkage between components to be concisely described. The tool allows the behaviour of the whole system over time, and the patterns of interaction between the components, to be visualised through graphical animation. The graphical display produced facilitates the analysis and evaluation of performance, and highlights areas where performance could be improved via better utilisation of parallel resources. VisualNets also allows the timing properties of the components and of the architecture that underlies them to be changed, to represent different component implementations or platform configurations. A case study, based on the dual pipeline architecture, is presented to show how the graphical animation capability of VisualNets can be used, firstly to evaluate performance, and secondly to guide the development of more efficient versions of the parallel system.

1 Introduction

The primary objective of parallelising many applications is improved performance; the primary tradeoff is the cost of additional computing resources required to support parallel execution. Thus an important goal when designing a parallel system is to ensure that the resources of the parallel platform on which the system executes are utilised efficiently, so that the performance increase obtained makes the tradeoff worthwhile. Managing the patterns of interaction between parallel components such that improved performance is achieved is one of the most fundamental problems in the field. It involves issues from high-level software architecture principles right down to low-level implementation details and hardware characteristics.

The emerging software architecture paradigm [1] emphasises the construction of parallel applications from reusable components. A number of frequently used parallel patterns, such as the pipeline and process farm, can be considered architectures [2] which define specific structures for combining interacting components. Parallelism can be introduced to a system by selecting a suitable architecture, dividing the overall task of the system into subtasks, and assigning these subtasks to processing components within the architecture.

However, no matter how much care is taken in the selection of the architecture and the partitioning of the task, no guarantee of high performance (the best possible utilisation of the available computing resources) can be obtained. There are a large number of lower level

issues which will impinge significantly on performance; even minor aspects in the implementation of the individual processing components may have a substantial effect on the overall performance of the system.

Thus, having performed the initial design of the parallel system, it is still necessary to tune the behaviour of the components involved in order to ensure that high performance is achieved. In this paper the Communicating Sequential Processes (*CSP*) notation [3] is used to concisely capture the behaviours and interfaces of components within a parallel architecture. The *VisualNets* tool [4, 5] then enables animation and performance profiling to be carried out directly on the abstract *CSP* description, without the need for lower level implementation. This paper demonstrates the effectiveness of this method in tuning the components of a parallel architecture to maximise performance.

Although a large number of tools allow visual performance profiling of parallel systems [6, 7, 8, 9, 10], they mainly operate on implementations as opposed to design descriptions. This inhibits their use until a late stage in the development cycle. Although these tools facilitate the detection of poor performance, they generally do not provide facilities for tracing the behaviour that causes this poor performance. This is inevitably a difficult task at the implementation level. By basing its profiling on a design description rather than an implementation, *VisualNets* can precisely trace the behaviour of all components in the system and thus make it easier for the developer to trace the source of inefficiency. Furthermore amending the description to rectify the inefficiency will be a simpler process for the developer than amending an implemented system in the same way.

2 Background

2.1 Communicating Sequential Processes

The formal method of Communicating Sequential Processes (*CSP*), first specified by Hoare [3], describes concurrent processes in terms of sequences of synchronisation events and channel communications.

CSP is an extremely effective notation for representing the architecture of parallel systems, since the architectural style imposed by *CSP* is extremely flexible. The only stipulation is that the connecting elements must either be message-passing channels or multiprocess barrier synchronisations. *CSP* has no notation for formally specifying the properties of data elements, meaning that it is more suited for processing-based analysis; behaviour tuning falls into this category. Processing elements are defined by combinations of primitive operators that represent sequence, choice, communication, parallelism, and similar.

This use of primitive concurrency operators simplifies the process of making the minor behaviour changes (such as varying the order in which communications are carried out) which are required in performance tuning. In implemented code, making these minor changes may be more difficult as several statements may be involved in preparing and performing the communication, some of which may have side effects which will cause problems if their sequence is changed.

The fact that standard *CSP* does not consider the implementation of operators can be valuable in performing design-level behaviour tuning, as some designed-in properties of behaviour may reduce performance regardless of platform. However, if the hardware on which a system will run is specified in advance, it is necessary to allow for properties related to it to completely optimise the system. An extension to *CSP* defined by Schneider [11] allows timing properties of a system to be represented. These can be used to model the performance properties of the hardware platform or parallelism library on which the system will run. Thus once untimed *CSP* has been used to model the design of a system, timed *CSP* can be use to

predict its behaviour on a particular platform without incurring the costs of implementation. Once the specification has been completed, producing an implementation based on it is a relatively simple process.

2.2 VisualNets

The *VisualNets* tool, developed at Reading and South Bank [4, 5], allows systems specified in both untimed and timed *CSP* to be visually manipulated and animated. For performance evaluation, the most important function it provides is the production of a *timing diagram* which visualises the progress of the network over time and can also provide indications of where performance loss is taking place. This paper focuses on the use of the timing diagram to diagnose performance problems. *VisualNets* can also produce a structure visualisation which corresponds to a "processing view" of the system as defined in [1], and provides a visual display of component status which is updated as animation proceeds; where a system has a large number of components, modular abstraction can be employed to simplify the diagram by combining several components into a subsystem which appears as a single element on the diagram. The tool will detect any deadlocks that arise during animation runs.

Additionally, a feature under development in *VisualNets* allows untimed *CSP* specifications to be transformed into timed *CSP* specifications based on a seperate model of a hardware platform. This enables a parallel algorithm written in untimed *CSP* to be profiled via simulation on a number of different hardware platforms without any need to develop multiple implementations or to modify the specification to reflect the timing properties of the platforms. At present, this feature is implemented although the method used for modelling the hardware is quite primitive.

VisualNets's implementation consists of two parts: a front end module written in Java [12, 13], which produces the graphical visualisations and tables and provides the user interface; and a back end module in the functional language Haskell [14], which actually performs the animation of the user's specification and supplies the results to the front end through an interface developed as part of the research [15]. The tool is platform independant, and can operate on both Windows and Unix environments. The prototype tool is not distributed at this time as it depends on a commercial graph layout system which cannot be realistically licensed for academic distribution; it is hoped that this dependancy will shortly be removed.

3 Case Study

To illustrate the need for behaviour tuning, and the use of *CSP* and the *VisualNets* tool in performing it, we consider a system implemented with a standard parallel architecture: a dual pipeline. The system in question is intended to perform long multiplication in any base; each process in the system deals with a single digit in the value to be multiplied.

The multiplier is "hard wired" into the system; each process is assigned one digit of the multiplier. Less significant digits appear earlier in the pipeline flow. The multiplicand is input to the pipeline one digit at a time, with the least significant digit first, and with a stream of zeroes following it (effectively leading zeroes). Since the structure is a dual pipeline, the result is also returned from the process at the start of the pipeline, in the same order and also followed by a stream of zeroes which continues as long as the input does.

The data elements involved are single digits of the multiplier, multiplicand, result, and intermediate values; the connection elements are message-passing channels.

The processing elements of the pipeline are nodes. With the exception of the end nodes, each node X is connected to the node N holding the multiplier digit one place more significant than the one held by X, and to the node P which holds the digit one place less significant than

the one held by *N*. *X* will have an input from *P* and an output to *N*; this forms the *input* pipeline, the direction of which will be termed "down". *X* will also have an input from *N* and an output to *P*; this forms the *result* pipeline, the direction of which will be termed "up". The first node of the pipeline substitutes for *P* the input loading and output collector processes; the final node has no *N* and therefore has only the two channels which refer to *P*.

The functional specification of each node is quite simple: it must input (from the input pipeline) a digit of the multiplier, and then (at some point before the next digit is input) return a digit of the result on the result pipeline. For all but the first digit produced, each node will need to allow for the carry values from its own prior multiplications, and the result of the multiplication of the remaining digits. The carry can be tracked within the node. To obtain the result for the remaining digits, each node must pass the digits it recieves down the input pipeline to the next node, which will perform the multiplication; at some point before sending another digit to the input pipeline, the node must obtain the result from the next node via the result pipeline for use in its own calculations.

The exception to this rule is the last node in the pipeline, which holds the most significant digit. Since there are no more digits to multiply, it needs only to keep track of the internal carry.

3.1 Refinement to CSP

The first step is to refine the functional specification into *CSP* notation. The notation actually used is CSP_F, the machine readable form of *CSP* accepted by the *VisualNets* tool. In the CSP_F description below, the variable *k* holds the number of the node in the network, *a* is the digit of the multiplier assigned to the cell, and *z* is used to track the internal carry and should be initialised to 0 when the process is instantiated. *msize* is the number of digits in the multiplier and thus the number of nodes in the pipeline. `x!a` indicates that message *a* is output on channel *x*, and `x?a` indicates that a message is read from channel *x* and placed in variable *a*. `->` indicates prefixing (sequence), and `A <| x |> B` is a guarded choice structure which behaves as A if *x* is true, and B otherwise.

```
NODE a k z = ( in1?(x :: Int) -> out1!(((a*x)+z) `mod` base)
               -> NODE a k (((a * x)+z) `div` base))
               <| k == (msize - 1) |>
               ( in1?(x :: Int) -> out2!x ->
               in2?(y :: Int) -> out1!((((a*x)+z)`mod` base))
               -> NODE a k (y+(((a*x) + z) `div` base)))
```

This refines the functional description given above. Each node that is not the end of the pipeline (ie, where *k*, the node's index, is not equal to $msize - 1$, the highest index) first receives a digit of the multiplicand (on channel $in1$), and passes it down the pipeline (on channel $out2$). It collects the result from the multiplication of the remaining digits (input from channel $in2$ and stored in *y*), then performs its own multiplication and passes its own result up, updating its own stored carry value based on the result obtained from the rest of the pipeline. The last node in the pipeline only retrieves results and performs its own multiplication; there is no further pipeline to pass values down to. Note that the structure of the specification is very strongly based on the communicating behaviour of the process. All of the computation performed by the process is encapsulated by the expressions for output values and guards. (For processes that perform more complex computation, *CSP* includes an assignment operator that can be used to represent purely computational processing steps.)

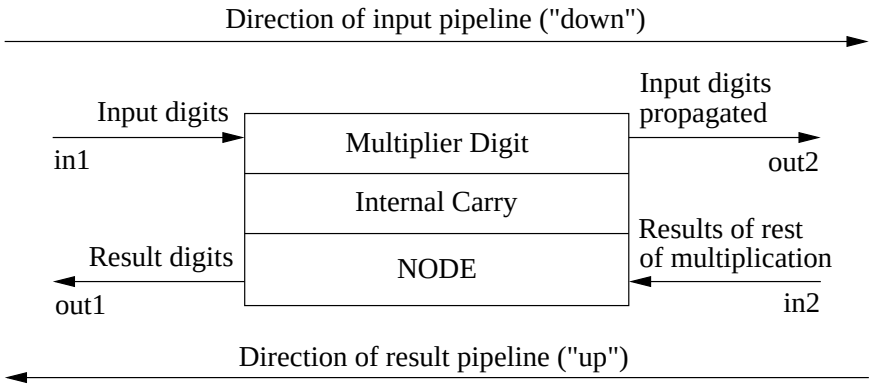

Figure 1: A single node of the multiplication pipeline.

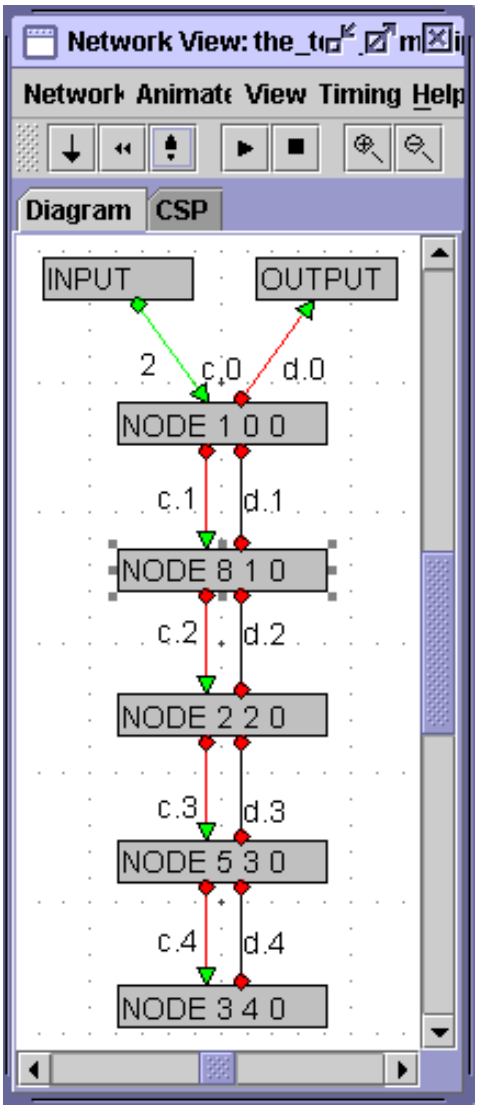

Figure 2: Structure visualisation of the Dual Pipeline Multiply.

VisualNets can be used to construct a pipeline of these processes, and to add additional processes representing the input and output ports. The structure visualisation of the network produced is shown in figure 2.

Each box on the visualisation represents a process - in this case a cell of the pipeline. The lines represent channels connecting the processes; the indicators at the end of each line are colour coded to represent the status of each process with regard to that channel. A red indicator means that the process is not offering communication; a green indicator means that it is doing so. Where a communication is offered by one of the processes (even if no cooperation is being received), an arrowhead is added to indicate the direction of the communication. The visualisation clearly shows the dual pipeline structure, with each node having one channel to the prior node and one to the subsequent node. The multiplier in this case is 35281, visible distributed through the parameters of the nodes.

When the network is animated, the behaviour of the network is as shown in the timing diagram in figure 3. The multiplicand, 49152, is input to the network and the solution to the multiplication, 1734131712, is shown as the output on channel $d.0$.

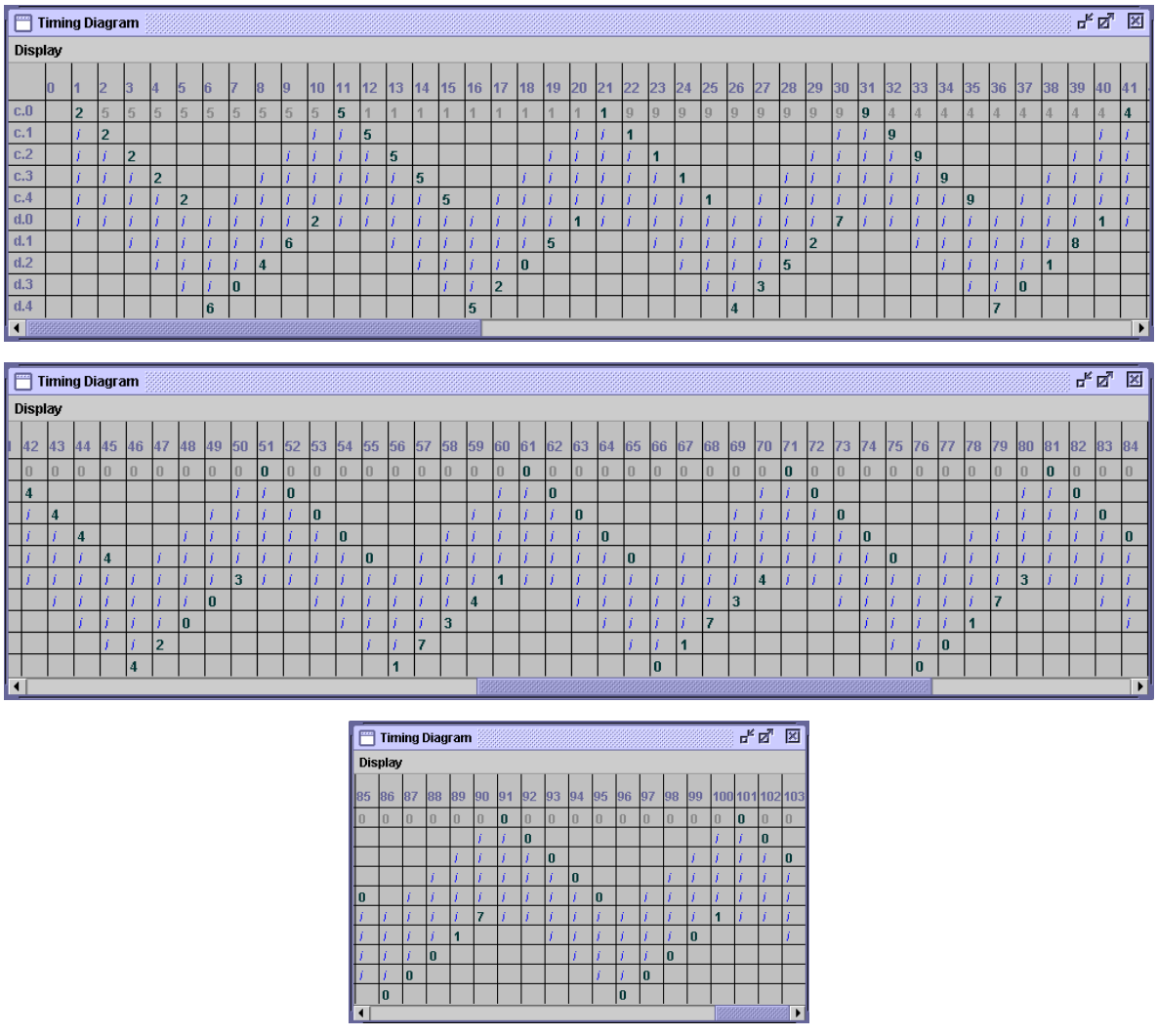

Figure 3: Timings for the first version of the Dual Pipeline Multiply.

The left hand side of the timing diagram shows the channels in the network; the top scale indicates the passage of time. A value in bold type indicates a successfully communicated value. Since no timings were specified for this system, *VisualNets* has applied the default timing model, in which computation takes zero (or negligable) time and any communication takes one time step.

The timing diagram also shows blocked communications. A blocked communication occurs whenever a process is attempting to communicate but is unable to do so as the other process involved is not cooperating. A value in faded type indicates that an attempt to output that value was blocked as the other process was not inputting; an italicised letter I indicates that an attempt to recieve input on the appropriate channel was blocked as the other process was not outputting anything. This blockage information is useful because it shows areas where performance loss is potentially being caused by difficulties in the concurrent design, as opposed to delays mandated by the timing model (ie, time taken by hardware) or caused by computation. This information can be removed from the timing diagram if the user wishes.

The total time passing between successful communications on a channel can be read as the number of spaces between boldfaced values; the delay imposed on a given communication by blockage within the system can be read as the number of faded values or italicised i's preceding the bold value representing the communication in question.

The timing diagram shows that the performance of this system is extremely bad; the last digit of the result is not sent until time step 100. The timing diagram further shows that the parallel properties of the architecture are being completely suppressed; only one process is ever engaged in communication at once. This has occured in spite of the fact that all of the processes are behaving in accordance with the functional specification, and in spite of the absence of any significant delays in the default timing model. In fact, it is likely that any system whose implemention behaved in the manner described by the *CSP* refinement would exhibit this performance problem regardless of the hardware on which it executed.

However, the failures information in the timing diagram informs the user that this performance problem may be rectifiable. The long string of failed inputs before each process indicates that time is being wasted. If these did not exist, then the time would be being spent on other purposes (such as computation) and it may genuinely not be possible to speed the network's operation up.

3.2 Performance tuning

By examining the timing diagram it can be seen that the delays tend to occur in between a node sending a value to be multiplied up the input pipeline and retrieving the result. Other tasks could be accomplished during this time. Since the multiplication result returned from the remainder of the pipeline for the current digit affects only the *future* digits of the result (the current result digit is not affected), it would be entirely possible for the node to send its result digit up the pipeline *before* recieving the result from the rest of the pipeline, provided it still did receieve the result from the rest of the pipeline before returning the result for any further digits.

This can be represented in the specification as follows:

```
NODE a k z = (in1?(x :: Int) -> out1!(((a *x)+z) `mod` base)
              -> NODE a k (((a * x)+z) `div` base))
              <| k == (msize - 1) |>
              ( in1?(x :: Int) -> out2!x ->
              out1!(((a*x)+z) `mod` base) -> in2?(y :: Int)
              -> NODE a k (y+(((a * x) + z) `div` base)))
```

The specification is the same as the previous one, except that the last two steps of behaviour have been reversed. Since no change has been made to the architecture, the structure visualisation of the system remains the same, but animating the system to produce a timing diagram shows radically altered performance. The timing diagram produced when animating this version of *NODE* is shown in Figure 4.

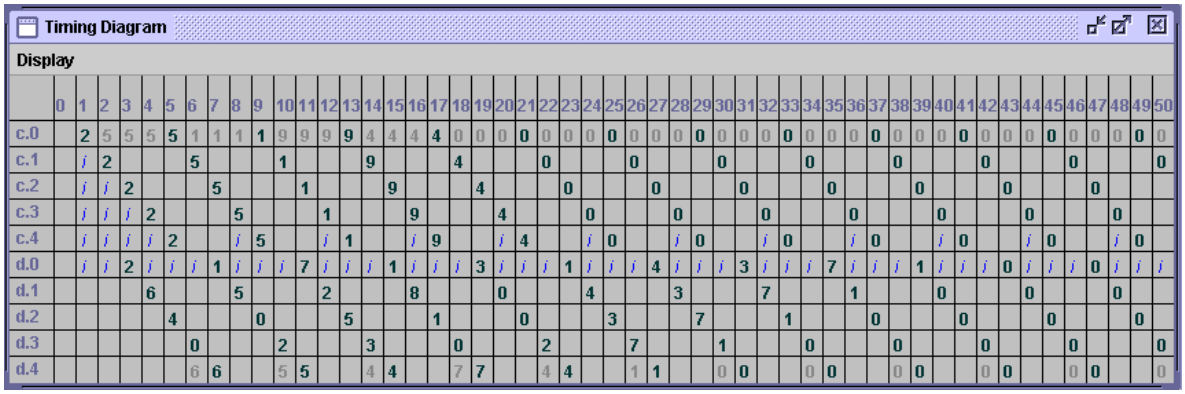

Figure 4: Timings for the second version of the multiplication pipeline.

This version of the system shows dramatically improved performance; the last digit of the result is output on time step 39. Although a large number of failures are still shown on $c.0$ and $d.0$, this is only due to the fact that these channels are connected to the pipeline loader and the result collector, both of which continuously attempt output and input respectively. Since each node sends its result for each digit back before recieving the corresponding result from the rest of the pipeline, the result digits can be propagating up the earlier nodes of the pipeline at the same time as the input digits propagate down the later nodes.

The only failures remaining are occasional failures on $c.4$ and $d.4$. These channels are connected to the pipeline end process, which has a different timing cycle to the others due to the fact that it does not have to pass its values down the pipeline. The shorter cycle means that it completes and returns its result slightly faster than any other node; this means it tries to return its result immediately after the input is sent, while the previous node is still sending its own result up the pipeline (this causes the output failures on $d.4$). Once its output is sent it does not have to wait for a result from a further node, meaning that it becomes ready for input slightly earlier than the previous node (this causes the input failures on $c.4$).

3.3 Further performance tuning

The previous specification worked quite well, but there is still a possible refinement that could be explored. In the previous specification, each node sent its own result up before receiving the result from the rest of the pipeline. Could performance be improved if each node sent its own result up before even propagating the input to the rest of the pipeline? Testing this requires only a simple change to the specification:

```
alpha NODE = [in1, out1, out2, in2]
NODE a k z = (in1?(x :: Int) -> out1!(((a*x)+z)`mod`base) ->
              NODE a k (((a * x)+z) `div` base))
              <| k == (msize - 1) |>
              (in1?(x :: Int) -> out1!((((a*x)+z)`mod`base))
              -> out2!x -> in2?(y :: Int) ->
              NODE a k (y+(((a * x) + z) `div` base)))
```

In this case, the second and third events in non-end nodes have been reversed, so the result is now sent immediately after the input is recieved, before the input is propagated down the pipeline. As before, this does not change the architecture of the system, but the performance is affected as shown by the timing diagram:

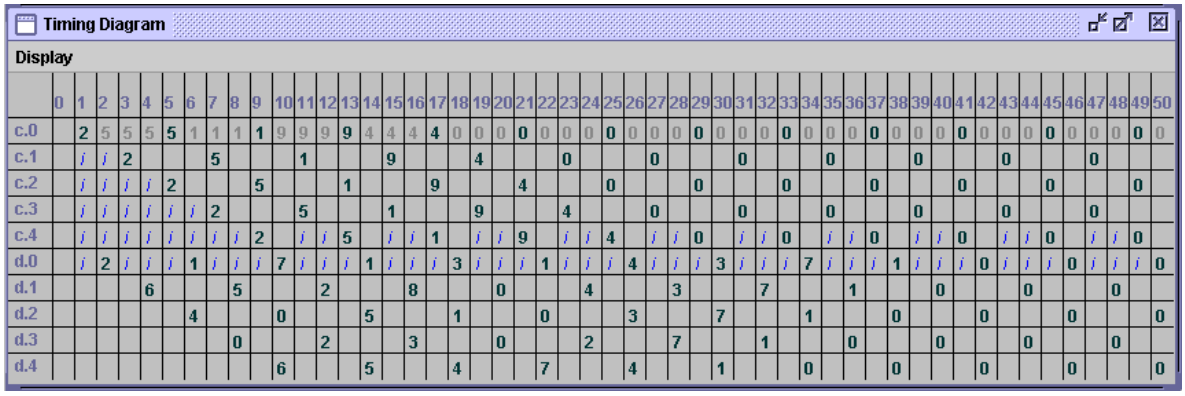

Figure 5: Timings for the third version of the Dual Pipeline Multiply.

This version of the specification shows a very small performance improvement - the last digit of result is sent one time step earlier, on step 38 rather than 39. The first digit of result is also sent a time step earlier. Both of these are entirely due to the one time step saved by returning the result before propagating the input. The failures caused by the loading and collecting processes remain (and cannot be avoided), and the failures caused by the different cycle of the end process also remain, except that they now appear as two input failures rather than one input and one output failure. This is because each node is now ready to recieve the result from the rest of the pipeline immediately after sending the input to it (which eliminates the output failure), but it has two communications to do instead of one between reading the result from the rest of the pipeline and propagating the next input digit (which causes the second input failure).

The *CSP* above describes the most efficient version of the dual pipeline multiply [3]. Most other tunings based on changing the order of the communications are either nonsensical (involving sending a calculated value before receiving a value on which the calculation is based) or cause an immediate deadlock.

Although the system discussed here is very simple, the fact that such major performance variations can exist within such a simple system demonstrates the value of performance tuning. In order to tune a more complex system it may be necessary to use modular abstraction to represent the system as a number of communicating subsystems. The *CSP* notation and the *VisualNets* tool facilitate this. The network of subsystems can be visualised, and the communications between them analyzed to identify if any particular subsystem is causing a performance problem; a problematic subsystem can then be scrutinised in detail within the overall context to establish if the problem is caused by poor performance within the subsystem or poor integration of the subsystem with the rest of the system.

4 Conclusion

The benefits of the use of simulation in performance tuning are clear, but the level of specification granularity necessary to produce an accurate simulation of a parallel system can inhibit its use. The *CSP* notation provides an adequate balance, as it specifies enough information about behaviour to enable reasonable simulation of the network, while remaining close to an architectural specification of the functions of processing nodes. *VisualNets*'s static visualisation provides a strong tie back to the architectural design of the system, and its behaviour visualisations provide a clear indication of present performance and of scope for performance improvement.

When further developed, the platform modelling features of *VisualNets* will enable performance optimisation to be performed with regard to specific platforms, or for multiple platforms without any need for reimplementation. It is also our intent to add support for additional visualisations (which will provide multiple views of the architecture of the system), and support for other input notations (such as parallel path expressions [16] as used in [1]).

References

[1] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM Software Engineering Notes*, 17(4):40–53, 1992.

[2] J. L. Ortega Arjona and G. Roberts. Architectural patterns for parallel programming. In *Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing*, 1998.

[3] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[4] M. Green. *Visualisation and Animation of Concurrent Systems specified in CSP*. PhD thesis, University of Reading, 2002.

[5] A. Abdallah and M. Green. A CSP-based integrated tool for the visualisation, animation, and performance evaluation of message passing algorithms. In *Proceedings of the IEEE Conference on Formal Engineering Methods 2000*, 2000.

[6] Topol, Brad, Stasko, Vaidy, John, and Sunderam. Pvanim: A tool for visualization in network computing environments. *Concurrency: Practice and Experience*, 10(14):1197–1222, 1998.

[7] A. Benguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing '91*, pages 435–444, 1991. Paper freely downloadable at `http://citeseer.nj.nec.com/beguelin91graphical.html`. The HeNCE tool is downloadable from `http://elib.zlib.de/netlib/hence`.

[8] T. Bemmerl and P. Braun. Visualisation of message passing parallel programs with the topsys parallel programming environment. In *Journal of Parallel and Distributed Computing 18, 2*, pages 118–128, 1993. ISSN 0743-7315.

[9] P-grade. `http://www.lpds.sztaki.hu/projects/p-grade/`.

[10] S. Utter-Honig and C. M. Pancake. Graphical animation of parallel fortran programs. In *Proceedings of Supercomputing '91*, pages 491–499, 1991.

[11] S. Schneider. *Concurrent and Real-time systems: The CSP approach*. John Wiley and Sons, Ltd, 2000. ISBN 0-471-62373-3.

[12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, 1999.

[13] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Sun Microsystems, 1999.

[14] S. Peyton Jones and J. Hughes. *Report on the programming language Haskell, a Non-Strict, Purely Functional Language*. `http://www.haskell.org/definition/`.

[15] M. Green and A. Abdallah. Interfacing Haskell with Java. In G. Michaelson and P. Trinder, editors, *Trends in Functional Programming*. Intellect, November 2000. ISBN 1-84-150024-0.

[16] R. H. Campbell and A. N. Habermann. The specification of process synchronisation by path expressions. In *Lecture Notes in Computer Science Number 16*, pages 89–102, 1974.