# Unrolling Loops With Indeterminate Loop Counts in System Level Pipelines

Hui Guo

Dept. of Computer Science & Electrical Engineering
The University of Queensland
Brisbane, Qld 4072
e-mail: guo@elec.uq.edu.au

Sri Paramewaran

Dept. of Computer Science & Electrical Engineering
The University of Queensland
Brisbane, Qld 4072
e-mail: sridevan@elec.uq.edu.au

**Abstract— This paper describes the unrolling of loops with indeterminate loop counts in system level pipelines. Two methods are discussed in this paper. The first method is the varied latency method, where the input is blocked until the pipeline is clear. This variation in the input arrival time gives rise to the name. In this method the output will be in the same order as the input. The second method, called the fixed latency method, allows for the input arrival time to remain unchanged. The loops with loop count in excess of the number of unrolled loops will have to be stored, until a suitable gap in the system becomes available. Analysis of the both methods is given, which shows that the fixed latency method is significantly faster but needs reordering of tasks and memory to store tasks.**

## I. Introduction

System level pipelines can be used when a group of data is sent through a pipeline of several stages. If each stage in a system level pipeline has the same execution time, the pipeline will exhibit the highest possible performance, and will use all available resources at all time. The goal of the designer is to balance the stages of the pipeline to be equal, in order to improve system performance and resource utilization.

For a system with no feed–back path (i.e., systems without loops) this goal of increased resource utilization and improved system performance can be met [7] [5] [9] [6]. In real world applications, however, several systems do have feedback paths.

Solutions proposed for loops in pipelines so far have mainly been on loops with determinate number of iterations [3] [11] [12] [10] [8] [2] [1]. The methods proposed for pipelines have been to either unroll or replicate the loops.

The Problem which has been explored in this paper can be expressed as follows: given a pipeline of system level hardware components, and tasks for the pipeline with indeterminate loop counts, find an efficient number of times the loop should be unrolled.

In order to find a solution, the system can be pipelined in two different ways. The first method is, where the output order is the same as the input order and there will be a blocking system which will stop the input when the pipeline is full (i.e. the latency will vary). The second method will never block the input (i.e. the latency will not vary), but the output will be in a different order to the input order. The second method also demands a large amount of memory to hold data whenever the pipeline is full.

This paper is arranged as follows. In section two, the two methods to handle varying counts of loops in the pipeline are explained; the calculations for the execution time and memory requirement are also addressed; and finally the number of times the loops should be unrolled is derived. Several verifications (via simulations) are presented in section three. Conclusions are given in the last section.

## II. Unfolding Indeterminate Loops

### A. Definitions

**Stage**, basic unit in the pipeline.

**Stage execution time**, the time required for each stage in the pipeline, denoted by $T_M$.

**Pipeline**, consisting of stages with each stage expected to have almost same execution time.

**Task**, a set of data which is processed by the pipeline.

**Latency**, arriving interval between two adjacent tasks given in the number of stages. The minimum latency of a task is 1, i.e., the next task should be at least one stage behind its preceding task.

**Loop size**, the body of a loop, given as the number of stages in the loop, denoted by $m1$.

**Unfolded loop**, the number of times a loop is unfolded in the pipeline, denoted by $l$.

### B. Varied Latency Pipeline

Let us consider a loop-containing system with folded loops, i.e., $l = 1$. The system contains $m$ stages with $m1$ stages of them forming the body of the loop, as shown

in Figure 1. $k$, which has probability density function of $f(k)$, is the number of times the loop is executed for a particular task. Note hardware to control (hereafter called the *controller* and shown in dotted lines) the branching of the loop can be merged into the last stage of the loop body.

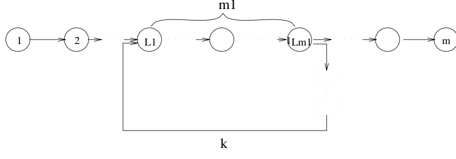


Fig. 1. *Folded loop system*

If the pipeline is to execute the tasks in the same order as they arrived, and if the pipeline is not unrolled indefinitely , then the input has to be blocked whenever the pipeline is full.

When the loop is unrolled $l$ times, the pipeline is made of $m + (l-1) \times m_1$ stages, as shown in Figure 2.

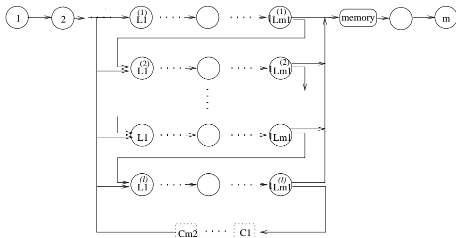


Fig. 2. *Unfolded loop system – varied latency method*

Tasks with loop count less than or equal to $l$ will be processed through the pipeline without blocking the pipeline. Tasks with loop count greater than $l$, will block the pipeline during processing.

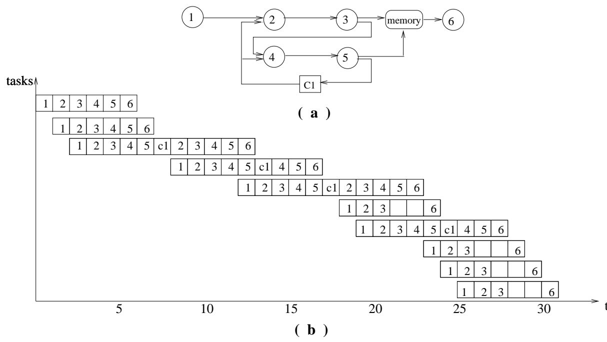


Fig. 3. *Example 1 (a)unfolded pipeline (b)space-time diagram for ten tasks*

As an example, Figure 3(a) gives a pipeline with the loop unfolded once. Stages 4 ,5 is an exact copy of stage 2, 3. Stage $c1$ is used for loop branching. Ten tasks, with each of loop count: 2, 2, 4, 3, 4, 1, 3, 1, 1 and 1, respectively, are executed in the pipeline as illustrated in figure(b). The total execution time is 31. The blank boxes indicate that the related tasks are queuing up in the memory for further processing. The maximum memory requirement is

$$Memory = (l-1) \times m_1 \times M \tag{1}$$

where $M$ is the memory needed for storing one task.

It can be seen that the blocking time can be represented in terms of the number of stages. We call this number the *blocking time* since it determines the arrival time of the next task to the pipeline and it varies from task to task.

For $n$ tasks, each having a loop count of $k_i$ where i = 1 to n, the blocking time, $b_i$, is given by

$$b_i = \begin{cases} (k_i - l) \times m1 & k_i > l \\ 0 & k_i \leq l \end{cases} \tag{2}$$

The total execution time for the n tasks is

$$T = T_{1st} + (n-1) \times T_M + \sum_{i=2}^{n} b_i \times T_M \tag{3}$$

where $T_M$ is the stage execution time and $T_{1st}$ the time taken for the first task through the pipeline. $T_{1st}$ can be defined as follows:

$$T_{1st} = \begin{cases} (m + (l-1) \times m1) \times T_M & k_1 > l \\ (m + (k_i - 1) \times m1) \times T_M & k_i \leq l \end{cases}$$

From Equation 3,

$$\begin{aligned} T &= T_{1st} + (n-1) \times T_M + (\sum_{i=2}^{n} b_i) \times T_M \\ &= T_{1st} + (n-1) \times T_M + (n-1) \times E(b) \times T_M \end{aligned} \tag{4}$$

Where $E(b)$ is the expectation value of blocking time $b$.

When n is large enough, the average execution time of each task ($T_M/n$) is

$$t \approx T_M + E(b) \times T_M. \tag{5}$$

Assuming that $b$ and $k$ are continuous variables,

$$E(b) = \int_{-\infty}^{+\infty} b f(b) db \tag{6}$$

$$= \int_{l}^{+\infty} (k-l) \times m1 \times f(k) dk. \tag{7}$$

The rate at which the execution time changes with the unrolled loop can be expressed as $\frac{dt}{dl}$, and is given by

$$\frac{dt}{dl} = -m1 \times T_M \times \int_{l}^{\infty} f(k) dk. \tag{8}$$

From equation 8, it is seen that the execution time decreases as we unroll the loop. To reduce the average execution time to the minimal value, it is required that the unfolded loop be the maximum value of $k$.

However, the decrease rate of the execution time is dependent on $\int_l^{\infty} f(k)dk$. If the variable, $k$, is in a right tailed distribution (e.g., Normal distribution), then from [4]

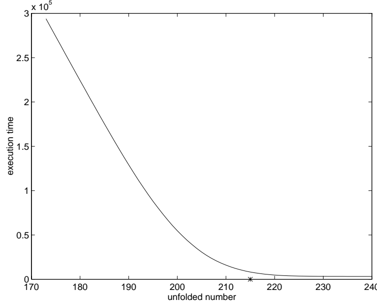


Fig. 4. *Execution time versus unfolded loop*

$$\frac{\int_l^{\infty} f(k)dk}{\int_{-\infty}^{\infty} f(k)dk} < 15\% \qquad l > E(k) + \sigma$$

where $\sigma$ is the standard deviation of the distribution. Therefore, when the *unfolded loop* is greater than the sum of expectation and deviation of $k$, the decrease is considered too slow, as indicated by $*$ in Figure 4.

Choosing the sum of expectation and variance of $k$ as the unfolded loop, we can achieve a good balance between speed and cost.

### *C. Fixed Latency Pipeline*

Bottlenecks in the pipeline not only slow down the execution but also lead to low utilization of the pipeline. As illustrated in the varied latency method, when a bottleneck occurs, stages outside the loop section are idle up to $b$ folds of stage execution times.

Tasks with small loop counts will free some stages in the unrolled loop section, which can be used by other tasks. If the memory resources are adequate, tasks with loop count larger than *unfolded loop* can be buffered until an opportunity arises to share the pipeline with smaller loop tasks. As a result the latency of each task can be guaranteed to be always same and small (small latency is required to obtain a fast pipeline).

With this strategy, tasks in example 1 can be processed in the pipeline shown in Figure 5(a). Again, stages 4, 5 and stages 2, 3 are two unfolded loops. Stage $c1$, is used for branching. The memory in the diagram is used for holding tasks which are incomplete. Figure 5(b) shows that the total execution time is reduced to 17 (as opposed to 31 in the varied latency method). It can be seen that this time improvement is at the expense of extra memory (and in need of re-ordering the tasks at the output of the pipeline if the order of the output is required). The blank boxes in Figure 5(b) represent the buffered state of the tasks. The total memory consumption is determined by the maximum number of tasks buffered at a time. In this example we need memory to buffer only a single task.

Assume the structure of the unrolled pipeline is as shown in Figure 6. The loop count of a task, $k$, is distributed in a range of $\{k_{min}, \cdots, k_{max}\}$, each with a

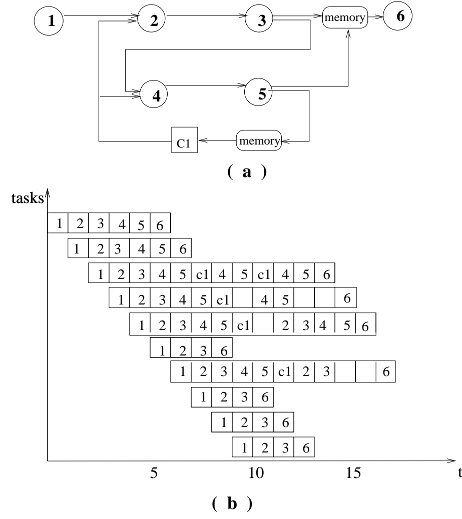


Fig. 5. *Example 2 (a)unfolded pipeline (b)space-time diagram for ten tasks*

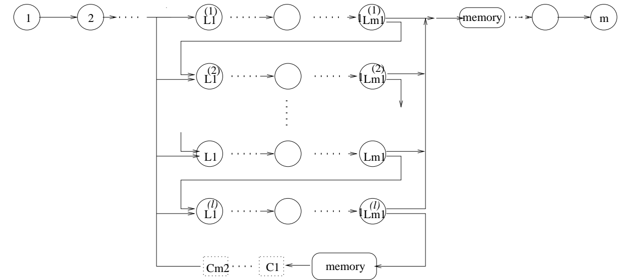


Fig. 6. *Unfolded loop system – fixed latency method*

probability of, $p(k_{min}), \cdots, p(k_{max})$.

## C.1 Memory

Assume the memory needed for each task is $M$. Assume the unfolded loop is $l$. For n tasks, the free loops provided by the tasks of loop count smaller than the unfolded loop are

$$\sum_{k=k_{min}}^{l} np(k)(l-k);$$

the excess loops produced by the tasks of loop count greater than the unfolded loop are

$$\sum_{k=l}^{k_{max}} np(k)(k-l).$$

The loops of the bufferred tasks are

$$\begin{aligned} l_{buff} &= \sum_{k=l}^{k_{max}} np(k)(k-l) - \sum_{k=k_{min}}^{l} np(k)(l-k) \\ &= \sum_{k=k_{min}}^{k_{max}} np(k)k - \sum_{k=k_{min}}^{k_{max}} np(k)l \\ &= n(E(k)-l). \end{aligned} \tag{9}$$

where E(k) is the expectation value of k.

The tasks needed to be buffered is

$$n_{task} = \frac{l_{buff}}{E(k_b)}. \tag{10}$$

where $k_b$ is the loop count of the buffered tasks, $E(k_b)$ is the expectation value of $k_b$ and $E(k_b) \geq E(k)$.

Therefore, the memory requirement is

$$\begin{aligned} Memory &= M \times n_{task} \\ &= M \times n \times \frac{E(k) - l}{E(k_b)}. \end{aligned} \tag{11}$$

From the above formula, it can be seen that the memory requirement is proportional to the number of tasks and the data size of the tasks. When $l = E(k)$, the memory requirement is statistically equal to 0.

However, there is the worst case when the loop counts are distributed so unevenly that the first part of tasks are all those with loop count over unfolded loop. All those tasks are therefore needed to be buffered until the tasks in the next part begin to release some stages in the unrolled loop section.

In a word, when the unfolded loop is equal to the expectation value of the loop count, the maximum of the memory could be

$$Memory = M \times n \sum_{exp(k)}^{k_{max}} p(k) \tag{12}$$

### C.2 Execution Time

Normally, the execution time taken by each task, except for the first task, is equal to one stage execution time, $T_M$. If the tasks need buffered, extra execution time is required. There are

$$n \times \frac{E(k) - l}{E(k_b)}$$

tasks demanding buffer. Therefore, the execution time for n tasks is

$$T = T_{1st} + (n-1) \times T_M + n \times \frac{(E(k) - l)}{E(k_b)} \times T_M \tag{13}$$

Where $T_{1st}$ is the time taken by the first task through the pipeline and is given by Formula B. If unfolded number is equal to the expectation value, $E(k)$, the execution time can be reduced to the minimum value, $nT_M$.

From the above analyses, it can be seen that the optimal unfolded loop is the expectation value of the distribution of the loop count.

## III. Results

### A. Varied Latency Method

In order to illustrate the reduction in execution time with unfolded loop, we chose three different distributions of $k$. These were uniform, normal and lognormal distributions.

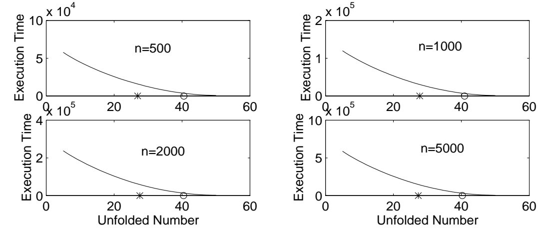


Fig. 7. *Uniform distribution under different number of tasks*

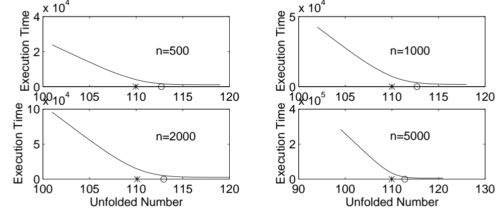


Fig. 8. *Normal distribution under different number of tasks*

The results are shown in Fig 7, Fig 8 and Fig 9.

where n is the number of tasks performed. $*$ indicates the *expectation value* (denoted by $e$) of the distribution and $o$, the *expectation value + standard deviation* (denoted by $e + \sigma$) of the distribution.

The figures show that when the *unfolded loop* is greater than the *expectation + deviation* of the distribution, the execution time becomes very small. In the cases of the normal and lognormal distributions, the slope of the reduction becomes very low, almost flat. The plot for the uniform distribution shows constant reduction.

### B. Fixed Latency Method

In this section, the simulation results are presented. Execution time versus unfolded loop is shown and compared to Method I for all three types of distributions (Figures 10 a, c, e). Memory cost versus unfolded loop is given in Figures 10 b, d, and f. We have also observed the effect of the differing expectation and plotted it against the unfolded loop.

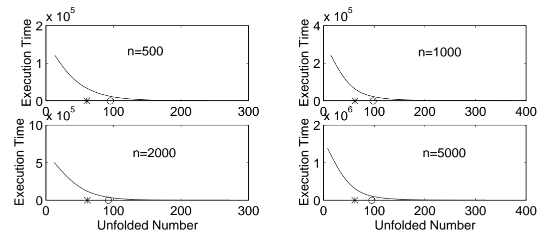


Fig. 9. *Lognormal distribution under different number of tasks*

### B.1 Comparison of Varied Latency Method and Fixed Latency Method

Figures 10 ((a), (c) and (e)) show the execution time taken for 5000 tasks versus the unfolded loop with different distributions under varied latency method (shown in dashed lines) and fixed latency method (shown in solid lines). The plots illustrate that speed improvement in the fixed latency method is substantial when the unfolded loop is small (less than $E(k)$). If the number of tasks at any time is limited, this method is significantly faster than the varied latency method. However, this improvement is at the expense of the memory cost, as shown in the Figure 10(b)(d)(f).

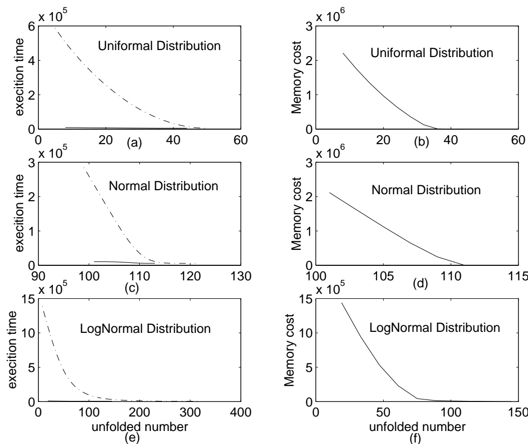


Fig. 10. *Execution time and memory requirements with different distributions*

### B.2 Memory requirement

The memory requirement were simulated under differing number of tasks (shown in Figures 11, 13 and 15). Figures 12, 14 and 16 are the plots of memory cost versus the number of tasks. It can be seen that the memory requirement linearly increases as the number of tasks is increased.

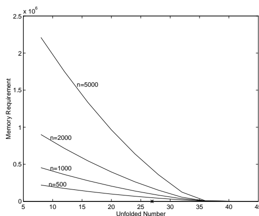


Fig. 11. *Memory requirement .vs. unfolded loop (uniform distribution)*

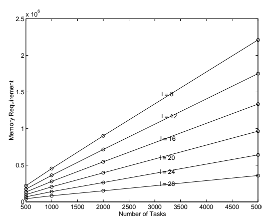


Fig. 12. *Memory requirement .vs. number of tasks (uniform distribution)*

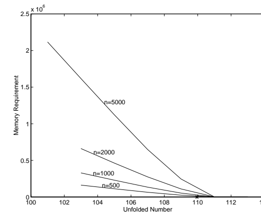


Fig. 13. *Memory requirement .vs. unfolded loop (normal distribution)*

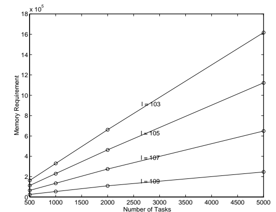


Fig. 14. *Memory requirement .vs. number of tasks (normal distribution)*

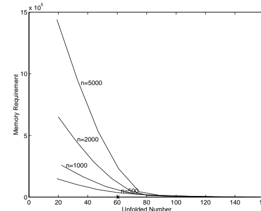


Fig. 15. *Memory requirement .vs. unfolded loop (lognormal distribution)*

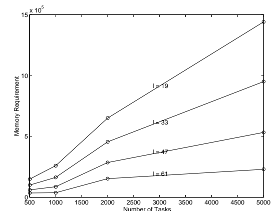


Fig. 16. *Memory requirement .vs. unfolded loop (lognormal distribution)*

### B.3 Execution Time

Figures 17, 19 and 21 are the simulation results of execution time under different number of tasks ($*$ again denotes the expectation value of loop count). The the changes of time versus the number of tasks are plotted in the Figures 18, 20 and 22.

### B.4 Effect of Expectation

Figure 23 shows the execution time with differing expectations of loop counts, as marked by asterisks. It can be seen that the efficient number of unrolled loop count will change with the expectation of the indeterminate loop count.

## IV. Conclusion

Two methods for solving loop-containing pipelining are proposed in this paper. In the first method the tasks are completed in the same order as they arrive. For a certain

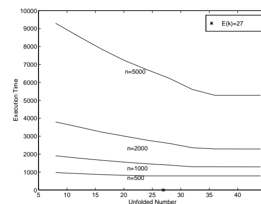


Fig. 17. *Execution time .vs. unfolded loop (uniform distribution)*

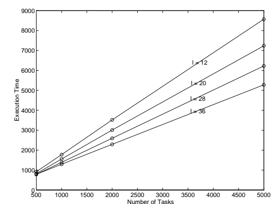


Fig. 18. *Execution time .vs. number of tasks (uniform distribution)*

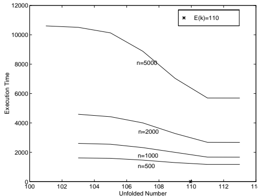


Fig. 19. *Execution time .vs. unfolded loop (normal distribution)*

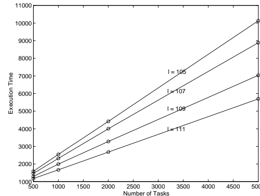


Fig. 20. *Execution time .vs. number of tasks (normal distribution)*

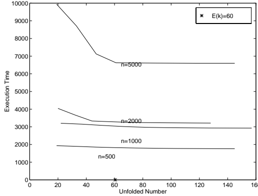


Fig. 21. *Execution time .vs. unfolded loop (lognormal distribution)*

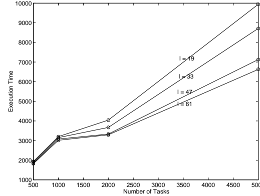


Fig. 22. *Execution time .vs. number of tasks (lognormal distribution)*

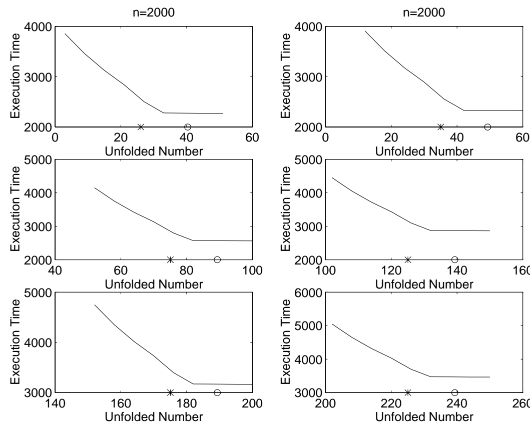


Fig. 23. *Effect of e on time – uniform distribution*

number of loops unrolled in the pipeline, tasks with loops greater than the unrolled number will take more time than the tasks with small number of loops. That is, with this method, the arriving interval of each task is dynamically changed according to the availability of the pipeline. The method is therefore called *varied latency method*. Another method, called *fixed latency method*, has varying memory requirements. In this method, the arriving interval between two adjacent tasks is fixed. Tasks with loops larger than the unfolded number will be stored until an opportunity to share the pipeline with tasks of smaller loop count.

Pipelines were simulated with the two methods under different kinds of distributions (*Uniform, Normal and Logmormal*) of indeterminate loop counts. Based on the mathematical analysis and simulations, we derived an efficient unrolled loop count so that a high speed and a low cost pipeline can be obtained. With a random count of loops specifically distributed in a range of numbers, the **varied latency method** works efficiently when the unrolled loop count is *expectation value + standard deviation* of the distribution of loop count, while the **fixed latency method** suggests the unrolled number be simply the *expectation value* of the loop count. Also, simulations show greater speed improvement with the **fixed latency method** when small unrolled loops are considered. However, this improvement is at the expense of the extra memory cost and a more complex controller for loop branching. The memory cost can be greatly reduced when suitable number of loops are unrolled.

## References

[1] Alexander Aiken, Alexandru Nicolau, and Steven Novack. Resource-constrained software pipelining. *IEEE Trans. on Parallel and Distributed Systems*, 6(No. 12):1248–1270, Dec. 1995.

[2] Vicki H. Allan and Randall M. Lee. Software pipelining. *ACM Computing Surveys*, 27(No. 3):367–432, Sep., 1995.

[3] Utpal Banerjee. A theory of loop permutations. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 54–74. The MIT Press, 1990.

[4] Roy Billinton and Ronald N Allan. *Reliability Evaluation of Engineering Systems: Concepts and Techniques*. Pitman Advanced Publishing Program, 1983.

[5] Guang R Gao and Zaharias Paraskevas. Compiling for dataflow software pipelining. In David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 275–306. The MIT Press, 1990.

[6] Hui Guo and Sri Parameswaren. System level pipelining. In *1996 APCHDL Conference Proceedings*, pages 28–33, 1996.

[7] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*,

[8] Alexandra Nicolau. Loop quantisation: A generalised loop unwinding technique. In Kai Hwang and Leonard Uhr, editors, *Journal of Parallel and Distributed Computing*, volume INC., 1988.

[9] Miodrag Potkonjak and Jan Rabaey. Fast implementation of recursive programs using transformations. *ICASSP-92: 1992 IEEE International Conference on Acoustics, Speech and Signal Proceedings*, Vol.5:569–72, 1992.

[10] J. Ramanujam. Non-unimodular transformations of nested loops. In *Proceedings of IEEE International Conference on Computer Aided Design*, pages 214–223, 1992.

[11] Jang-Ping Sheu and Tsu-Huei Tai. Partitioning and mapping nested loops on multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 430-439:430–439, Oct., 1991.

[12] Micheal E. Wolf and Monica S. Lam. A loops transformation theory and an algorithm to maximise parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 452-471:452–471, Oct, 1991.