Determining the Criticality of Processes in Kahn Process Networks for Design Space Exploration

David Hofstee† and Ben Juurlink‡
†Philips Research Laboratories WDCp-048, R. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
†‡ Computer Engineering Laboratory, Delft University of Technology,
P.O. Box 5031, 2600 GA Delft, The Netherlands
Phone: +31 (0)15 2781572, Fax: +31 (0)15 2784898
E-mail: {D.Hofstee|B.H.H.Juurlink}@its.tudelft.nl

Abstract— **Designing embedded, heterogeneous systems consisting of various hardware and software components is a complex task. One important task of the system designer is to determine which parts of the application can be implemented in software and which parts are more critical and, hence, should be performed by dedicated hardware. To help the designer determine the critical parts of the system, this paper presents an algorithm for determining the relative criticality of processes in Kahn Process Networks (KPNs). Each process is assigned a *criticality number* between 0 and 1. Intuitively, if a process P has a criticality number of $c(P)$, this indicates that process P can be slowed down by a factor of $c(P)^{-1}$ without increasing the overall execution time. Being allowed to slow down a process has several advantages. For example, since the power consumption decreases when the frequency is reduced, power can be saved by executing the process on a slower device.**

Keywords— **Kahn process network, design space exploration, parallelism, YAPI, system cost**

©Philips Research Laboratories, Eindhoven, The Netherlands

I. Introduction

Kahn process networks [3] and, more specifically, a derivative called Y-chart Application Programmers Interface (YAPI [2]) are often used within Philips to design embedded systems. In this model an application consists of a set of parallel processes, each of which performs a well-defined task, that communicate with each other via unbounded FIFO channels. Evidently, not all tasks require the same amount of time, so some processes should be performed by dedicated hardware while other processes can be implemented in software (which is generally slower than dedicated hardware).

To help the designer make this decision, this paper proposes using *criticality numbers*. Each criticality number is a number between 0 and 1 and indicates by how much a process can be slowed down. The criticality of a process not only depends on the execution time of the process, but also on the extent to which the process is on the critical path. For example, processes on the critical path must have a criticality of 1 since none of them can be slowed down without increasing the length of the critical path. As another example, a process which is executed in parallel with a process on the critical path can be slowed down by a factor equal to the execution time of the process on the critical path divided by its own execution time.

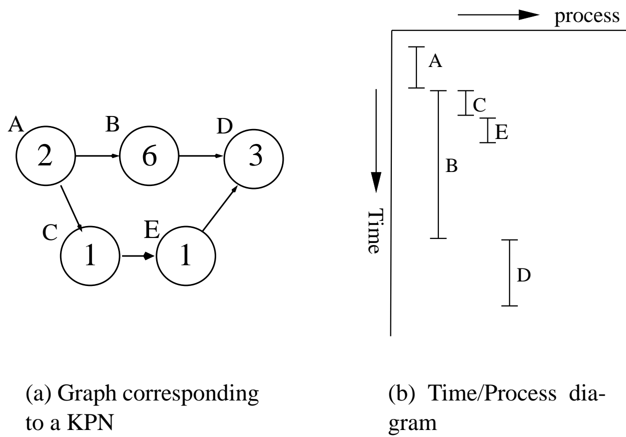
Besides identifying the most critical processes to determine which processes should be performed by dedicated hardware, there are various other reasons for determining the criticality of processes. For example, if the performance of a process is less of an issue, then the design time of its implementation is reduced since less effort needs to be spent on it. Furthermore, as mentioned previously, by executing a process on a slower processing unit, the power consumption generally decreases. The general assumption made in this paper is that cost factors such as design time and power consumption are reduced if each process is executed on the slowest processing unit within the time constraints.

In this paper only non-cyclic Kahn Process Networks are considered. In the future we plan to extend our approach to cyclic KPNs.

This paper is organized as follows. Because our analysis is not performed on KPNs but on weighted directed acyclic graphs (DAGs), Section II briefly describes how the DAG is obtained from the KPN. Section III formally defines criticality numbers and presents an algorithm to compute criticality numbers. After that, Section IV gives several applications of criticality numbers, and Section V summarizes the paper and gives directions for future research.

II. Background

We extended SpaceCAKE framework[4] developed at Philips so that the execution of KPNs can be traced. To construct a weighted DAG that corresponds to the KPN, the KPN is supplied with a small test input and simulated on a single processor. When the result is produced, a



(a) Graph corresponding to a KPN

(b) Time/Process diagram

Fig. 1. A graph and the corresponding time/process diagram

weighted DAG $G = (V, E, w)$ can be constructed whose nodes correspond to the processes of the KPN, which has an edge (u, v) if the process corresponding to node u communicated with the process corresponding to node v, and where the weight $w(v)$ of node v matches the execution time of the corresponding process. In some cases (in particular, when a process is preempted while it is being executed), a process may correspond to two or more nodes.

We assume that the *relative* execution time of each process is independent of the input data. In other words, if node u requires time $w_1(u)$ on input I_1 and time $w_2(u)$ on input I_2, and node v requires time $w_1(v)$ on input I_1 and time $w_2(v)$ on input I_2, then $w_1(u)/w_1(v) = w_2(u)/w_2(v)$. Many signal processing applications have this property.

III. Criticality Numbers

In this section we define criticality numbers and present an algorithm to obtain criticality numbers.

A. Introduction

Fig. 1(a) depicts a simple graph G. In this figure each node is labeled with the processing time it requires. When node A has been processed, nodes B and C can start. When C finishes, E can start, and when B and E have been processed, node D can start processing. In Fig. 1(b) a diagram is depicted that illustrates the execution when each process is executed on a separate processor.

In this example, the time needed by node B ($w(B) = 6$) is larger than $w(C) + w(E) = 2$. Because of this, node C and E could be executed on a processor that is $(w(C) + w(E))/w(B) = 1/3$ times as fast (in other words, 3 times as slow) as the processor that executes node B without incurring a performance penalty. This is what criticality numbers indicate. They indicate by how much a process can be slowed down without increasing the overall execution time. Obviously, the nodes on the critical path (in this case (A, B, D)) must have a criticality of 1, since they cannot be slowed down without incurring a performance penalty. Criticality numbers are properties of the network program and do not depend on the architecture it is executed on.

Note that in this example it would also be allowed to, for example, assign node C a criticality of $1/4$ and node E a criticality of $1/2$. Thus, the criticality number need not be unique.

The following definition formally defines when a criticality assignment is valid.

Definition 1: Given a weighted DAG $G = (V, E, w)$ with critical path length L. A criticality function $c : V \rightarrow (0, 1]$ is *valid* if for every path $p = (v_1, v_2, \ldots, v_n)$ in G

$$\sum_{v \in p} w(v)/c(v) \leq L.$$

This condition ensures that there is no performance penalty if each node v is slowed down by a factor of $c(v)^{-1}$.

Definition 2: Given a weighted DAG $G = (V, E, w)$ with critical path length L. A valid criticality function $c : V \rightarrow (0, 1]$ is *saturated* if for every node v there exists a path p through v such that

$$\sum_{v \in p} w(v)/c(v) = L.$$

This condition states that each node v cannot be slowed down by more that a factor of $c(v)^{-1}$ because this will increase the time needed to execute the DAG.

B. Algorithm to Obtain Criticality Numbers

In this section we present an algorithm that computes a saturated criticality function.

The algorithm is given in Fig. 2. First it computes a topological sort of the weighted DAG $G = (V, E, w)$. We recall that a topological sort is a function $f : V \rightarrow \{1, \ldots, n\}$ such that if $(u, v) \in E$, the $f(u) < f(v)$. The ordering produced by a topological sort allows to compute certain attributes associated with each node in an efficient way.

The attributes associated with node v are *begin*(v) and *end*(v). *begin*(v) is the earliest point in time node v can start processing and *end*(v) is the latest point in time node v must be finished. In other words, *begin*(v) is the length of the longest path from an initial node to v (but not including v), and *end*(v) is the critical path length L minus the length of the longest path from v to a final node. *begin*(v) can be recursively defined as

$$\textit{begin}(v) = \max_{u \in \textit{pred}(v)} \textit{begin}(u) + w(u),$$

(1) Let $v_1, v_2, \ldots, v_n$ be the nodes of the graph in the ordering produced by a topological sort.
(2) Let L be the weight of the critical path.
(3) **for** each $v \in V$ **do** $\mathit{begin}(v) \leftarrow 0$
(4) **for** $i \leftarrow 1$ to $|V|$ **do**
(5) **for** each $u \in \mathit{succ}(v_i)$ **do**
(6) **if** $\mathit{begin}(v_i) + w(v_i) > \mathit{begin}(u)$ **then** $\mathit{begin}(u) \leftarrow \mathit{begin}(v_i) + w(v_i)$
(7) **for** each $v \in V$ **do** $\mathit{end}(v) \leftarrow L$
(8) **for** $i \leftarrow |V|$ downto 1 **do**
(9) **for** each $u \in \mathit{pred}(v_i)$ **do**
(10) **if** $\mathit{end}(v_i) - w(v_i) < \mathit{end}(u)$ **then** $\mathit{end}(u) \leftarrow \mathit{end}(v_i) - w(v_i)$
(11) **for** each $v \in V$ **do** $c(v) \leftarrow 1$
(12) **for** $i \leftarrow 1$ to $|V|$ **do**
(13) let v_j be the node such that $r(v_j) = \frac{w(v_j)}{\mathit{end}(v_j) - \mathit{begin}(v_j)}$ is maximal
(14) $c(v_j) \leftarrow r(v_j)$
(15) **for** each $u \in \mathit{succ}(v_j)$ **do**
(16) **if** $\mathit{end}(v_j) > \mathit{begin}(u)$ **then** $\mathit{begin}(u) \leftarrow \mathit{end}(v_j)$
(17) **for** each $u \in \mathit{pred}(v_j)$ **do**
(18) **if** $\mathit{begin}(v_j) < \mathit{end}(u)$ **then** $\mathit{end}(u) \leftarrow \mathit{begin}(v_j)$
(19) remove v_j and all its incident edges from the graph
(20) **for** $k \leftarrow j + 1$ to $|V|$ **do**
(21) **for** each $u \in \mathit{succ}(v_k)$ **do**
(22) **if** $\mathit{begin}(v_k) + w(v_k) > \mathit{begin}(u)$ **then** $\mathit{begin}(u) \leftarrow \mathit{begin}(v_k) + w(v_k)$
(23) **for** $k \leftarrow j - 1$ downto 1 **do**
(24) **for** each $u \in \mathit{pred}(v_k)$ **do**
(25) **if** $\mathit{end}(v_k) - w(v_k) < \mathit{end}(u)$ **then** $\mathit{end}(u) \leftarrow \mathit{end}(v_k) - w(v_k)$

Fig. 2. Algorithm that computes a saturated criticality assignment.

where $\mathit{pred}(v) = \{u | (u, v) \in E\}$ is the set of predecessor nodes of v. The *begin* attribute of initial nodes is 0. Similarly, $\mathit{end}(v)$ can be recursively defined as

$$\mathit{end}(v) = L - \min_{u \in \mathit{succ}(v)} \mathit{end}(u) - w(u),$$

where $\mathit{succ}(v) = \{u | (v, u) \in E\}$ is the set of successor nodes of v and L is the length of the critical path. The *end* attribute of *end* nodes is L. By processing the nodes in the ordering produced by a topological sort, both attributes can be computed in linear time, as shown in lines (3)-(10) of the algorithm.

The algorithm then executes $|V|$ iterations. In each iteration it selects the node v_j so that $r(v_j) = w(v_j)/(\mathit{end}(v_j) - \mathit{begin}(v_j))$ is maximal over all nodes. Intuitively, this node has the least time to complete its task. Although this is not necessary for correctness of the algorithm, this heuristic may yield a better criticality function than if an arbitrary node is selected. The selected node is assigned the criticality number $r(v_j)$. Because node v_j is now slowed down by a factor of $r(v)^{-1}$, it must start processing at time $\mathit{begin}(v_j)$) (the earliest time it could start) and finish processing at time $\mathit{end}(v_j)$ (the latest point in time it must be finished to complete the DAG within time L). This implies that every predecessor of v_j must be finished before time $\mathit{begin}(v_j)$, and the earliest point in time any successor of v can start processing is $\mathit{end}(v_j)$. Accordingly, the *begin* (*end*) attribute of every successor (predecessor) of v_j is updated in lines (15)-(18) of the algorithm. After that, the selected node v_j and all its incident edges are removed from the graph so that v_j will not be considered in the next iteration of the algorithm. Finally, the *begin* attribute of every node that can be reached from v_j is updated in lines (20)-(22) of the algorithm, and the *end* attribute of every node from which v_j can be reached is corrected in lines (23)-(25).

Theorem 1: The described algorithm yields a saturated criticality assignment.

Proof: First we observe that the algorithm maintains the following invariant:

Invariant 1: Let $W(p) = \sum_{u \in p} w(u)/c(u)$ be the *weight* of the path p. For each node v

$$\mathit{begin}(v) = \max\{W(p) | \; p \text{ is a path from an initial node to a predecessor of } v\}.$$

Proof: Initially this is true because *begin*(v) is initialized to the length of the longest path from an initial node to v (but not including v) and because each node is assigned a criticality of 1 in line (11) of the algorithm. Furthermore, when the node v_j is selected and assigned the criticality of $w(v_j)/(\textit{end}(v_j) - \textit{begin}(v_j))$, then by induction $\textit{begin}(v_j)$ is the weight of the heaviest path from an initial node to a predecessor of v_j, and $w(v_j)/c(v_j) = \textit{end}(v_j) - \textit{begin}(v_j)$. It follows that $\textit{end}(v_j)$ is the weight of the heaviest path from an initial node to v_j (including v_j). By setting $\textit{begin}(u) \leftarrow \textit{end}(v_j)$ if $\textit{end}(v_j) > \textit{begin}(u)$ in lines (24)-(28) of the algorithm and by recalculating the *begin* attributes of nodes that can be reached from v_j in lines (35)-(41), the invariant is restored. ■

Similarly, the following invariant is also maintained:

Invariant 2: For each node v

$$\textit{end}(v) = L - \max\{W(p) | \\ p \text{ is a path from a successor of } v \text{ to a final node}\},$$

where L is the critical path length.

We also have to show that the algorithm maintains the following invariant as well:

Invariant 3: For each node v, $\textit{end}(v) - \textit{begin}(v) \geq w(v)$.

Proof: Consider a node v whose *begin* attribute has changed in a certain iteration of the algorithm. Then there has to be a path $(v_j, u_1, u_2, \ldots, u_n, v)$ from the selected node v_j to v such that the *begin* attribute of *all* nodes on the path has changed. Otherwise, the *begin* attribute of v would have stayed the same. Because of the way the *end* attributes are calculated, we must have $\textit{end}(v_j) \leq \textit{end}(v) - w(v) - \sum_{i=1}^{n} w(u_i)$. Furthermore, the new value of the *begin* attribute of v is $\textit{begin}(v) = \textit{end}(v_j) + \sum_{i=1}^{n} w(u_i)$. By combining these two equations we obtain $\textit{begin}(v) \leq \textit{end}(v) - w(v)$. ■

It remains to be shown that for each node v there exists a path p through v such that

$$\sum_{u \in p} w(u)/c(u) = L.$$

Consider the iteration in which node v is selected by the algorithm. By Invariant 1 the weight of the heaviest path q_1 from an initial node to a predecessor of v is $\sum_{u \in q_1} w(u)/c(u) = \textit{begin}(v)$. By Invariant 2 $\sum_{u \in q_2} w(u)/c(u) = L - \textit{end}(v)$ where q_2 is the heaviest path from a successor of v to a final node. So the required path is obtained by concatenating q_1 with v and q_2 since

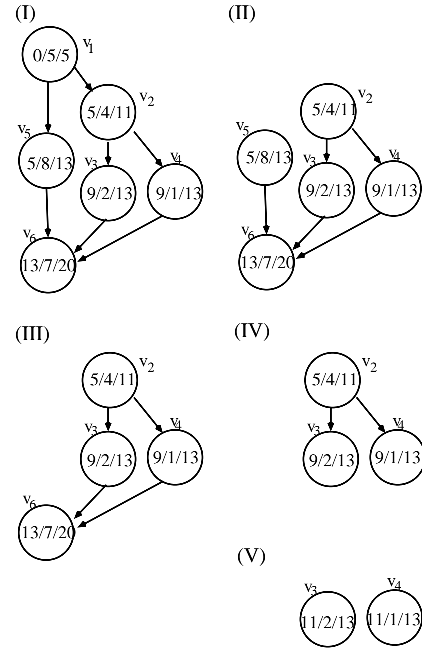


Fig. 3. Illustration of the algorithm.

$$\begin{aligned} & \sum_{u \in q_1} w(u)/c(u) + w(v)/c(v) + \sum_{u \in q_2} w(u)/c(u) \\ = \; & \textit{begin}(v) + \textit{end}(v) - \textit{begin}(v) + L - \textit{end}(v) \\ = \; & L. \end{aligned}$$

■

C. Example

Fig. 3(I) depicts a directed acyclic graph G. Each node is labeled with its *begin* attribute, its weight, and its *end* attribute, respectively. In the first three iterations the nodes on the critical path (v_1, v_5, v_6) are selected since each of them is assigned the criticality 1. Fig. 3(IV) depicts the graph that remains after the third iteration. In this fourth iteration, node v_2 is selected because $w(v_2)/(\textit{end}(v_2) - \textit{begin}(v_2))$ is larger than the corresponding value of v_3 and v_4. The criticality number of v_2 is $4/(11-5) = 2/3$. Afther that, the *begin* attribute of v_3 and v_4 are set to 11 and the situation depicted in Fig. 3(V) is obtained. Node v_3 is assigned the criticality 1 and v_4 the criticality $1/2$.

D. Complexity

The time taken by the initialization phase of the algorithm (lines (1)-(11)) is $O(|V| + |E|)$. This can be seen as follows. A topological sort can be performed in time $O(|V| + |E|)$ [1]. Furthermore, the time needed to initialize the *begin* and *end* attributes in lines (3)-(10) is also $O(|V| + |E|)$ since each edge is visited exactly once.

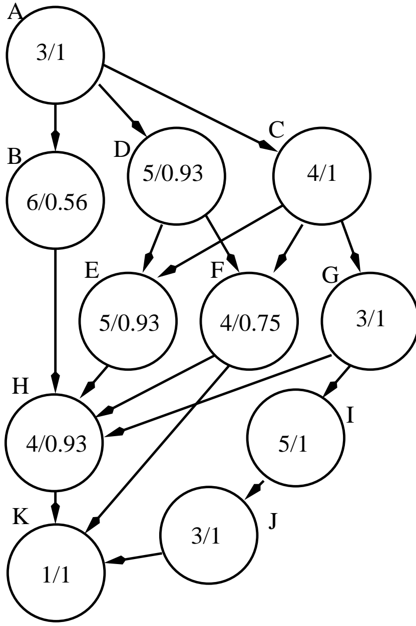


Fig. 4. Example graph with given criticality assignment.

Moreover, the length of the critical path can be determined by a simple modification of the algorithm that computes the *begin* attributes, since $L = \max_{v \in V} \mathit{begin}(v) + w(v)$.

Similarly, it can be seen that the time taken by each iteration is also $O(|V| + |E|)$. Since there are $|V|$ iterations, the total time taken by the algorithm is $O(|V| \cdot (|V| + |E|))$. Small improvements are possible because only the *begin* attributes of nodes reachable from the selected node v_j need to be recalculated and not of all nodes that occur after v_j in the topological sort, but this will not improve the asymptotic time requirements of the algorithm. We remark that a linear list is an example of a graph that forces the algorithm to consume $\Omega(|V| \cdot (|V| + |E|))$ time.

IV. Applications

In this section we give two examples of how criticality numbers can aid the designer to perform design space exploration. The first example shows that criticality numbers can be used to determine the number of homogeneous processors needed to complete the DAG within time L. The second example shows how a criticality/time diagram can be constructed from which the requirements for a heterogeneous multiprocessor system can be derived.

Throughout this section we will use the graph and criticality function depicted in Fig. 4. In this graph each node is labeled with its weight and criticality number. It can be verified that the criticality assignment is valid and saturated.

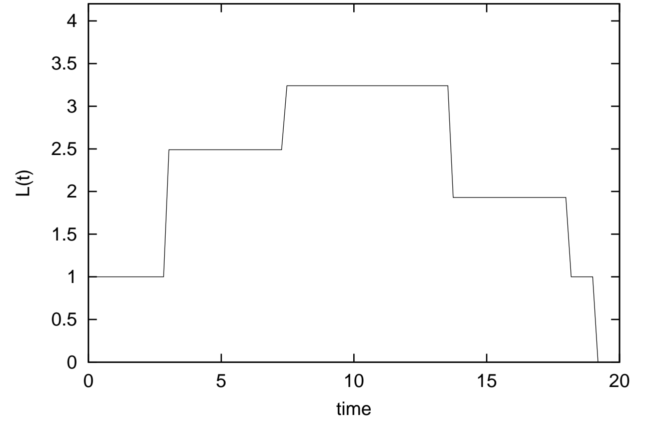


Fig. 5. Plot of the function L

A. Homogeneous Multiprocessors

For the designer of a homogeneous multiprocessor system an important question is if the number of processors is sufficient to complete the task within the time limit. Usually, this requires experimentation and investigation of various schedulers but in this section it is shown that criticality numbers can be used to find an answer to this question.

Let $A(t) = \{v | v \in V \wedge \mathit{begin}(v) \leq t < \mathit{end}(v)\}$ be the set of nodes/processes *active* at time t. Furthermore, let the function $L : R \rightarrow R$ be defined as

$$L(t) = \sum_{v \in A(t)} c(v).$$

L can be considered the *load* at time t.

Fig. 5 plots the function L for the graph depicted in Fig. 4. From this figure it can be observed immediately that four processors are be sufficient to complete the graph within time L.

B. Heterogeneous Multiprocessors

Criticality numbers can also be used to construct a criticality/time diagram such as the one depicted in Fig. 6. For each node v there is a line in this diagram from $(\mathit{begin}(v), c(v))$ to $(\mathit{end}(v), c(v))$. If there is another node u such that $\mathit{begin}(u) = \mathit{end}(v)$ and $c(u) = c(v)$, then the two lines are drawn as one line. In the figure the line labeled "path I" corresponds to the path (A, C, G, I, J, K) (the critical path), the line labeled "path II" corresponds to (D, E, H), the line labeled "path III" corresponds to (F), and the line labeled "path IV" corresponds to (B).

From this diagram it can be immediately derived that the program can be run on a heterogeneous multiprocessor with four processors whose relative speeds are 1, 0.93, 0.75, and 0.6, respectively.

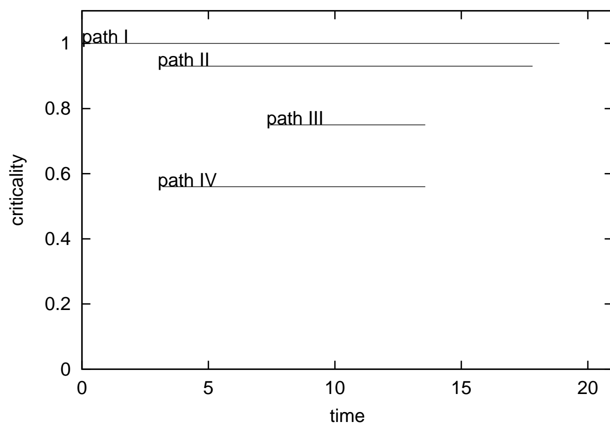


Fig. 6. Criticality/Time diagram

V. Summary and Future Work

In this paper we have defined criticality numbers. Each criticality number indicates by how much a process can be slowed without increasing the overall execution time. We have presented an algorithm that computes a criticality assignment and have given several applications of criticality numbers.

Although the presented algorithm yields a saturated criticality assignment, meaning that each process cannot be slowed down by more than the factor indicated by its criticality number, it does not necessarily yield an optimal criticality assignment (according to some cost function). We are currently devising an algorithm that produces an optimal criticality assignment. Furthermore, currently only non-cyclic networks (DAGs) can be evaluated. Since most Kahn Process Networks are cyclic, we plan to extend our approach to cyclic networks.

References

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 23.4, pages 485–488. The MIT Press, 1997.

[2] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. YAPI: Application Modelling for Signal Processing Systems. In *Proc. of the 37th Design Automation Conference*, pages 402–405, June 2000.

[3] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *J. L. Rosenfeld, editor, Information Processing 74: Proc. IFIP Congress 74, North-Holland*, pages 471–475, August 1974.

[4] P. Stravers and J. Hoogerbrugge. Homogeneous Multiprocessing and the Future of Silicon Design Paradigms. In *Proc. of the 2001 International Symposium on VLSI Technology, Systems, and Applications.*, pages 184–187, April 2001.