ISRN SICS-R–91/15–SE

Formal Derivation of Concurrent Assignments from Scheduled Single Assignments

Björn Lisper

SICS research report
R91:15
ISSN 0283-3638

Formal Derivation of Concurrent Assignments from Scheduled Single Assignments

Björn Lisper

SICS research report R91:15

Swedish Institute of Computer Science
Box 1263, S-164 28, KISTA, Sweden

October 17, 1991

Abstract

Concurrent assignments are commonly used to describe synchronous parallel computations. We show how a sequence of concurrent assignments can be formally derived from the schedule of an acyclic single assignment task graph and a memory allocation. In order to do this we develop a formal model of memory allocation in synchronous systems. We use weakest precondition semantics to show that the sequence of concurrent assignments computes the same values as the scheduled single assignments. We give a lower bound on the memory requirements of memory allocations for a given schedule. This bound is tight: we define a class of memory allocations whose memory requirements always meets the bound. This class corresponds to conventional register allocation for DAGs and is suitable when memory access times are uniform. We furthermore define a class of simple "shift register" memory allocation. These allocations have the advantage of a minimum of explicit storage control and they yield local or nearest-neighbour accesses in distributed systems whenever the schedule allows this. Thus, this class of allocations is suitable when designing parallel special-purpose hardware, like systolic arrays.

1 Introduction

A particular class of deterministic algorithms is the class of *static* algorithms. Static algorithms have a structure that is totally known in advance; every time a static algorithm is executed, the same steps will be carried out, in the same (possibly partial) order, regardless of indata. In a conventional, imperative language, static algorithms correspond to straight-line code.

A static algorithm can be formally specified by a fixed set of *single assignments* (cf. single assignment languages [1]). The basic constituents of such sets are assignments, just as for imperative programs. But in a set of single assignments, as indicated by the name, a variable is assigned at most once. Thus, single assignment variables stand for *values*, not storage locations. Sets of single assignments can be given a simple semantics [18, 19] which reflects their closeness to functional programs.

A set of single assignments can be directly interpreted as a *data dependence graph* which is a certain kind of acyclic task graph. The nodes are the single assignments and there is an arc from an assignment to another iff the variable assigned by the first assignment is used by the second. This opens up the possibility of optimizing a synchronous execution by *scheduling* the tasks (assignments) in advance. Once a schedule is found, a sequence of instructions implementing the schedule can be derived. Some instructions will be parallel, if the schedule calls for concurrent execution of tasks. Scheduling is an important part in code generation for tightly coupled synchronous systems like pipelined processors [15], VLIW architectures [8] and superscalar processors [11]. It is also possible to build synchronous hardware that supports the schedule directly: in that case, we have performed *hardware synthesis* from the functional specification given by the set of single assignments. Thus, scheduling of operations is also an important phase in high-level hardware synthesis [25]. Particular classes of schedules, known as *space-time mappings*, have furthermore been extensively studied for direct synthesis of regular hardware structures (systolic arrays and alike) [5, 6, 12, 17, 19, 21, 26, 27, 29, 30].

Synchronous computation, possibly in parallel, can be modeled by *concurrent* or *multiple assignments* [7, 10]. The execution of a simple concurrent assignment $\langle x_1, \ldots, x_n \rangle \leftarrow \langle RHS(x_1), \ldots, RHS(x_n) \rangle$, where $x_1, \ldots, x_n$ are distinct program variables, can be seen as a three-stage operation:

1. Read all variables occurring in any right-hand side.
2. Evaluate the right-hand sides with the previously read values of the occurring variables.
3. Assign each left-hand variable the value of the corresponding evaluated right-hand side.

This read-evaluate-write cycle is essentially a step in a CREW PRAM computation. A particular class of non-deterministic concurrent assignment programs is considered in Chandy and Misra's UNITY [4]. With suitable restrictions, concurrent assignments can be used to model synchronous hardware [24]. Transformations of concurrent assignments that keep desired properties invariant can be used to find efficient hardware implementations, provided that the initial specification is formulated with such assignments [3, 31].

In this paper we consider static algorithms specified by set of single assignments and we use concurrent assignments as the target model for synchronous computation. We show how a set of single assignments, if a schedule and a "register allocation" is given, can be formally translated into a sequence of concurrent assignments. We prove that this sequence really computes the functions defined by the single assignments. This is intuitively the case and is in fact often implicitly assumed throughout the literature on instruction scheduling, but so far we have seen no formal proof of this. We give a lower bound on the memory requirements of register allocations for a given schedule. The bound is tight: we define a class of memory allocation that always meet it. Finally we define a second class of register allocations, where a synchronous queue of suitable length

is allocated to each data transfer, and show that these allocations indeed are valid. This kind of allocation is simple to derive from the given schedule, it can be efficiently implemented in hardware as chains of shift registers and it is usually not wasteful of memory for highly repetitive computations where the queues are filled most of the time. Therefore it is of interest when synthesizing regular hardware systems by space-time mapping methods.

2 Preliminaries

In this paper we will use substitutions to model assignments of various kinds. We assume that the reader is somewhat familiar with universal algebra [9]. For simplicity, we consider homogenous algebras $\mathcal{A} = \langle A; F \rangle$, where A is a set and F is a set of finitary operators. (Many-sorted algebras can also be used but would lead to more complicated definitions.) Each operator f in F has an arity $n(f)$ (or *type* $A^{n(f)} \to A$) and is interpreted as a function from $A^{n(f)}$ to A. *First order expressions* over A are built from constants (interpreted as elements in the sets of A), first order variables (interpreted as *projections* [9]) and the operators in F. ϕ denotes the *natural homomorphism*: it provides an interpretation for each first order expression by mapping it to its corresponding *polynomial*, a function formed by successive applications of (interpreted) operators in exactly the same way as the expression is built up. For each polynomial p there is at least one first order expression e such that $p = \phi(e)$. $varset(e)$ denotes the set of variables in the expression e.

We define a (first order) *substitution* in X over $\mathcal{A}$ to be a partial function from the variables in X to expressions over $\mathcal{A}$, where any variables in the expressions belong to X. The *domain* of σ, $dom(\sigma)$, is the set of all variables x for which $\sigma(x)$ is defined. The *range* of σ, $rg(\sigma)$, is the set of all variables that occur in any $\sigma(x)$, i.e., $\bigcup(\, varset(\sigma(x)) \mid x \in dom(\sigma)\,)$. The set of *variables* of σ, $V(\sigma)$, is $dom(\sigma) \cup rg(\sigma)$. Substitutions are naturally extended to general expressions: $\sigma(e)$ is the expression obtained when all occurrences in e of variables x in $dom(\sigma)$ are replaced with $\sigma(x)$. *Composition* of substitutions is essentially composition of functions. The main difference is that if $\sigma(x)$ is defined but not $\sigma'(x)$, then $\sigma\sigma'(x) = \sigma(x)$ whereas the composed partial function would be undefined [19].

A usual, alternative way is to define substitutions as *total* functions from X to expressions over $\mathcal{A}$, and consider the domain of σ to be the set of variables x for which $\sigma(x) \neq x$ [13]. Which definition to choose is merely a matter of taste. We will often consider substitutions as sets of variable-expression pairs where each pair is interpreted as an assignment. In this context our particular definition of substitutions is more natural.

Finally some notation: if we have two functions $F: A \to B$ and $G: A \to C$, then $F \times G$ denotes the function $A \to B \times C$ for which $(F \times G)(a) = F(a) \times G(a)$ for all $a \in A$.

3 Single assignments

We now formalize the concept of single assignments. Informally, the following should hold:

- A variable is assigned at most once.
- There should be no data dependence cycles, i.e. chains of def-use relations such that a variable is dependent on the precence of its own value to be evaluated.

Under these conditions, each single assignment $x \leftarrow RHS(x)$ can be given the following simple operational semantics: whenever the values are present for all variables in $RHS(x)$, evaluate $RHS(x)$ and assign the resulting value to x. Note that each variable will be assigned a unique value. Furthermore, the only sequencing constraints come from def-use relationships: a variable must be assigned a value before it can be used in a right-hand side. Thus, a single assignment representation of a static algorithm does not hide the inherent parallellism that may be present.

Let us now formalize the above. (Cf. [18, 21, 22], where a similar development is made.)

Definition 1 *For any substitution σ in X, define the relation $\prec_\sigma$ on X by: $x \prec_\sigma y$ iff $x \in varset(\sigma(y))$.*

Definition 2 *The substitution σ is* causal *iff $\prec_\sigma^+$ is well-founded (a strict partial order and no infinite decreasing chains exist).*

In a causal substitution σ, each pair $\langle x, \sigma(x)\rangle$ of argument and function value can be directly interpreted as a single assignment $x \leftarrow \sigma(x)$. The relation $\prec_\sigma$ can be considered a *data dependence relation*: if $x \prec_\sigma y$, then the value of x is needed when $\sigma(y)$ is evaluated. Thus, $x \leftarrow \sigma(x)$ must be executed before $y \leftarrow \sigma(y)$ and the value of x must be sent from the time and place of the first execution to the time and place of the second. If we for a moment consider $\prec_\sigma$ (restricted to $dom(\sigma)$) as a relation on σ seen as a set of pairs, then $\langle \sigma, \prec_\sigma\rangle$ is a task graph for the set of single assignments.

The evaluation of a the right-hand side of a single assignment is considered to be strict. This will not cause any termination problems since (the transitive closure of) the dependence relation $\prec_\sigma^+$ is well-founded for causal substitutions. Furthermore, we can give causal substitutions a simple semantics.

Definition 3 *For any causal σ, σ^* is the unique substitution with domain $dom(\sigma)$ such that $\sigma^* = \sigma^*\sigma$. For any x in $dom(\sigma)$, the* output function *of x is $\phi \circ \sigma^*(x)$.*

σ^* is always well-defined for any causal σ: due to the well-foundedness of $\prec_\sigma^+$ it will, for any x, eventually hold that $\sigma^n(x) = \sigma^{n+1}(x)$ for some $n > 0$, and we find that $\sigma^*(x) = \sigma^n(x)$. Furthermore, we have the following theorem:

Theorem 1 $rg(\sigma^*) = rg(\sigma) \setminus dom(\sigma)$.

Proof. $rg(\sigma^*) \subseteq rg(\sigma) \setminus dom(\sigma)$ was proved in [19, Th. 6.2]. In order to prove $rg(\sigma^*) \supseteq rg(\sigma) \setminus dom(\sigma)$, we simply note that if a $y \notin dom(\sigma)$ belongs to $varset(e)$, then it belongs to $varset(\sigma(e))$ as well. Thus, for all $x \in dom(\sigma)$, any $y \in varset(\sigma(x))$ which is not in $dom(\sigma)$ will also belong to $varset(\sigma^n(x))$ for all $n > 0$. $y \in varset(\sigma^*(x))$ follows. ∎

Any causal substitution σ thus, for each $x \in dom(\sigma)$, defines a function $\phi \circ \sigma^*(x)$ in the variables in $rg(\sigma) \setminus dom(\sigma)$. We refer to these as *free variables*.

4 Space-time mappings of single assignments

A causal substitution can be scheduled in advance as to optimize the use of resources and minimize the runtime control overhead. The usual way to define a schedule of a task graph is to assign a time (i.e., natural number) to each task, such that precedence constraints are preserved into time. The time assigned to a task is then interpreted as the time when it is carried out. Our definition will be slightly different: we will assign a time to each *variable* in $rg(\sigma) \cup dom(\sigma)$ rather than each assignment task. This time should be interpreted as the time when the corresponding value is available for the computation(s) where it is used as input.

A basic assumption is that a processing unit can only process one task at a time. Thus, we need an allocation that for each task, besides its scheduled time, defines where it is to be executed. Together, schedule and allocation constitutes what is known as a *space-time mapping*. More formally, we assume a *space* R of possible processor coordinates, and we obtain the following definition (cf. figure 1):

Definition 4 *A* space-time mapping F *of a causal substitution* σ *is a function* $V(\sigma) \rightarrow N \times R$, *where* N *is the set of the natural numbers.* F *consists of a* schedule $F_t: V(\sigma) \rightarrow N$ *and an* allocation $F_r: V(\sigma) \rightarrow R$ *such that* $F = F_t \times F_r$. *Furthermore,* F *fulfils:*

1. *If* $x \prec_\sigma y$, *then* $F_t(x) < F_t(y)$. *(causality)*

2. F *is 1-1. (injectivity)*

This definition is based on the assumption that an input is required to be present one time unit before the results are present, i.e., the computation of each assignment takes one time unit to complete. The definition can be changed to model e.g. internally pipelined execution, non-unit time computations or various constraints on communication [19, 20, 21].

5 Prescheduled memory allocation

A schedule of a causal substitution is not enough to define an imperative program, since it does not specify how memory is used to transfer values between

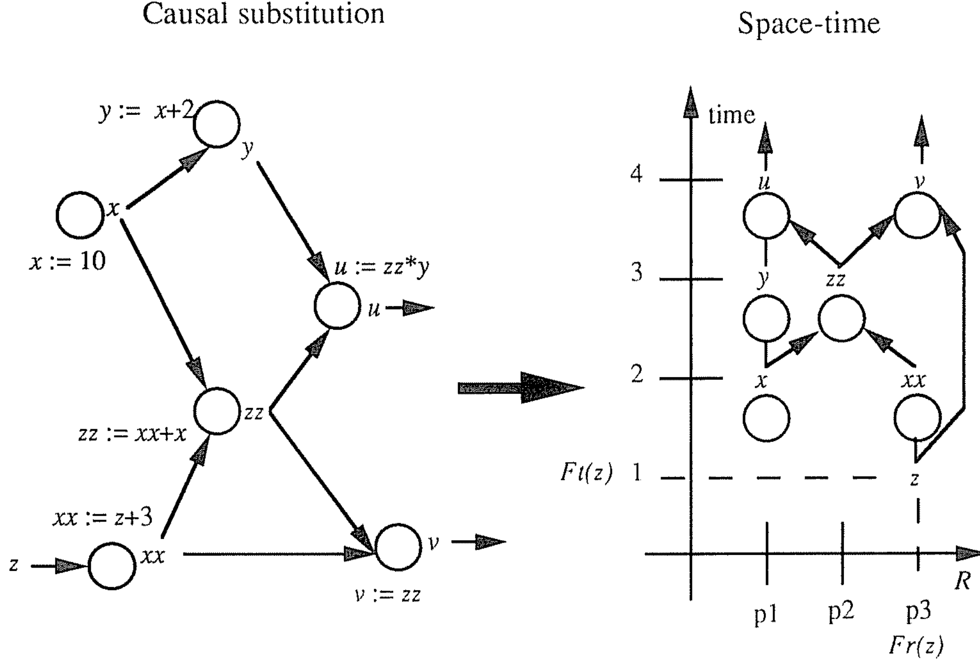

Figure 1: A space-time mapping.

the different computations. In this section we develop a formalism for this. It is based on a variation of the *transfer relations* defined in [23], where they were used to specify and verify the correctness of various prescheduled permutations of distributed data fields.

Definition 5 *Let M be a set of variables. Then $N \times M$ is an* address space-time.

Variables used to model memory will be given a different interpretation than (single assignment) variables in causal substitutions. We will refer to the former as *program variables* whenever confusion may arise.

Definition 6 *A relation $\rightarrow$ on the address space-time $N \times M$ is a* transfer relation *if it fulfils the following:*

1. *For all $\langle t, m\rangle$, $\langle t', m'\rangle$ in $N \times M$, $\langle t, m\rangle \rightarrow \langle t', m'\rangle \implies t < t'$. (causality)*

2. *$\rightarrow$ is a forest. (uniqueness of source)*

Intuitively, if $\langle t, m\rangle \rightarrow \langle t', m'\rangle$, then the data item stored in program variable m at time t becomes stored in m' at time t'. The causality property 1 rules out the transfer of data backwards in time. Forests are disjoint unions of trees: thus, elements in forests have unique immediate predecessors. Property 2 then ensures that two different data items never are written to the same address at the same time. When an address space-time event is connected to another event there is a transfer relation path between them, which means that data will be transferred from the first to the second in a number of steps.

Definition 7 *A transfer relation* $\rightarrow$ *on* $N \times M$ *is a* temporally local *if* $\langle t, m\rangle \rightarrow \langle t', m'\rangle$ *always implies* $t' = t + 1$.

A temporally local transfer relation specifies the data movements for each cycle in a synchronous system. See figure 2. It therefore provides a close description of how the transfer of data can be implemented. We will use temporally local transfer relations to specify prescheduled memory or "register" allocations. Note that these relations in a sense are more general than "conventional" register allocations, since they can describe situations where data is shifted around between several addresses during the execution rather than being fixed in a certain register. While this usually is not good practice in conventional uniprocessor programs, it is often a useful feature in parallel hardware architectures like systolic arrays.

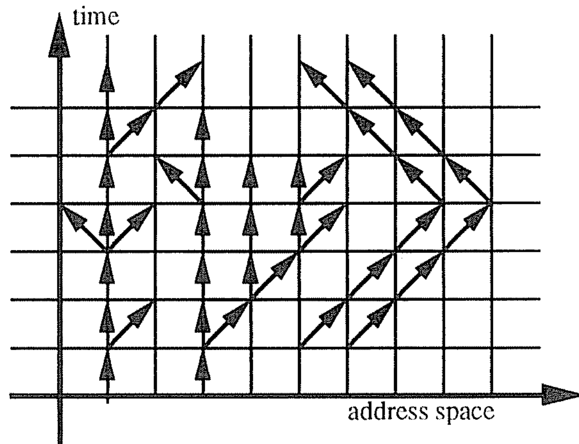

Figure 2: A temporally local transfer relation.

6 Concurrent assignments

Let us now turn to imperative programs with concurrent assignments. We are not interested in syntactic details: thus, the following definition is appropriate:

Definition 8 *A* concurrent assignment *is a substitution.*

Note that the empty substitution $\emptyset$ is allowed: this assignment is sometimes denoted "*skip*". As mentioned earlier, static algorithms correspond to straight-line programs. Therefore, we study a restricted language of imperative straight-line programs that consists of possibly infinite sequences of concurrent assignments. For convenience, we introduce the following notation for sequences: θ_n is the n-th element of the sequence θ, and θ^n_m ($m \leq n$) denotes the subsequence $\langle \theta_m, \ldots, \theta_n\rangle$ of θ. The semantics for each finite (sub)program θ^n_m in this language is given by the following two well-known weakest precondition axioms [7]:

- $wp(\theta^n_n, Q) \iff \theta_n(Q)$.

- If $m < k \leq n$, then $wp(\theta^n_m, Q) \iff wp(\theta^{k-1}_m, wp(\theta^n_k, Q))$.

7 Translating single assignments into concurrent assignments

We will now formally define a straight-line program of concurrent assignments for each causal substitution, schedule and memory allocation. We will show that in a sense, the straight-line program always computes the same values as defined by the causal substitution according to definition 3.

Definition 9 *For any causal substitution σ, schedule F_t and set of program variables M such that $M \cap V(\sigma) = \emptyset$, a* memory allocation of σ and F_t in M *is a triple $\langle W, \{ r_x \mid x \in dom(\sigma) \}, \rightarrow \rangle$ where:*

- *W is a function $V(\sigma) \rightarrow 2^M \setminus \{\emptyset\}$ such that if $x \neq y$, $m_x \in W(x)$ and $m_y \in W(y)$, then $\langle F_t(x), m_x \rangle \neq \langle F_t(y), m_y \rangle$.*
- *For each $x \in dom(\sigma)$, r_x is a function $varset(\sigma(x)) \rightarrow M$.*
- *$\rightarrow$ is a temporally local transfer relation on $N \times M$, such that:*
  1. *For any $x \in V(\sigma)$, there is no $s \in N \times M$ and no $m \in W(x)$ such that $s \rightarrow \langle F_t(x), m \rangle$.*
  2. *For all $x \in dom(\sigma)$ and all $y \in varset(\sigma(x))$ holds that there is an $m_y \in W(y)$ such that $\langle F_t(y), m_y \rangle \rightarrow^* \langle F_t(x) - 1, r_x(y) \rangle$.*

The interpretation is as follows: W ("write") defines, for all single assignment variables, where the corresponding value is to be stored upon creation (or where it is to be input, in case of a free variable). r_x ("read") tells where the inputs to the evaluation of x are to be taken from. We can note that r_x formally is a substitution in $M \cup V(\sigma)$. Property 1 of the transfer relation ensures that there are no write conflicts. (This property is not strictly necessary but will simplify the consequent definitions.) Property 2, finally, makes sure that there is a transfer path through memory from the appearance of a value to each time and place where it is used. See figure 3.

Definition 10 *For any causal substitution σ, schedule F_t and time τ, we define $X_\tau = \{ x \mid F_t(x) = \tau \}$.*

Definition 11 *For any temporally local transfer relation $\rightarrow$ and time τ, we define $M_\tau = \{ m \mid \exists m' : \langle \tau - 1, m' \rangle \rightarrow \langle \tau, m \rangle \}$.*

Definition 12 *Let $\rightarrow$ be a transfer relation. For any times τ, τ' and program variables m, m', if $\langle \tau', m' \rangle \rightarrow \langle \tau, m \rangle$ we define $anc_\tau(m) = m'$ (the "memory ancestor" of m at time τ).*

Since $\rightarrow$ is a forest, $anc_\tau(m)$ is well-defined whenever it exists. Furthermore, $anc_\tau(m)$ exists for all $m \in M_\tau$.

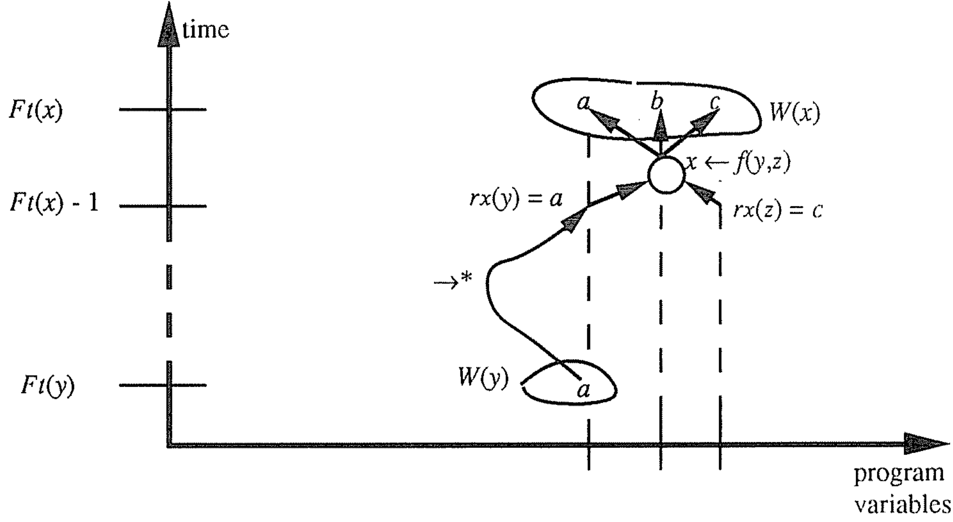

Figure 3: A memory allocation.

Definition 13 *For a causal substitution σ, a schedule F_t, a memory allocation $\langle W, \{ r_x \mid x \in dom(\sigma) \}, \rightharpoondown \rangle$ in M and any time τ, we define:*

- $c_\tau = \{ m \leftarrow anc_\tau(m) \mid m \in M_\tau \}$.
- $a_\tau = \bigcup(\{ m \leftarrow r_x(\sigma(x)) \mid m \in W(x) \} \mid x \in X_\tau \cap dom(\sigma))$.
- $i_\tau = \bigcup(\{ m \leftarrow x \mid m \in W(x) \} \mid x \in X_\tau \cap (rg(\sigma) \setminus dom(\sigma)))$.

Furthermore, we define the straight-line program of σ, F_t and $\langle W, \{ r_x \mid x \in dom(\sigma) \}, \rightharpoondown \rangle$ *as the infinite sequence θ where, for all times τ, $\theta_\tau = c_\tau \cup a_\tau \cup i_\tau$.*

Proposition 1 *c_τ, a_τ, i_τ and θ_τ are all well-defined substitutions.*

Proof. That c_τ is well-defined is immediate. From definition 9 follows that $W(x)$ and $W(y)$ are disjoint for distinct x and y whenever $F_t(x) = F_t(y)$. This is the case for all variables in X_τ: thus a_τ and i_τ are well-defined. Finally we note that the domains for c_τ, a_τ and i_τ are distinct. For a_τ and i_τ this follows similarly to the above, and that $dom(c_\tau)$ is distinct from the other two domains follows from property 1 in definition 9. Thus, their union is also a well-defined substitution. ∎

Intuitively, the construction of the concurrent assignments and the straight-line program is as follows. c_τ are assignments transferring data that is created before time τ and used after τ. a_τ are single assignments whose results are to be available at time τ: the occuring single assignment variables are mapped to program variables as specified by the memory allocation. i_τ maps free variables becoming available at time τ to program variables: this corresponds to "input" statements of conventional imperative languages. For each τ, a concurrent assignment is then formed from these three parts and the resulting straight-line

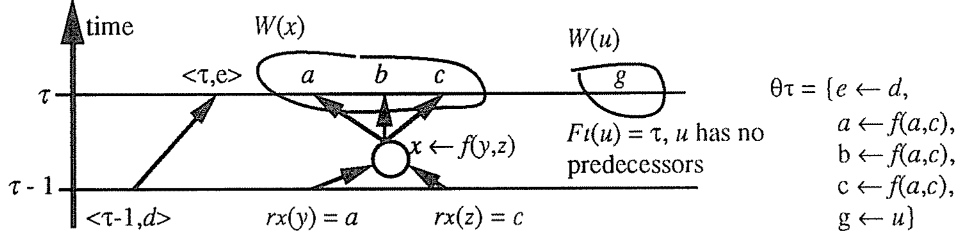

Figure 4: The construction of a concurrent assignment.

program is simply the infinite sequence of such assignments ordered by time. See figure 4.

The straight-line program obtained in this way is by no means the most efficient straight-line program for the given causal substitution, schedule and memory allocation. If for instance $\langle \tau - 1, m\rangle \rightharpoonup \langle \tau, m\rangle$, then an assignment $m \leftarrow m$ will be included in c_τ: such trivial assignments can be removed without changing the semantics. Furthermore, empty concurrent assignments can be removed from the sequence. (This may, however, change the times when results become available, and then it will strictly speaking violate the schedule F_t.) If $dom(\sigma)$ is finite, then only a finite number of concurrent assignments will be nonempty: in this case, a finite straight-line program exists which is equivalent to the infinite program of definition 13.

We now prove that the straight-line program for a causal substitution σ, schedule F_t and memory allocation $\langle W, \{ r_x \mid x \in dom(\sigma) \}, \rightharpoonup\rangle$ in M computes the same values as σ. We first show a lemma regarding the transfer of values. To make the lemma more concise we introduce the empty sequence ϵ into the language of straight-line programs, we define $wp(\epsilon, Q) \iff Q$ and finally we define, for all times τ, $\theta^{\tau}_{\tau+1} = \epsilon$.

Lemma 1 *If* $\langle \tau, m\rangle \rightharpoonup^* \langle \tau', m'\rangle$, *then* $wp(\theta^{\tau'}_{\tau+1}, m' = e) \iff m = e$.

Proof. By induction on τ'. Since $\langle \tau, m\rangle \rightharpoonup^* \langle \tau', m'\rangle$, it must hold that $\tau' \geq \tau$.

- $\tau' = \tau$: then $\langle \tau, m\rangle \rightharpoonup^* \langle \tau, m'\rangle$ which implies $m = m'$. Thus, $wp(\epsilon, m' = e) \iff m' = e \iff m = e$. Furthermore, $\theta^{\tau'}_{\tau+1} = \epsilon$ which proves the base case.

- $\tau' > \tau$: assume as induction hypothesis that for all m, m'' $\langle \tau, m\rangle \rightharpoonup^* \langle \tau' - 1, m''\rangle$ implies $wp(\theta^{\tau'-1}_{\tau+1}, m'' = e) \iff m = e$. Since $\tau' > \tau$ it follows from $\langle \tau, m\rangle \rightharpoonup^* \langle \tau', m'\rangle$ that $\langle \tau, m\rangle \rightharpoonup^* \langle \tau' - 1, anc_\tau(m')\rangle$. The induction hypothesis then implies that $wp(\theta^{\tau'-1}_{\tau+1}, anc_\tau(m') = e) \iff m = e$. Now,

$$\begin{aligned} wp(\theta^{\tau'}_{\tau+1}, m' = e) &\iff \\ wp(\theta^{\tau'-1}_{\tau+1}, wp(\theta_{\tau'}, m' = e)) &\iff \\ (c_{\tau'} \text{ and thus } \theta_{\tau'} \text{ contains } m' \leftarrow anc_{\tau'}(m')) &\iff \\ wp(\theta^{\tau'-1}_{\tau+1}, anc_{\tau'}(m') = e) &\iff m = e. \end{aligned}$$

∎

Lemma 2 *For any expression e with $varset(e) \subseteq V(\sigma)$ and any time τ, $\theta_\tau(e) = e$.*

Proof. Since $dom(\theta_\tau) \subseteq M$ and M is disjoint from $V(\sigma)$. ∎

Theorem 2 *For all times τ, all $x \in X_\tau$ and all $m \in W(x)$, it holds that $wp(\theta_0^\tau, m = \sigma^*(x)) \iff true$.*

Proof. We first prove the equality for all $x \in X_\tau \cap (rg(\sigma) \setminus dom(\sigma))$. For all such x and $m \in W(x)$, i_τ contains an assignment $m \leftarrow x$. Furthermore, $\sigma(x) = x = \sigma^*(x)$. Thus,

$$\begin{aligned} wp(\theta_0^\tau, m = \sigma^*(x)) &\iff \\ wp(\theta_0^\tau, m = x) &\iff \\ \text{(by the axioms for } wp\text{, and since } \theta_\tau(x) = x \text{ by lemma 2)} &\iff \\ wp(\theta_0^{\tau-1}, x = x) &\iff \\ wp(\theta_0^{\tau-1}, true) &\iff true. \end{aligned}$$

For $x \in X_\tau \cap dom(\sigma)$ we prove the result by induction over the times τ.

- $\tau = 0$: For any $x \in X_0 \cap dom(\sigma)$ there can be no y such that $y \prec_\sigma x$. Thus, $\sigma(x)$ must be a ground term, and $\sigma^*(x) = \sigma(x)$ follows. For the same reason, $r_x(\sigma(x)) = \sigma(x)$. Furthermore, a_0 for all $m \in W(x)$ contains $m \leftarrow r_x(\sigma(x))$. Therefore,

$$\begin{aligned} wp(\theta_0, m = \sigma^*(x)) &\iff \\ wp(\theta_0, m = \sigma(x)) &\iff \\ (\theta_0(\sigma(x)) = \sigma(x) \text{ by lemma 2)} &\iff \\ r_x(\sigma(x)) = \sigma(x) &\iff true. \end{aligned}$$

- $\tau > 0$: assume as induction hypothesis that for all $\tau' < \tau$, all $x' \in X_{\tau'} \cap dom(\sigma)$ and all $m' \in W(x')$ holds that $wp(\theta_0^{\tau'}, m' = \sigma^*(x')) \iff true$. By the first result in the proof, this will then hold for all $x' \in X_{\tau'}$ where $\tau' < \tau$. Now, for any $x \in X_\tau$ and all $m \in W(x)$,

$$wp(\theta_0^\tau, m = \sigma^*(x)) \iff wp(\theta_0^{\tau-1}, wp(\theta_\tau, m = \sigma^*(x))).$$

By lemma 2, $wp(\theta_\tau, m = \sigma^*(x)) \iff \theta_\tau(m) = \sigma^*(x)$. Furthermore, $\theta_\tau(m) = a_\tau(m) = r_x(\sigma(x))$. By property 2 of definition 9, it must hold for any $x'' \in varset(\sigma(x))$ that there is an $m'' \in W(x'')$ such that $\langle F_t(x''), m'' \rangle \rightarrow^* \langle F_t(x) - 1, r_x(x'') \rangle$. It holds that $F_t(x) = \tau$. Then, by lemma 1, $wp(\theta_{F_t(x'')+1}^{\tau-1}, r_x(x'') = \sigma^*(x'')) \iff m'' = \sigma^*(x'')$. Thus,

$$\begin{aligned} wp(\theta_0^{\tau-1}, r_x(x'') = \sigma^*(x'')) &\iff \\ wp(\theta_0^{F_t(x'')}, m'' = \sigma^*(x'')) &\iff true \end{aligned}$$

by the (extended) induction hypothesis. From definition 3 follows that $\sigma^*(x) = \sigma^*(\sigma(x)) = \sigma^*|_{varset(\sigma(x))}(\sigma(x))$. ($\sigma^*|_{varset(\sigma(x))}$ is σ^* restricted to $varset(\sigma(x))$.) We can then use distributivity of conjunction over wp [7] to show, for all $m \in W(x)$:

$$
\begin{array}{rcl}
wp(\theta_0^\tau, m = \sigma^*(x)) & \iff & \\
wp(\theta_0^{\tau-1}, r_x(\sigma(x)) = \sigma^*(x)) \wedge true & \iff & \\
wp(\theta_0^{\tau-1}, r_x(\sigma(x)) = \sigma^*(x)) \wedge & & \\
\bigwedge_{x'' \in varset(\sigma(x))} wp(\theta_0^{\tau-1}, r_x(x'') = \sigma^*(x'')) & \iff & \\
wp(\theta_0^{\tau-1}, r_x(\sigma(x)) = \sigma^*(x) \wedge & & \\
\bigwedge_{x'' \in varset(\sigma(x))} r_x(x'') = \sigma^*(x'')) & \iff & \\
wp(\theta_0^{\tau-1}, \sigma^*|_{varset(\sigma(x))}(\sigma(x)) = \sigma^*(x) \wedge & & \\
\bigwedge_{x'' \in varset(\sigma(x))} r_x(x'') = \sigma^*(x'')) & \iff & \\
wp(\theta_0^{\tau-1}, \sigma^*(x) = \sigma^*(x) \wedge & & \\
\bigwedge_{x'' \in varset(\sigma(x))} r_x(x'') = \sigma^*(x'')) & \iff & \\
wp(\theta_0^{\tau-1}, true) \wedge & & \\
\bigwedge_{x'' \in varset(\sigma(x))} wp(\theta_0^{\tau-1}, r_x(x'') = \sigma^*(x'')) & \iff & \\
true \wedge true & \iff & true.
\end{array}
$$

■

8 Memory requirements

An important parameter for a program is its memory requirements, i.e., how many memory locations does it need for its execution. In this section we will give a simple definition of memory requirements for the straight-line programs of concurrent assignments, and prove a lower bound for the memory requirements of such a program implementing a given schedule.

Definition 14 *For any straight-line program θ, its* set of program variables *is* $V(\theta) = \bigcup(\ dom(\theta_\tau) \mid \tau \in N\ )$ *and its* memory requirement *is* $m(\theta) = |V(\theta)|$.

In the following we will talk about memory requirements for causal substitutions, schedules and memory allocations. We then mean the memory requirement of the corresponding straight-line program.

For convenience, we will make some extensions to previous definitions. We add an infinite element ∞ to the natural numbers, with the properties that for any natural number $n \neq \infty$ holds that $n < \infty$, $\infty - n = \infty$ and $\infty + n =$

∞. Furthermore, for any causal substitution σ, we introduce a (non-program) variable x^* that is distinct from $V(\sigma)$. We extend any schedule $F_t: V(\sigma) \rightarrow N$ into a function $V(\sigma) \cup \{x^*\} \rightarrow N \cup \{\infty\}$ by defining $F_t(x^*) = \infty$. For any x in $V(\sigma)$ we now define $x\uparrow$, "the last variable using x", by the following:

1. If there is no y such that $x \prec_\sigma y$, then $x\uparrow = x$.
2. If there exist a y such that: (1) $x \prec_\sigma y$ and (2) for any y' such that $x \prec_\sigma y'$, $F_t(y') \leq F_t(y)$, then $x\uparrow = y$.
3. If there is a y such that $x \prec_\sigma y$, and if for all y such that $x \prec_\sigma y$ exists a y' such that $F_t(y) < F_t(y')$, then $x\uparrow = x^*$.

Lemma 3 *Let θ be the straight-line program of σ, F_t and some memory allocation $\langle W, \{ r_x \mid x \in dom(\sigma) \}, \rightarrow \rangle$. Then, for each time τ, $|dom(a_\tau \cup i_\tau)| \geq |X_\tau|$ and $|dom(c_\tau)| \geq |\{ x \mid F_t(x) < \tau \wedge F_t(x\uparrow) > \tau \}|$.*

Proof. From definition 13,

$$\begin{aligned} dom(a_\tau \cup i_\tau) &= \bigcup(W(x) \mid x \in X_\tau \cap dom(\sigma)) \cup \\ &\quad \bigcup(W(x) \mid x \in X_\tau \cap (rg(\sigma) \setminus dom(\sigma))) \\ &= (\text{since } dom(\sigma) \cup (rg(\sigma) \setminus dom(\sigma)) = V(\sigma) \text{ and } X_\tau \subseteq V(\sigma)) \\ &= \bigcup(W(x) \mid x \in X_\tau). \end{aligned}$$

Since all $W(x)$ are disjoint for $x \in X_\tau$, it then follows that $|dom(a_\tau \cup i_\tau)| \geq |X_\tau|$.

In order to prove the second inequality, we note that $dom(c_\tau) = M_\tau$. Furthermore it holds that $F_t(x\uparrow) > \tau$ exactly when there is an x' such that $x \prec_\sigma x'$ and $F_t(x') > \tau$. We now show that

$$|M_\tau| \geq |\{ x \mid F_t(x) < \tau \wedge \exists x' : x \prec_\sigma x' \wedge F_t(x') > \tau \}|.$$

Consider any $x, x', y, y' \in V(\sigma))$ such that $x \neq y$, $x \prec_\sigma x'$ and $y \prec_\sigma y'$. By property 2 of definition 9, there must be program variables m_x and m_y such that there are chains $\langle F_t(x), m_x \rangle \rightarrow \cdots \rightarrow \langle F_t(x') - 1, r_{x'}(x) \rangle$ and $\langle F_t(y), m_y \rangle \rightarrow \cdots \rightarrow \langle F_t(y') - 1, r_{y'}(y) \rangle$. $\rightarrow$ is a transfer relation and thus a forest. Furthermore, by property 1 of definition 9, it cannot be the case that $\langle F_t(x), m_x \rangle \rightarrow^* \langle F_t(y), m_y \rangle$, or vice versa. It follows that the two chains are disjoint. Now assume that $F_t(x) < \tau$, $F_t(y) < \tau$, $F_t(x') > \tau$ and $F_t(y') > \tau$. Since $\rightarrow$ is temporally local, there must by the above exist program variables m'_x, m''_x, m'_y and m''_y, where $m''_x \neq m''_y$, such that $\langle \tau - 1, m'_x \rangle \rightarrow \langle \tau, m''_x \rangle$, $\langle \tau - 1, m'_y \rangle \rightarrow \langle \tau, m''_y \rangle$, $\langle \tau, m''_x \rangle \rightarrow^* \langle F_t(x') - 1, r_{x'}(x) \rangle$ and $\langle \tau, m''_y \rangle \rightarrow^* \langle F_t(y') - 1, r_{y'}(y) \rangle$. Thus, there must be at least one distinct element m''_x in M_τ for each pair x, x' where $x \prec_\sigma x'$, $F_t(x) < \tau$ and $F_t(x') > \tau$, and the inequality follows. ■

$a_\tau \cup i_\tau$ and c_τ have disjoint domains. Thus, according to lemma 3,

$$|dom(\theta_\tau)| = |dom(a_\tau \cup i_\tau)| + |dom(c_\tau)| \geq |X_\tau| + |\{ x \mid F_t(x) < \tau \wedge F_t(x\uparrow) > \tau \}|.$$

For any τ it furthermore holds that $|dom(\theta_\tau)| \leq |\bigcup(dom(\theta_\tau) \mid \tau \in N)|$. We obtain the following result.

Theorem 3 *The memory requirement of the causal substitution σ and the schedule F_t is, for any memory allocation, bound from below by*

$$\max_{\tau \in N}(|X_\tau| + |\{\, x \mid F_t(x) < \tau \wedge F_t(x\!\uparrow) > \tau \,\}|)$$

if this maximum exists. Otherwise the memory requirement is infinite.

Theorem 3 is illustrated in figure 5. As will be seen in the next section, the bound it gives is tight. This theorem points out the importance of scheduling to memory allocation. If a low memory requirement is the goal, then the limit of theorem 3 should be part of the objective for the scheduling.

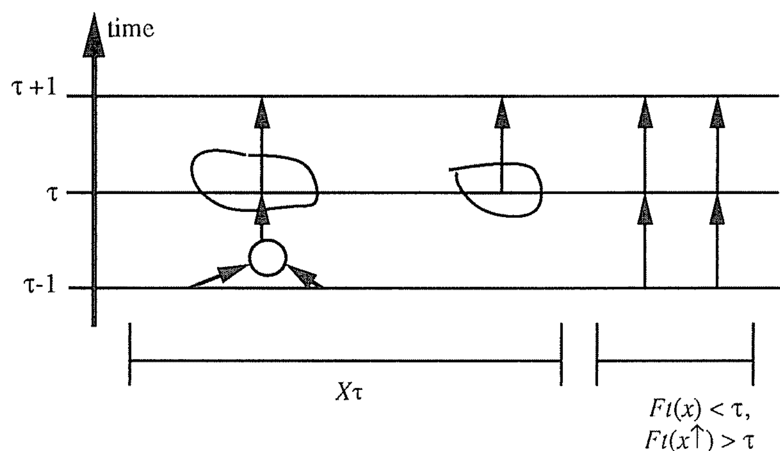

Figure 5: An illustration of theorem 3.

9 Strategies for memory allocation

In this section we will define two *memory allocation strategies*, i.e., systematic methods to find a memory allocation from a given schedule of a causal substitution. The first strategy is the natural one in systems with global RAM: it minimizes the memory requirements for the schedule, and it uses a minimum of data movements. The latter means that it will minimize the number of explicit memory handling instructions in systems where such instructions are needed to move data. This strategy is essentially conventional register allocation.

The second strategy is natural in heavily pipelined hardware implementations. It was formally described by *communication structures* in [19] and implicitly used before, most notably in work on two-level pipelining in systolic arrays [16, 28]. Here, spatial locality is important. The strategy is to allocate, for each pair of spatial coordinates r, r' and positive integer δ, a chain of δ shift registers from r to r' iff there are x, y such that: (1) $x \prec_\sigma y$, (2) x is mapped to r at time t, (3) y is mapped to r' at time t' and (4) $t' - t = \delta$. The advantage of this strategy is twofold: first, it will make the "communication part" of the concurrent assignment program time invariant, which implies that it can be implemented in hardware. Second, if the spatial allocation is such that

only spatial neighbours will ever communicate, the memory allocation allows communication through a local channel between the neighbours.

9.1 Minimal memory allocation

Definition 15 *For any causal substitution σ and schedule F_t of σ, a* minimal memory allocation $\langle W, \{ r_x \mid x \in dom(\sigma) \}, \rightarrow \rangle$ *for σ and F_t is given by:*

- *For each x in $V(\sigma)$, $W(x) = \{m_x\}$ where all m_x are distinct.*
- *For each y in $dom(\sigma)$ and each x in $varset(y)$, $r_y(x) = m_x$.*
- *$\rightarrow$ is given by: for all x in $rg(\sigma)$, $\langle F_t(x), m_x \rangle \rightarrow \langle F_t(x) + 1, m_x \rangle \rightarrow \cdots \rightarrow \langle F_t(x\dagger) - 1, m_x \rangle$. For no other s, s' holds that $s \rightarrow s'$.*

m_x is chosen according to the following. If $x \in X_0$, then m_x is chosen freely so it is distinct from any other $m_{x'}$ where $x' \in X_0$. Furthermore, we define $FV(0) = \emptyset$. For $\tau \geq 0$, we define $NV(\tau) = \{ m_x \mid F_t(x\dagger) = \tau \}$ and:

- *If $|X_\tau| \leq |FV(\tau - 1) \cup NV(\tau - 1)|$, then for each $x \in X_\tau$ m_x is uniquely selected from $FV(\tau - 1) \cup NV(\tau - 1)$ and $FV(\tau) = (FV(\tau - 1) \cup NV(\tau - 1)) \setminus \{ m_x \mid x \in X_\tau \}$.*
- *If $|X_\tau| > |FV(\tau - 1) \cup NV(\tau - 1)| = n$, then for the first n elements x from X_τ, m_x is chosen from $FV(\tau - 1) \cup NV(\tau - 1)$, and for the remaining x in X_τ fresh, distinct m_x are selected. Furthermore, $FV(\tau) = \emptyset$.*

A minimal memory allocation is shown in figure 6. It is not hard to verify that minimal memory allocations are well-defined memory allocations of σ and F_t, and the verification is left as an excercise to the interested reader. More interesting is the fact that they meet the memory requirement limit of theorem 3. Before proving this, we show two lemmas. From now on, we assume that σ is a causal substitution, that F_t is a schedule of σ, and θ denotes the straight-line program of σ, F_t and a minimal memory allocation $\langle W, \{ r_x \mid x \in dom(\sigma) \}, \rightarrow \rangle$ of σ and F_t.

Lemma 4 *For all times τ,*

$$|dom(\theta_\tau)| = |X_\tau| + |\{ x \mid F_t(x) < \tau \wedge F_t(x\dagger) > \tau \}| .$$

Proof. For a minimal memory allocation each $W(x)$ is a singleton $\{m_x\}$. Since $dom(a_\tau \cup i_\tau) = \bigcup(W(x) \mid x \in X_\tau)$ (see the proof of lemma 3), it follows that $|dom(a_\tau \cup i_\tau)| = |X_\tau|$. Furthermore, $dom(c_\tau) = M_\tau$. From the definition of $\rightarrow$ follows that M_τ contains exactly the program variables m_x, $x \in rg(\sigma)$, for which $F_t(x) < \tau$ and $F_t(x\dagger) > \tau$. Finally, there is no x outside $rg(\sigma)$ which has a successor w.r.t. $\prec_\sigma$: thus, it must be the case that $|M_\tau| = |\bigcup(x \mid F_t(x) < \tau \wedge F_t(x\dagger) > \tau)|$. The lemma follows. ∎

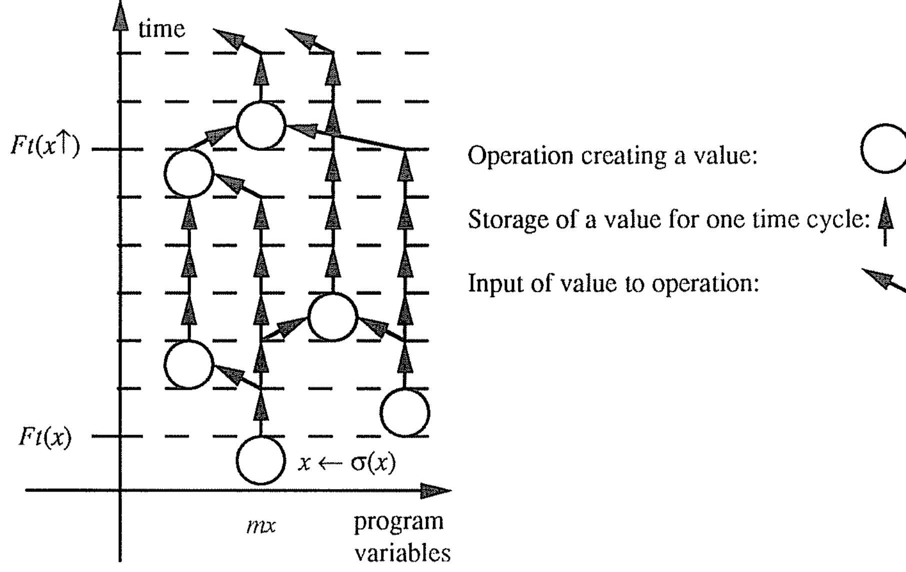

Figure 6: A minimal memory allocation.

Lemma 5 *For all times $\tau > 0$, $dom(c_\tau) \cup NV(\tau - 1) = dom(\theta_{\tau-1})$.*

Proof. We show $dom(c_\tau) \subseteq dom(\theta_{\tau-1})$, $NV(\tau - 1) \subseteq dom(\theta_{\tau-1})$ and finally $dom(c_\tau) \cup NV(\tau - 1) \supseteq dom(\theta_{\tau-1})$. From definition 15 follows that all program variables involved are of the form m_x for some $x \in V(\sigma)$.

- $dom(c_\tau) \subseteq dom(\theta_{\tau-1})$: if $m_x \in dom(c_\tau)$, then $\langle \tau - 1, m_x \rangle \rightarrow \langle \tau, m_x \rangle$. Then either $\langle \tau - 2, m_x \rangle \rightarrow \langle \tau - 1, m_x \rangle$, in which case $m_x \in dom(c_{\tau-1})$, or $\langle \tau - 2, m_x \rangle \not\rightarrow \langle \tau - 1, m_x \rangle$, in which case $F_t(x) = \tau - 1$ which implies $m_x \in dom(a_{\tau-1} \cup i_{\tau-1})$.
- $NV(\tau - 1) \subseteq dom(\theta_{\tau-1})$: if $m_x \in NV(\tau - 1)$, then $F_t(x\dagger) = \tau - 1$. Then either $x = x\dagger$, in which case $F_t(x) = \tau - 1$ and thus $m_x \in dom(a_{\tau-1} \cup i_{\tau-1})$, or $x \prec_\sigma x\dagger$, in which case $\langle \tau - 2, m_x \rangle \rightarrow \langle \tau - 1, m_x \rangle$ and then $m_x \in dom(c_{\tau-1})$.
- $dom(c_\tau) \cup NV(\tau - 1) \supseteq dom(\theta_{\tau-1})$: let m_x belong to $dom(\theta_{\tau-1})$. Then there are two cases:
  1. $x = x\dagger$: then $m_x \in NV(\tau - 1)$.
  2. $F_t(x) < F_t(x\dagger)$: then $\langle \tau - 1, m_x \rangle \rightarrow \langle \tau, m_x \rangle$ which implies that $m_x \in dom(c_\tau)$.

■

Theorem 4 $m(\theta) = \max_{\tau \in N}(|X_\tau| + |\{ x \mid F_t(x) < \tau \wedge F_t(x\dagger) > \tau \}|)$ *if this maximum exists, and infinite otherwise.*

Proof. We prove that for all times τ, there is a $\tau' \leq \tau$ such that for all $\tau'' \leq \tau$,

$$\begin{array}{rcl} \tau'' \geq \tau' & \Longrightarrow & dom(\theta_{\tau''}) \cup FV(\tau'') = dom(\theta_{\tau'}) \\ \tau'' \leq \tau' & \Longrightarrow & dom(\theta_{\tau''}) \cup FV(\tau'') \subseteq dom(\theta_{\tau'}). \end{array}$$

This implies that for all times τ, $|\bigcup(\, dom(\theta_{\tau'}) \mid \tau' \leq \tau\,)| = \max_{\tau' \leq \tau} |dom(\theta_{\tau'})|$. Together with lemma 4 the theorem then follows.

Let us now prove the statement above. This is done with induction over τ.

- $\tau = 0$: Then $\tau' = \tau'' = \tau = 0$ and $FV(0) = \emptyset$, so the statement holds trivially.
- $\tau > 0$: Assume that the statement holds for $\tau - 1$. In definition 15 there are two cases for the allocation of program variables to variables in X_τ:

  1. $|X_\tau| \leq |FV(\tau-1) \cup NV(\tau-1)|$: then $dom(a_\tau \cup i_\tau) \subseteq FV(\tau-1) \cup NV(\tau-1)$, and $FV(\tau) = (FV(\tau-1) \cup NV(\tau-1)) \setminus dom(a_\tau \cup i_\tau)$. Thus, $FV(\tau) \cup dom(a_\tau \cup i_\tau) = FV(\tau-1) \cup NV(\tau-1)$, and it follows that

     $$\begin{array}{rcl} FV(\tau) \cup dom(\theta_\tau) & = & FV(\tau) \cup dom(c_\tau) \cup dom(a_\tau \cup i_\tau) \\ & = & FV(\tau-1) \cup NV(\tau-1) \cup (c_\tau) \\ & = & FV(\tau-1) \cup dom(\theta_{\tau-1}) \end{array}$$

     where the last equality follows from lemma 5. By the induction hypothesis there is a $\tau' \leq \tau - 1$ such that $FV(\tau-1) \cup dom(\theta_{\tau-1}) = dom(\theta_{\tau'})$: it follows that also $FV(\tau) \cup dom(\theta_\tau) = dom(\theta_{\tau'})$ which, since $\tau > \tau - 1 \geq \tau'$, proves the induction step in this case.
  2. $|X_\tau| > |FV(\tau-1) \cup NV(\tau-1)|$: then $dom(a_\tau \cup i_\tau) \supseteq FV(\tau-1) \cup NV(\tau-1)$ and $FV(\tau) = \emptyset$: thus,

     $$\begin{array}{rcl} dom(\theta_\tau) & = & dom(c_\tau) \cup dom(a_\tau \cup i_\tau) \\ & \supseteq & FV(\tau-1) \cup NV(\tau-1) \cup (c_\tau) \\ & = & FV(\tau-1) \cup dom(\theta_{\tau-1}) \end{array}$$

     where lemma 5 is used once more. Again there is a $\tau' \leq \tau - 1$ such that $FV(\tau'') \cup dom(\theta_{\tau''}) = dom(\theta_{\tau'})$ for all τ'' between and including τ' and $\tau - 1$, and $FV(\tau'') \cup dom(\theta_{\tau''}) \subseteq dom(\theta_{\tau'})$ for all $\tau'' < \tau'$. It follows that for all $\tau'' < \tau$, $FV(\tau'') \cup dom(\theta_{\tau''}) \subseteq dom(\theta_\tau)$. Furthermore, $dom(\theta_\tau) \cup FV(\tau) = dom(\theta_\tau) \cup \emptyset = dom(\theta_\tau)$. Thus, the statement holds also for τ, with $\tau'' = \tau$.

■

Register allocation is in practice usually done through *graph colouring* [2]. Minimal memory allocation is related to graph colouring in the following way. For any variable $x \in V(\sigma)$, we can interpret the time interval $[F_t(x), F_t(x\dagger) - 1]$ as the *lifetime* of x. We then define the undirected conflict graph $\langle V(\sigma), E\rangle$ by $\{x, x'\} \in E$ iff their lifetimes intersect. Graph colouring can now be carried out on the conflict graph as usual.

9.2 Pipelined memory allocation

In the following we assume that R is a space, and that M is an infinite set of program variables. Let Z^+ denote the set of positive integers. Assume, finally, that μ is an injective function $R^2 \times (Z^+)^2 \rightarrow M$. In the following we will use the function m to specify series of shift registers between points in R: "$\mu(r, r', \delta, i)$ is shift register no. i in the chain of δ shift registers from r to r'". We will also use the following simplifying notation: for any two variables x, y scheduled by F_t we denote $F_t(y) - F_t(x)$ by δ_{xy}.

Definition 16 *For any causal substitution σ and space-time mapping $F_t \times F_r$ of σ, a* pipelined memory allocation *$\langle W, \{ r_x \mid x \in dom(\sigma) \}, \rightarrow \rangle$ for σ and $F_t \times F_r$ is given by the following:*

1. *For any $x \in V(\sigma)$:*
   - *If $x \prec_\sigma y$ for some y, then $W(x) = \{ \mu(F_r(x), F_r(y), \delta_{xy}, 1) \mid x \prec_\sigma y \}$.*
   - *Otherwise $W(x) = \{m_x\}$, where m_x is chosen distinct from any other $W(x')$ where $F_t(x') = F_t(x)$.*
2. *For any $y \in dom(\sigma)$ and $x \in varset(\sigma(y))$, $r_y(x) = \mu(F_r(x), F_r(y), \delta_{xy}, \delta_{xy})$.*
3. *For any x and y such that $x \prec_\sigma y$, for all times τ and for all i where $1 \leq i < \delta_{xy}$ holds that $\langle \tau, \mu(F_r(x), F_r(y), \delta_{xy}, i)\rangle \rightarrow \langle \tau + 1, \mu(F_r(x), F_r(y), \delta_{xy}, i + 1)\rangle$. For no other s, s' holds that $s \rightarrow s'$.*

Again, it is simple to verify that $\langle W, \{ r_x \mid x \in dom(\sigma) \}, \rightarrow \rangle$ is a well-defined memory allocation of σ and F_t. That $W(x) \cap W(y) = \emptyset$ whenever $x \neq y$ and $F_t(x) = F_t(y)$ follows from $F_r(x) \neq F_r(y)$, which must be true then, and the injectivity of m. The other required properties are immediate.

A pipelined memory allocation has the following property, which makes it interesting for hardware implementation:

Theorem 5 *Let θ be the straight-line program of σ, F_t and the pipelined memory allocation $\langle W, \{ r_x \mid x \in dom(\sigma) \}, \rightarrow \rangle$ for σ and $F_t \times F_r$. Then, for all times $\tau > 0$, $c_\tau = c_{\tau+1}$.*

Proof. For a pipelined memory allocation $\rightarrow$ holds, for any time τ and program variables m, m', that $\langle \tau, m\rangle \rightarrow \langle \tau + 1, m'\rangle$ iff $\langle \tau + 1, m\rangle \rightarrow \langle \tau + 2, m'\rangle$. It follows that for any time $\tau > 0$, $M_\tau = M_{\tau+1}$ and, for any m such that $anc_\tau(m)$ is defined, that $anc_{\tau+1}(m)$ is defined and equal to $anc_\tau(m)$. $c_\tau = c_{\tau+1}$ follows. ∎

Thus, $c_\tau = c$ for all $\tau > 0$, and since $dom(c)$ is disjoint from any $W(x)$ it follows that θ_0 can be replaced by $\theta_0 \cup c$ without altering the semantics of θ. *Therefore, since c is time invariant, it can be hardwired.* See figure 7. This will greatly reduce the number of explicit memory handling instructions required to implement the memory allocation.

From definition 16 another interesting property for pipelined memory allocations follows. For any chain of shift registers $\mu(r, r', \delta, 1), \ldots, \mu(r, r', \delta, \delta)$, $\mu(r, r', \delta, 1)$ will be accessed only by writes from computations mapped to r, and $\mu(r, r', \delta, \delta)$ will be accessed only by reads from computations mapped to r'. All other memory accesses will be local shifts from $\mu(r, r', \delta, i)$ to $\mu(r, r', \delta, i + 1)$. Thus, the chain can be laid out with the input to $\mu(r, r', \delta, 1)$ hardwired to r, the output of $\mu(r, r', \delta, i)$ hardwired to the input of $\mu(r, r', \delta, i+1)$ for $1 \leq i < \delta$ and the output of $\mu(r, r', \delta, \delta)$ hardwired to r'. If the allocation F_r is such that $x \prec_\sigma y$ always implies that $F_r(x)$ and $F_r(y)$ are neighbours in R, then all PE connections will be local.

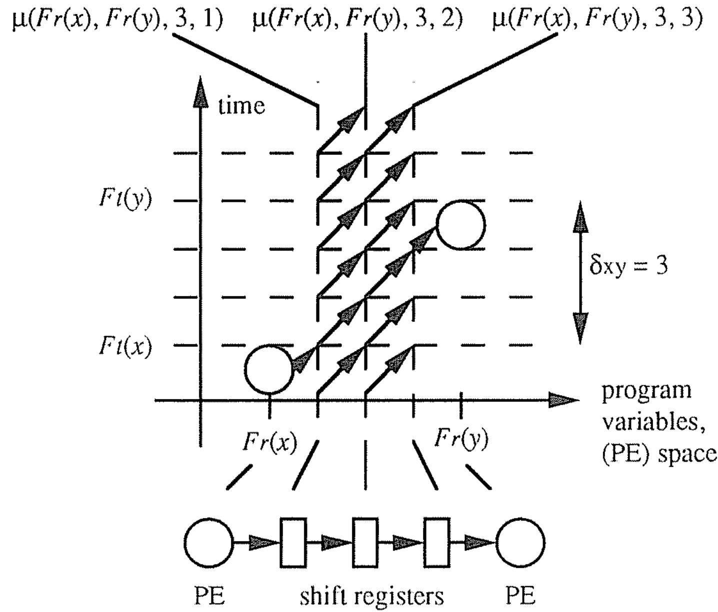

Figure 7: A pipelined memory allocation, and the resulting hardwired structure.

In general, a straight-line program for a pipelined memory allocation can be implemented as follows [19]. A processing element is allocated at each space point r such that $r = F_r(x)$ for some x. The program variables are implemented as hardwired shift registers as indicated above. The PE located at r will execute the following local straight-line program in synchrony with the others. For $\tau = 0, 1$, etc.: if there is an $x \in dom(\sigma)$ such that $F_r(x) = r$ and $F_t(x) = \tau$, then execute $\{\, m \leftarrow r_x(\sigma(x)) \mid m \in W(x) \,\}$, i.e., read all values on the shift registers $r_x(y)$, for all $y \in varset(\sigma(x))$, compute $\sigma(x)$, and write the result to the shift registers in $W(x)$. (Note that all shift registers $r_x(y)$ and in $W(x)$ are connected to in- and outports of the PE at r, respectively.) If $x \in rg(\sigma) \setminus dom(\sigma)$, then read the value of x into the shift registers in $W(x)$. If, finally, there is no x with $F_r(x) = r$ and $F_t(x) = \tau$, then the PE is free to do any action in time step τ.

Sometimes it is of interest to cluster space points. The clusters are then seen as processing element locations and the points in the cluster as locations for local "functional units" within a PE. This gives an opportunity to save memory:

if the same value is sent over two lines, with the same number of shift registers, to space points in the same cluster, then we may as well use a single line and let the receiving functional units read the same line. Formally, we consider equivalence relations "$\simeq$" on the space R. For any such equivalence relation, we postulate an injective function $\mu_{\simeq}: R \times R/\simeq \times (Z^+)^2 \to M$. Definition 16 can be modified to use $\mu_{\simeq}$ rather than μ. That the resulting memory allocation is well-defined follows in the same way as before. (Note the extreme case when $\simeq\ = E$, the total relation. It is simple to see that its memory requirement is minimal within the class of pipelined memory allocations defined here. On the other hand, any notion of locality is lost.)

9.3 An application of pipelined memory allocation

We now show a simple application of pipelined memory allocation. It is the synthesis of a standard systolic array for in-place matrix multiplication, since long known from the literature [14]. See also [19, ch. 11], where space-time mapping step of the synthesis is carried out in some detail.

The following set of single-assignments describes an algorithm for multiplying an $m \times p$ matrix A with an $p \times n$ matrix B. The algorithm is reduced w.r.t. fanout, so the elements of A and B are explicitly reused during the computation.

$$\begin{aligned}
&1 \le i \le m,\ 1 \le j \le n: \\
&\qquad\qquad c_{ij0} = 0 \\
&1 \le i \le m,\ 1 \le k \le p: \\
&\qquad\qquad a_{ik0} = a_{ik} \\
&1 \le j \le n,\ 1 \le k \le p: \\
&\qquad\qquad b_{kj0} = b_{kj} \\
&1 \le i \le m,\ 1 \le j \le n,\ 1 \le k \le p: \\
&\qquad\qquad c_{ijk} = c_{ijk-1} + a_{ikj-1} b_{kji-1} \\
&\qquad\qquad a_{ikj} = a_{ikj-1} \\
&\qquad\qquad b_{kji} = b_{kji-1}.
\end{aligned}$$

The algorithm above can be scheduled in the following way: any assignment producing an output c_{ijk}, a_{ikj} and/or b_{kji} is labeled with an *index vector* (i, j, k, α) (where $\alpha \in \{a, b, c\}$). These index vectors are linearly mapped to four-dimensional space-time coordinates (t, x, y, α) as follows:

$$\begin{pmatrix} t \\ x \\ y \\ \alpha \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ \alpha \end{pmatrix}.$$

(By a slight abuse of notation, "1" is also interpreted as identity element and "0" as absorbing element for the symbols a, b, c. The meaning of the mapping above should be clear.)

If the "α"-dimension is interpreted as a "local" dimension, i.e., $(x, y, \alpha) \simeq (x, y, \alpha')$ for all α, α', then the resulting system supporting this mapping is a

two-dimensional processor array with processor coordinates x, y ranging between 0 and m and 0 and n, respectively. The rows $x = 0$ and $y = 0$ are used only for input of elements of A and B, respectively. The following communication is needed: c_{ijk} at time and space (t, x, y) $(1 \leq x \leq m,$ $1 \leq y \leq n)$ needs c_{ijk-1} from $(t-1, x, y)$, a_{ikj-1} from $(t-1, x-1, y)$ and b_{kji-1} from $(t-1, x, y-1)$. A pipelined memory allocation w.r.t. $\simeq$ then yields the program variables $\mu_{\simeq}((x, y, c), (x, y), 1, 1)$ (which we immediately rename to $c(x, y)$) for c_{ijk-1}, $\mu_{\simeq}((x-1, y, a), (x, y), 1, 1) = a(x-1, y)$ for a_{ikj-1} and $\mu_{\simeq}((x, y-1, b), (x, y), 1, 1) = b(x, y-1)$ for b_{kji-1}, $1 \leq x \leq m$, $1 \leq y \leq n$. a_{ikj} and b_{kji} needs a_{ikj-1} and b_{kji-1}, respectively. Since their space coordinates are "clustered" with the coordinate for c_{ijk}, they will read from the same program variable. Finally, for the "output" values c_{ijp}, a_{ikn} and b_{kjm}, we define $W(c_{ijp}) = \{c(i, j)\}$, $W(a_{ikn}) = a(i, n)$ (distinct from any other $a(x, y)$) and $W(b_{kjm}) = b(m, j)$ (distinct from any other $b(x, y)$). See figure 8.

Denote the vectors (i, j, k) and (t, x, y) by $\boldsymbol{i}$ and $\boldsymbol{s}$, respectively. A system of linear inequalities for i, j and k is then of the form $\boldsymbol{A}\boldsymbol{i} \leq \boldsymbol{b}$. If $\boldsymbol{i}$ is mapped linearly to $\boldsymbol{s}$, i.e., $\boldsymbol{s} = \boldsymbol{T}\boldsymbol{i}$, then the inequalities for $\boldsymbol{i}$ yields the inequalities $\boldsymbol{A}\boldsymbol{T}^{-1}\boldsymbol{s} \leq \boldsymbol{b}$ for $\boldsymbol{s}$. The single assignments above are defined by indices constrained by linear inequalities: the given space-time mapping thus transforms these into constraints on the space-time coordinates. For any given time τ, we obtain constraints on x and y for the concurrent assignment θ_τ in the straight-line program for the schedule. Each θ_τ is then given by the following:

$1 \leq x \leq m,\ 1 \leq y \leq n,\ x + y = \tau$:

$$c(x, y) \leftarrow 0$$

$1 \leq x \leq m,\ x \leq \tau - 1,\ x \geq \tau - p$:

$$a(x, 0) \leftarrow a_{ik} \text{ (where } i = x,\ k = \tau - x)$$

$1 \leq y \leq n,\ y \leq \tau - 1,\ y \geq \tau - p$:

$$b(0, y) \leftarrow b_{kj} \text{ (where } j = y,\ k = \tau - y)$$

$1 \leq x \leq m,\ 1 \leq y \leq n,\ x + y \leq \tau - 1,\ x + y \geq \tau - p$:

$$\begin{aligned} c(x, y) &\leftarrow c(x, y) + a(x-1, y)b(x, y-1) \\ a(x, y) &\leftarrow a(x-1, y) \\ b(x, y) &\leftarrow b(x, y-1). \end{aligned}$$

Note that θ_τ is empty whenever $\tau < 2$ or $\tau > p + n + m$.

The memory requirement of the allocation above is $3mn + m + n$, which is easily verified. It is interesting to compare this with the lower limit for the memory requirement of the schedule. For any two variables u, v above, such that $u \prec_\sigma v$, holds that $F_t(v) = F_t(u + 1)$. Thus, the second term in the right-hand side of theorem 3 is zero for all times τ, and the lower limit equals $\max_{\tau \in N}(|X_\tau|)$. If $p \geq m + n - 1$, then this number is $3mn$ (which occurs for $m + n + 1 \leq \tau \leq p - 2$). Thus, the memory requirement of the pipelined memory allocation differs from the optimal number with $m + n$. This difference is, however, due entirely to the saving of the elements a_{ikn} and b_{kjm}: if they are not needed, then the corresponding program variables can be dropped and an optimal memory allocation results.

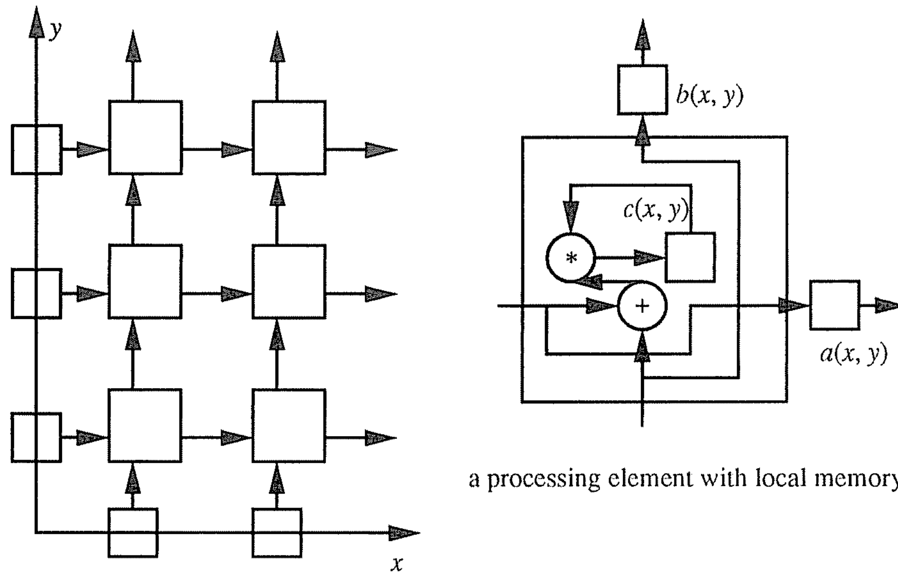

Figure 8: A pipelined memory allocation, and the resulting hardwired structure.

In this example, all c_τ are empty since there is no communication taking time greater than one. Examples with non-empty c_τ include two-level pipelined arrays for FIR filtering [16, 21, 28] and an $O(\log n)$ processor $O(n)$ time semisystolic array for n-point FFT [19].

10 Conclusions and further research

We have provided a framework for reasoning about memory allocation. This framework unifies such seemingly distant activities as register allocation and synthesis of synchronous hardware. They have in common that values must be transferred from the source computations to the destinations: this requires memory, and the two memory allocation strategies that we define are designed to meet the different objectives that arise in the different situations. It is our belief that memory allocation strategies for other types of synchronous target systems can be defined, verified and evaluated in our framework. Further development is certainly necessary to adapt the simple strategies defined here to more realistic conditions, e.g. including memory hierarchies, taking the cost of memory handling instructions into account, etc.

We also want to point out the bridging provided between the "concurrent assignment school" and the "space-time mapping school" in the field of regular hardware synthesis. We have demonstrated here how concurrent assignments can be formally derived from single assignments, schedule and memory allocation. Which school to choose depends on whether one prefers to specify the intended action of the hardware by imperative programs or recursive functions.

A limitation of the development here is the restriction to static algorithms, or basic blocks. The extension to more general classes of algorithms is an interesting task for further study.

11 Acknowledgements

I want to thank Per Hammarlund for interesting discussions and criticism.

References

[1] W. B. Ackerman. Data flow languages. *Computer*, 15:15–25, Feb. 1982.

[2] G. J. Chaitin. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices*, 17(6):201–207, 1982.

[3] K. M. Chandy and J. Misra. Systolic algorithms as programs. *Distributed Comput.*, 1:177–183, 1986.

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

[5] M. C. Chen. Transformation of parallel programs in Crystal. In H.-J. Kugler, editor, *INFORMATION PROCESSING 86*, pages 455–462. Elsevier Publishers B.V. (North-Holland), 1986.

[6] J.-M. Delosme and I. Ipsen. Efficient systolic arrays for the solution of Toeplitz systems: an illustration of a methodology for the construction of systolic architectures in VLSI. In W. Moore, A. McCabe, and R. Urquhart, editors, *Systolic Arrays*, pages 37–46, Bristol, UK, 1987. Adam Hilger.

[7] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, N.J., 1976.

[8] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, Cambridge, MA, 1986.

[9] G. Grätzer. *Universal Algebra.* Springer-Verlag, New York, NY, 1979.

[10] D. Gries. The multiple assignment statement. *IEEE Trans. Software Eng.*, SE-4(2):89–93, Mar. 1978.

[11] J. H. S. Warren. Instruction scheduling for the IBM RISC system/6000 processor. *IBM J. Res. Develop.*, 34(1):85–92, Jan. 1990.

[12] C.-H. Huang and C. Lengauer. The derivation of systolic implementations of programs. *Acta Inform.*, 24:595–632, 1987.

[13] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. Assoc. Comput. Mach.*, 27(4):797–821, Oct. 1980.

[14] E. Katona. Cellular algorithms for binary matrix operations. In *Proc. CONPAR-81*, pages 203–216. Springer-Verlag, 1981.

[15] S. M. Krishnamurthy. A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Notices*, 25(7):97–106, July 1990.

[16] H. T. Kung and M. S. Lam. Wafer-scale integration and two-level pipelined implementations of systolic arrays. *J. Parallel Distrib. Comput.*, 1(1):32–63, Aug. 1984.

[17] S. Y. Kung. VLSI array processors. In W. Moore, A. McCabe, and R. Urquhart, editors, *Systolic Arrays*, pages 7–24, Bristol, UK, 1987. Adam Hilger.

[18] B. Lisper. Single assignment semantics for imperative programs. In E. Odjik, M. Rem, and J.-C. Syre, editors, *Proc. PARLE'89 vol. II: Parallel Languages*, pages 321–334, Berlin, June 1989. Volume 366 of *Lecture Notes in Comput. Sci.*, Springer-Verlag.

[19] B. Lisper. *Synthesis of Synchronous Systems by Static Scheduling in Space-Time*, volume 362 of *Lecture Notes in Comput. Sci.* Springer-Verlag, Heidelberg, May 1989.

[20] B. Lisper. Linear programming methods for minimizing execution time of indexed computations. In *Proc. Int. Workshop on Compilers for Parallel Computers*, pages 131–142, Dec. 1990.

[21] B. Lisper. Synthesis of time-optimal systolic arrays with cells with inner structure. *J. Parallel Distrib. Comput.*, 10(2):182–187, Oct. 1990.

[22] B. Lisper. Detecting static algorithms by partial evaluation. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 31–42, June 1991.

[23] B. Lisper and S. Rajopadhye. Affine permutations of matrices on mesh-connected arrays. invited chapter in *Parallel Algorithms and Architectures for DSP Applications*, Ed. M. A. Bayoumi, Kluwer Academic Publishers (in press).

[24] A. R. Martin and J. V. Tucker. The concurrent assignment representation of synchronous systems. *Parallel Computing*, 9:227–256, 1988/89.

[25] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proc. IEEE*, 78(2):301–318, Feb. 1990.

[26] D. I. Moldovan. On the analysis and synthesis of VLSI algorithms. *IEEE Trans. Comput.*, C-31:1121–1126, Oct. 1982.

[27] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. 11th Annual Int. Symp. on Comput. Arch.*, pages 208–214, June 1984.

[28] P. Quinton and P. Gachet. Automatic design of systolic chips. Res. Rep. RR 450, INRIA, Rennes, Oct. 1985.

[29] S. V. Rajopadye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Comput.*, 3:88–105, 1989.

[30] S. K. Rao and T. Kailath. Regular iterative algorithms and their implementation on processor arrays. *Proc. IEEE*, 76(3):259–269, Mar. 1988.

[31] J. L. A. Van de Snepscheut and J. B. Swenker. On the design of some systolic algorithms. *J. Assoc. Comput. Mach.*, 36(4):826–840, Oct. 1989.