1989

Controlling fine-grain non-numeric parallelism on a combinator-based multiprocessor system

Pong Ping Chu
Iowa State University

Order Number 9014890

Controlling fine-grain non-numeric parallelism on a combinator-based multiprocessor system

Chu, Pong Ping, Ph.D.

Iowa State University, 1989

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Controlling fine-grain non-numeric parallelism on a combinator-based multiprocessor system

by

Pong Ping Chu

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department: Electrical Engineering and Computer Engineering
Major: Computer Engineering

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa
1989

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Because of the demand for more powerful computers, interest in multiprocessor systems has increased rapidly in recent years. Many projects are investigating obtaining an order-of-magnitude higher performance by using replicated low-cost hardware units [35] [53]. After dividing and distributing computations into subtasks, several processors can work for the same task concurrently and hence the overall throughput can be accelerated.

Because the computations can be performed in several different places, multiprocessor systems require additional control mechanisms to coordinate the execution. Included are how to divide a task into subtasks (partitioning), how to allocate resources (scheduling), how the subtasks interact (communication), and how to enforce time-ordering constraints (i.e., sequencing) among the subtasks (synchronization) [16]. Since these operations do not contribute any "useful" computations to the original task, they are just overhead associated with concurrent execution. To achieve good performance, the granularity/overhead ratio should be reasonably large; i.e., the overhead should be small in comparison to the total operations performed in the subtask.

Controlling fine-grain parallelism is difficult since it cannot tolerate complicated or time-consuming scheme for scheduling, communication or synchronization. Thus,

most fine-grain multiprocessor systems, such as SIMD or vector computers, exploit only "structured" fine-grain parallelism in which massive, homogeneous operations can be initiated by simple control constructs. There are very few multiprocessor systems attempting to exploit irregular fine-grain parallelism, such as those found in symbolic-oriented applications, because of the potentially high overhead and the programming difficulty.

One approach to exploiting irregular parallelism is focused on using functional programming languages. Because of the lack of side-effects, functional languages do not impose any ordering constraints [17] [27], and hence do not impose severe requirements on synchronization and communication. However, despite their attractive properties at the abstract level, functional languages have not yet proven competitive because of their slow execution on von Neumann architectures.

In recent years, a new implementation model for functional languages, known as *combinator reduction* [56], has received attention. In this paradigm, we translate functional programs into a form that does not contain any bound variables. Instead, we employ a special kind of operator, called a combinator, to denote where the variables were located before they were removed. Since variables are eliminated, there is no need to maintain a centralized, global environment. Therefore, the combinator paradigm offers a better opportunity to exploit potential fine-grain parallelism.

The work described in this dissertation investigates extending combinators to include the control of fine-grain parallelism. To accomplish this, we first develop a new control mechanism that can initiate irregular fine-grain parallelism, and then evaluate its effectiveness through simulation of some representative benchmark pro-

grams. The major effect of the extended control mechanism is to override the original lazy semantics by augmenting proper "eagerness" information. We have chosen to focus on Turner's SASL language due to its simplicity and fully-functional nature. Our benchmark programs were annotated with the new control information and then translated to the proper combinator control tags by a new set of optimization rules. Simulation results show that this scheme can extract a significant amount of unstructured parallelism and is particularly effective on the irregular manipulation of lists.

The remainder of this dissertation is organized as follows: Chapter 2 provides necessary background information on combinators and reviews related research work; Chapter 3 describes the extended multiprocessor combinator system including the annotation of SASL programs, the control tags of combinators and the extended compiling algorithm; Chapter 4 describes the design of the simulator; Chapter 5 studies the effects of the proposed scheme via the simulation of a set of selected benchmark programs; and the last Chapter gives conclusions of this research and outlines the potential future work.

CHAPTER 2. BACKGROUND

In this chapter, we first introduce the relevant concepts of functional languages, combinators, λ-calculus and the bound variable abstraction mechanism. Then, we review related research projects.

Functional Languages

Functional languages are based on the formal study of mathematical functions and function application [17] [27]. A functional program can be regarded as a collection of function definitions, and its execution as repeated function application. Unlike programs written in procedural languages, functional programs do not need to explicitly specify the order and the state of execution because the control is implicitly imbedded in the function application mechanism. The elimination of the notion of order of execution and machine state has a significant impact on the language's syntax and semantics. Syntactically, because the major "bulky" structures of procedural languages are removed [5], the syntax of a functional language is simple, clear and expressive. Semantically, because ordering constraints are removed, the meaning of a functional program can be interpreted as simple term substitution [17]. The following SASL examples illustrate the general style and capability of functional languages:

```
|| example 1:  Fibonacci number
```

```
   fib 1 = 1
   fib 2 = 1
   fib n = fib (n-1) + fib (n-2)

|| example 2:  Insertion sort

   isort   ()  = ()
   isort (a:x) = insert a (sort x)
   insert a   ()  = a:NIL
   insert a (b:x) = a<b -> a:b:x ; b:(insert a x)

|| example 3:  Homogeneous operation on a list

   map f   ()  = ()
   map f (a:x) = (f a):(map f x)

|| example 4: Fibonacci list

   fib_list = f_aux 1 1
   f_aux a b = a:(f a a+b)

|| example 5: Increment by one:

   plus x y = x + y
   inc = plus 1
```

Despite the simplicity of these examples, they show many important properties, some of which are frequently encountered in functional languages, such as recursion, pattern-matching, higher order argument, infinite data structures, and partially-evaluated function.

λ-calculus

Although different functional languages may have totally different syntactic structures, all of them are "sugared" forms of λ-calculus. Because of the compactness of a λ-expression, λ-calculus will be used as our base language. The SASL programs shown in the previous section and in the later chapters should be treated as a syntactic variation of λ-calculus rather than a new language. We will use λ-expressions and SASL interchangeably in the remainder of this thesis.

λ-calculus is a formal system used to describe functions and function application. Its syntax and operation semantics are formally defined in the following paragraphs. Syntactically, a λ-term (or a λ-expression) can be defined inductively as:

- Every constant is a λ-expression (called a build-in constant, e.g., numbers).
- Every variable is a λ-expression.
- If M and N are λ-expressions, then MN is a λ-expression. MN is called an *application*, where M is the applicator (or, informally, function definition) and N is the applicant (or actual parameter).
- If M is a λ-expression and x is a variable, then $\lambda x.M$ is a λ-expression. $\lambda x.M$ is called an *abstraction*, in which x is the bound variable (or formal parameter) and M is the function body.
- If M is a λ-expression, then (M) is also a λ-expression.

The major operation in λ-calculus is β-reduction, which substitutes all free occurrences of a formal parameter in the function body with an actual parameter.

Formally, β-reduction can be defined as

$$(\lambda x.E)M \rightarrow E[M/x]$$

where $E[M/x]$ is the expression E with M substituted for free occurrences of x. $E[M/x]$ can be defined inductively as:

- $x[M/x]$ returns M.
- $c[M/x]$, where c is any variable or constant other than x, returns c.
- $(EF)[M/x]$ returns $E[M/x]\ F[M/x]$.
- $(\lambda x.E)[M/x]$ returns $\lambda x.E$ (i.e., x is bound variable).
- $(\lambda y.E)[M/x]$, where y is any variable other than x, returns $\lambda y.E[M/x]$ if x is not free in E or y is not free in M, and returns $\lambda y.(E[z/y])[M/x]$ (where z is a new variable name) if x is free in E and y is free in M.

β-reduction essentially defines the execution of a function application in which the formal parameters are replaced by the actual parameters. Under this interpretation, the λ-abstraction $\lambda x.E$ can be thought as a function, M as an actual parameter, and the β-reduction $(\lambda x.E)M$ as applying the function to an argument. Because of the reduction, a λ-expression of the form $(\lambda x.E)M$ is normally called a *redex* (for *red*ucible *ex*pression). If an expression contains no redexs then evaluation is complete, and the expression is said in *normal form.*

In the practical system, there always exist some built-in functions that can further reduce an expression otherwise considered to be in its normal form. For example,

we cannot apply β-reduction to + 2 3 but the expression can be further reduced to 5 by the definition of +. The operation of these functions constitutes another form of reduction, known as δ-conversion (and the built-in functions as δ-rules). The definition of δ-rules is ad hoc and system dependent; they normally include primitive arithmetic operations or predicates. δ-conversion sometimes constitutes another phase of the execution of a function application in that it reduces the expression after the actual parameters have been substituted into their proper positions via β-reduction. For our purpose, it is convenient to extend the definition of redex to the expression that can be applied by either β-reduction or δ-conversion.

Conventional implementation of functional languages

Since function application (β-reduction) is the only control mechanism for functional languages, it governs the implementation of functional systems. Unlike procedural languages, purely functional languages have no corresponding language-level assignment and sequencing constructs. The implementation has to add necessary instructions to control the order of execution and manage the allocation and deallocation of memory.

From an implementation point of view, function application can be divided into two basic phases: first, the actual parameters are distributed to the positions held by the designated bound variables (i.e., perform a β-reduction); and second, the substituted expression is evaluated by further invoking function application or by applying δ-rules so that the expression is reduced to normal form.

The traditional implementation model for functional languages is based on the

environment model. In this scheme, the system maintains a dynamic symbol-table that contains a mapping from formal parameters to actual parameters. During the first phase, the system establishes a table and puts the actual parameters into the corresponding entries. These values can be retrieved during the evaluation process of the second phase. The major complication of this approach is the resolution of variable names, which comes from the last rule of $E[M/x]$. Because duplicated names are allowed, it is vital to keep track the scope of variables for the correct operation of recursion, non-local access to variables and higher order structures. These complex access schemes greatly complicate the structure of the symbol table, and increase the time required to construct the *closure*. Due to the large overhead for environment operations, the execution of functional programs is quite slow. In order to make the implementation more efficient, it is desirable to eliminate the environment, which leads us to the combinator model discussed in the next section.

Combinators

Combinators are used to describe the general properties of operators and the composition of operators [31] [12]. Although combinators have their own theoretical interest and are equivalent to λ-calculus in terms of the representation power, we primarily employ them as a model to implement functional languages and treat them as a low-level representation of λ-calculus (much like the relation between high-level languages and machine instructions).

Formally, a combinator is a λ-expression that contains no occurrences of free variables [31]. Since there are no free variables, the application of a combinator de-

pends only on the values of the actual parameters and can be treated as a rewrite rule that makes local transformations. For example, consider the λ expression "$S = \lambda fgx.fx(gx)$". S is a combinator since f, g and x are local with respect to $\lambda fgx.fx(gx)$. The corresponding rewrite rule is $Sfgx = fx(gx)$.

With a set of carefully selected combinators, any λ-expression can be transformed into a combinatory expression. There are a variety of ways to construct a combinator set. For example, the simplest set is composed of only two combinators [31], and the complex sets contain an infinite number of dynamically-created combinators [1] [34] [33].

The operation of a combinator-based programming system consists of a compilation phase and an execution phase. In the compilation phase, a λ-expression is transformed to a combinatory expression by abstracting the bound variables. In the execution phase, the expression is evaluated (reduced) according to the corresponding rewrite rules. By the Church-Rosser Theorem [31], sub-expressions of a computation may be evaluated in any order because they will end up with the same result as long as the computation terminates.

In the remainder of this section, we demonstrate the operation of a simple $\{S, K, I\}$-based combinator system. We will also examine the intuition behind these combinators, and then describe Turner's system, which will be used as the basis for our system.

A system based on $\{S, K, I\}$

To demonstrate the operation of combinators, we use the simple $\{S, K, I\}$ combinator set [31] as an example. The definitions of S, K, I are:

$$S = \lambda fgx.f\ x\ (g\ x)$$

$$K = \lambda xy.x$$

$$I = \lambda x.x$$

and their corresponding rewrite rules are:

$$S\ f\ g\ x = f\ x\ (g\ x)$$

$$K\ x\ y = x$$

$$I\ x = x$$

The abstraction algorithm for $\{S, K, I\}$ can be recursively described by three rules [56]. The notation"**A** x $[e]$" denotes abstracting variable x from expression e.

$$\mathbf{A}\ x\ [f\ g] = S\ (\mathbf{A}\ x\ [f])\ (\mathbf{A}\ x\ [g])$$

$$\mathbf{A}\ x\ [c] = K\ c \qquad \text{if } x \text{ does not occur in } c$$

$$\mathbf{A}\ x\ [x] = I$$

The abstraction and execution (application) procedure can be demonstrated by the simple example, $f\ 2$, where $f\ =\ \lambda x.*xx$. The abstraction of f is:

$$\begin{aligned}
\mathbf{A}\ x\ [f] &= \mathbf{A}\ x\ [\lambda x.*xx] \\
&= S\ (\mathbf{A}\ x\ [*\ x])\ (\mathbf{A}\ x\ [x]) \\
&= S\ (S\ (\mathbf{A}\ x\ [*])\ (\mathbf{A}\ x\ [x]\ )\ I \\
&= S\ (S\ (K\ *)\ I)\ I
\end{aligned}$$

Note that the final expression does not contain an occurrence of the variable x. The normal order application of f to 2 is:

$$
\begin{aligned}
f\ 2 &= S\ (S\ (K\ *)\ I\ )\ I\ 2 \\
&= (S\ (K\ *)\ I\ 2)\ (I\ 2) \\
&= ((K\ *\ 2)\ (I\ 2))\ (I\ 2) \\
&= (*\ (I\ 2))\ (I\ 2) \\
&= (*\ 2)\ (I\ 2) \\
&= *\ 2\ 2 \\
&= 4
\end{aligned}
$$

The intuition behind combinators

Although the previous abstraction and rewrite rules are fairly simple, they lack the intuitive clarity of λ-calculus. For example, it is difficult to see that the combinatory expression "$f\ =\ S\ (S\ (K\ *)\ I\ )\ I$" is equivalent to "$f\ x\ =\ x * x$". We will attempt to provide a more comprehensible interpretation in this sub-section.

The key observation in the process of function application is the role played by the bound variables (i.e., formal parameters). They are involved only in the distribution phase and play no roll in the evaluation of the expression. In other words, bound variables only serve as *symbolic place holders* that assist the distribution of actual parameters. Therefore, they can be removed if the actual parameters can be distributed in another way. Combinators are an alternative mechanism to perform this task. Instead of using variables, actual parameter distribution is guided by a sequence of combinator instructions.

One intuitive interpretation for combinators is to treat them as *director strings* [42], that indicate the actual parameters should be distributed to their places in the expression. This method differs from using bound variables, in which the designated

places are explicitly indicated by variables. The difference between the two schemes is similar to how we tell a taxi driver about our destination: we can express it explicitly, such as "1234 Oak street ···", or by a sequence of "direction instructions", such as "turn left, go straight, turn right ···".

The meaning of $\{S, K, I\}$ can be reinterpreted under this concept. The abstraction algorithm can be rewritten as:

- *curry* the function definition and construct a binary tree from the curried function, with the root representing function application (denoted by "."), the left child as the applicator, and right child as the applicant.
- annotate the nodes according to the following method:
  - if a node represents function application, annotate it with S.
  - if a node is same as the variable being abstracted, replace it with I.
  - otherwise, annotate it with K.
- "list" the tree, with additional parentheses for each subtree, by pre-ordered traversal.

The execution of the rewrite rules also can be interpreted under this concept. When an actual parameter arrives, it is distributed according to the annotation (combinator direction) of the node:

- if the tag is S, redistribute the parameter to the two child branches.
- if the tag is K, ignore the parameter.

- if the tag is I, replace this node with the parameter.

After the parameters are distributed, reduction can proceed according the δ-rule of the operator. Using the previous example of f 2, the process of abstracting x from f is shown in Figure 2.1, and the application of f to 2 is shown in Figure 2.2.

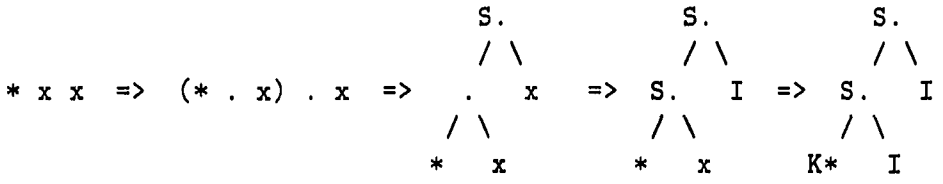


Figure 2.1: Abstraction of x from $* \; x \; x$

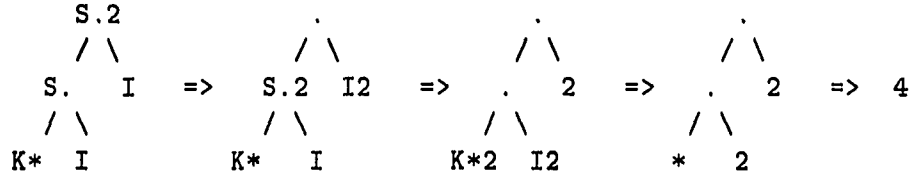


Figure 2.2: Application of $S \; (S \; (K \; *) \; I) \; I$ to 2

Under the new interpretation, the meaning behind the abstraction and rewrite rules is more clear: S is the *distributor*, which distributes the actual parameter to the two child branches; I is the *identity function*, which reserves a space for the actual parameter; and K is the *eliminator* that eliminates the arrived actual parameter. During the function application, the actual parameter is sent to its place(s) through the tree via the guidance of the $\{S, K, I\}$ set; after the parameter reaches the designated positions, the expression tree can be evaluated accordingly.

Regular Combinators

$$I\ x = x \qquad B\ f\ g\ x = f\ (g\ x)$$
$$K\ x\ y = x \qquad B1\ k\ f\ g\ x = k\ (f\ (g\ x))$$
$$S\ f\ g\ x = f\ x\ (g\ x) \qquad C\ f\ g\ x = f\ x\ g$$
$$S1\ k\ f\ g\ x = k\ (f\ x)\ (g\ x) \qquad C1\ k\ f\ g\ x = k\ (f\ x)\ g$$

List-Oriented Combinators

$$Sp\ f\ g\ x = (f\ x) : (g\ x)$$
$$Bp\ f\ g\ x = f : (g\ x)$$
$$Cp\ f\ g\ x = (f\ x) : g$$

Figure 2.3: Definitions of Turner's Original Combinators

Turner's combinator system

Although the previous $\{S, K, I\}$-based system is simple and elegant, it is very inefficient because the size of the compiled combinator expression becomes unacceptably large ($O(2^N)$ complexity) and may require a large number of reductions during the evaluation. In late 1970s, Turner introduced several new combinators, $B, C, S1, B1, C1$ (and Sp, Bp, Cp for list operation) and a set of optimization rules for these combinators. His scheme dramatically improves the complexity to $O(N \log N)$ on average, and makes combinator-based systems feasible. The rewrite rules and the corresponding abstraction algorithm are shown in Figure 2.3 and Figure 2.4 respectively. The major extension over the $\{S, K, I\}$-based system is the additional optimization phase in the compiling algorithm. This phase employs a set of optimization rules that recognize predetermined patterns in the combinator expression and reduces it to a simpler form. In the other words, the notation **O** in Figure 2.4 can be thought as a function with combinator expressions as its domain and range.

Abstraction Rules

$$\begin{aligned}
&\mathbf{A}\ x\ [hd : tl] \ =\ \mathbf{O}\ [Sp\ (\mathbf{A}\ x\ [hd])\ (\mathbf{A}\ x\ [tl])] \\
&\mathbf{A}\ x\ [f\ g] \ =\ \mathbf{O}\ [S\ (\mathbf{A}\ x\ [f])\ (\mathbf{A}\ x\ [g])] \\
&\mathbf{A}\ x\ [x] \ =\ I \\
&\mathbf{A}\ x\ [y] \ =\ K\ y \qquad \text{if } x \text{ does not occur in } y
\end{aligned}$$

Optimization Rules for Regular Combinators

$$\begin{aligned}
&\mathbf{O}\ [S\ (K\ f)\ (K\ g)] = K\ f\ g \\
&\mathbf{O}\ [S\ (K\ f)\ I)] = f \\
&\mathbf{O}\ [S\ (K\ f)\ (B\ g\ h)] = B1\ f\ g\ h \\
&\mathbf{O}\ [S\ (K\ f)\ g] = B\ f\ g \\
&\mathbf{O}\ [S\ (B\ f\ g)\ (K\ h)] = C1\ f\ g\ h \\
&\mathbf{O}\ [S\ f\ (K\ g)] = C\ f\ g \\
&\mathbf{O}\ [S\ (B\ f\ g)\ h)] = S1\ f\ g\ h
\end{aligned}$$

Optimization Rules for List-Oriented Combinators

$$\begin{aligned}
&\mathbf{O}\ [Sp\ (K\ f)\ (K\ g)] = K\ (f : g) \\
&\mathbf{O}\ [Sp\ (K\ f)\ g] = Bp\ f\ g \\
&\mathbf{O}\ [Sp\ f\ (K\ g)] = Cp\ f\ g
\end{aligned}$$

$\mathbf{A}\ x\ [f]$: Abstracting x from f.
$\mathbf{O}\ [e]$: Optimizing expression e.

Figure 2.4: Turner's Original Abstraction Algorithm

The actual implementation of Turner's system can be explained by its code representation and its evaluator. In this system, a combinatory expression is represented by a binary tree with the root denoting the function application, the left child as the applicant, and the right child as the applicator. Because the left branch chain of application nodes has special significance, it is called a *spine*. Similarly, the node at the end of the spine is called a *tip*, and the expressions along the spine are *ribs*. With this kind representation, the operation of the evaluator is fairly simple and can be described by the following pseudo code:

```
while TRUE
  begin
    while (current node is not tip)
       begin
          make left branch the current node;
       end
    if (tip is not a redex)
      return current subgraph; EXIT;
    if (the strict arguments have not been evaluated)
      evaluate the corresponding ribs;
    modify the graph corresponding to rewrite rules or built-in functions;
  end
```

Despite its simplicity, this evaluator implicitly performs lazy evaluation; i.e., it evaluates only the actual parameters when their values are needed rather than always evaluating them when the function is applied. Furthermore, it evaluates the actual parameters at most once. The final results of this evaluator is not in normal form since the non-strict branches are not required to be evaluated and may contain some redexs. Instead, the evaluator only returns an expression that contains no top-level redex, which sometimes is known as *weak head normal form* [37]. Since the remainder of thesis is discussed in this "lazy" context, for convenience, we will use the term

"normal form" to represent the term "weak head normal form".

Turner's scheme provides a simple but effective way to use combinators to implement functional languages. In subsequent chapters, we will extend Turner's combinators for use in a multiprocessor environment.

Literature Review

In this section, we will give a brief overview of two relevant topics, namely combinator-based functional systems, and non-combinator-based functional multiprocessor systems.

Functional systems based on the combinator paradigm

Combinators were introduced by Schöfinkel in the early 1920s and re-discovered later by Curry. Although their theory is well understood [12] [31], their application as an abstract implementation model for functional languages was not introduced until late 1970s. Turner first utilized this idea and implemented a virtual system using the $\{S, K, I, B, C\}$ combinator set [56] [55]. Since then, the combinator paradigm has been widely used in the implementation of functional languages.

Combinator sets vary widely, but can be divided into three basic groups. The first group contains only a finite number of combinators. The second group is composed of an infinite number of combinators that are normally divided into several indexed families. The third group is compromised of dynamically created combinators whose rewrite rules are defined by the program expressions.

Most sets in the first group are variations or extensions of Turner's set [38]

[24]. Because of their simplicity, it is possible to incorporate the reduction rules into the hardware of the machine as part of its instruction set. There are several machines built around these sets: SKIM [11] and its successor SKIM II [54] employ conventional bit-slice logic; NORMA [52] utilizes more sophisticated hardware which includes specialized service processor, graph processor, allocator, and graph memory; and CURRY [50] implements the entire reduction engine in a VLSI chip.

A slightly different scheme known as director strings employs director symbols $\{\hat{},\backslash,/,-\}$ as its basic combinator set [42]. Despite its appearance, it is based on the concept similar to Turner's long-reach combinators. A message-passing multiprocessor system, COBWEB [4] [6], has been built to support this scheme. An additional combinator P is used to control eager evaluation [23].

A major problem with finite combinator sets is their poor efficiency. For a λ-expression of length n, the length of compiled combinatory code is $O(n^2)$ in the worst case [38], and $O(n^{1.5})$ on the average [28]. While the required space can be reduced by special coding techniques [47] [48], there is no effective way to decrease the number of reduction steps. Several empirical studies [26] [51] suggest that there is considerable run-time overhead in using a finite combinator set.

The sets in the second and third groups are aimed at constructing more compact $O(n \log n)$ combinator expressions. Sets in the second group extend simple combinators into "long-reach" ones that can operate on any abstracted variable. Abdali [1] introduced a combinator set containing three families which extend the K, I and B into the K_n, I_n^m and B_n^m families. Knox [45] further expanded it to include the S_n^m, C_n^m, $X_n^{m,k}$ and $Y_n^{m,k}$ families for additional optimization. Castan et al. [8] proposed

a different set comprising $\{K_n, M_n, N_n, Q_n, W_n\}$ and claimed that parameters can be distributed faster. Because of the regularity of these sets, they can be implemented in the hardware as machine instructions. However, since a CPU implements a finite instruction set, the maximal n is restricted. The MaRS computer [9] [46] [10] is a message-passing multiprocessor system based on the Castan's set.

Combinator sets in the third group do not have fixed combinators. Instead, the abstraction algorithm creates the required combinators during the compilation. Hughes [34] first proposed a technique to create the required combinators (called as *super combinators*) from functional programs. Hudak and Goldberg [33] extended it to *serial combinators*, which take into account some pragmatic issues, and claims that they have "optimal" granularity. Due to the dynamic nature of this approach, sets in this group are not feasible for hardware implementation and therefore are used as an intermediate form during code generation. Serial combinators have been used to generate code for the G-machine [36], an abstract graph reduction processor. The G-machine is implemented in hardware [43] and reportedly can achieve 3.6 MIPS performance [44]. Several multiprocessor systems also employ super combinators, including GRIP [49] and Flagship [59]. Serial combinators have also been simulated on a message-passing multiprocessor system (DAPS) [33], and implemented in Alfalfa [18], a hypercube system based on the Intel iPSC.

Functional multiprocessor systems based on other paradigms

There are several other approaches for building a functional or a quasi-functional programming system, including a non-combinator-based graph-reduction paradigm,

the environment paradigm, and the dataflow paradigm. Since these approaches cover a wide variety of systems, our literature review is limited to multiprocessor systems. A more general review on sequential systems can be found in [58] [32] .

The non-combinator-based graph-reduction paradigm employs the basic graph reduction concept, but without the assistance of combinators [37]. ALICE [13] [25] is the first parallel system based on graph reduction. It is a shared-memory system connected by a crossbar switch and two rings. Rediflow [40] [39] is a message-passing system connected as mesh. A pressure gradient load balancing mechanism is used to distribute the tasks. Its successor, Rediflow II, adopts the PSCED architecture (Process, Stack, Control, Environment and Dump) for its reduction processor and employs a more sophisticated communication unit called Redilink [41].

The environment paradigm is used in LISP systems [3]. Since most LISP systems contain constructs that can cause side-effects, they are at best only quasi-functional systems. SPUR [29] is a shared-memory system targeted for Common LISP. It contains 6 to 12 specialized RISC processors that provide high-level support for LISP constructs. The Multilisp [20] [21] system employs a special control/synchronization construct known as a *future*. Multilisp has been implemented on the Concert system [19], which is a shared-memory architecture with 28 MC68000 processors connected by a Ringbus. A more sophisticated processor, known as MASA (Multilisp Architecture for Symbolic Applications) [22], also has been proposed. The Connection machine [30] is a message-passing system containing up to 64,000 single bit processors. It is a SIMD machine and is aimed at structured symbolic operations.

Dataflow systems are based on the data-driven model. Despite their syntactic

differences, dataflow languages and functional languages are semantically equivalent. However, since data flow systems are normally targeted at numerical applications and lack a general, list-like structure, they are not appropriate for most symbolic computing applications.

Thesis Problem Statement

The effort of this research is towards developing an effective mechanism to control parallelism for Turner's combinator system. To achieve our goal, we will extend the original scheme by augmenting combinators with *eagerness information* to control concurrent execution, and then evaluate the its efficiency through simulation of an idealized shared-memory multiprocessor system. The major differences between our approach and the COBWEB system are: first, we use Turner's original compiling scheme so that we can take full advantage of its optimization phase; and second, we focus more on a shared-memory based multiprocessor system rather than on a message-passing system.

Despite the elegant design of Turner's system, it is primarily designed for sequential execution. As we observed in the previous section, the evaluator is sequential and is fully-lazy; it will not initiate the evaluation of the rib until its value is required. Although this works well in a sequential system, it is unacceptable for a multiprocessor system where the speedup depends on the early initiation of the execution of subtasks. An alternative way is to make the evaluator *eager*; i.e., initiate all the ribs encountered in the unwinding of the spine. This approach is sketched out below:

```
while TRUE
  begin
```

```
while (current node is not tip)
   begin
      initiate the evaluation of rib;
      make left branch the current node;
   end
if (tip is redex)
   return the subgraph; EXIT;
if (the evaluation of the strict arguments have not been completed)
    wait for completion;
modify the graph by corresponding rewrite rules or built-in functions;
end
```

Although the modified evaluator is simple, it is not satisfactory for several reasons. First, since the evaluator initiates all the reducible expressions, it introduces some useless computation and tends to waste system resources. Second, and more importantly, it overrides SASL's lazy semantics and may introduce non-terminated computation. Because many fundamental constructs of SASL, such as stream and infinite data structures, depend on laziness, unconditional eager evaluation is unacceptable.

A hybrid approach would seem appropriate, where one can initiate eager evaluation only on *selected expressions* and can preserve the lazy semantics for other expressions. In other words, we need to add a new mechanism that can specify which expression is to be eagerly evaluated and perform eager evaluation during the execution accordingly. To accomplish this goal, we extend Turner's work in two ways. First, we augment SASL and the combinator set to incorporate parallelism control information, and we expand the abstraction algorithm to translate and propagate this new information. Second, we modify the virtual machine layer from a single processing system to a multiprocessor system that contains multiple reduction units.

The scope of this study can be best explained by Figure 2.5 in which Turner's

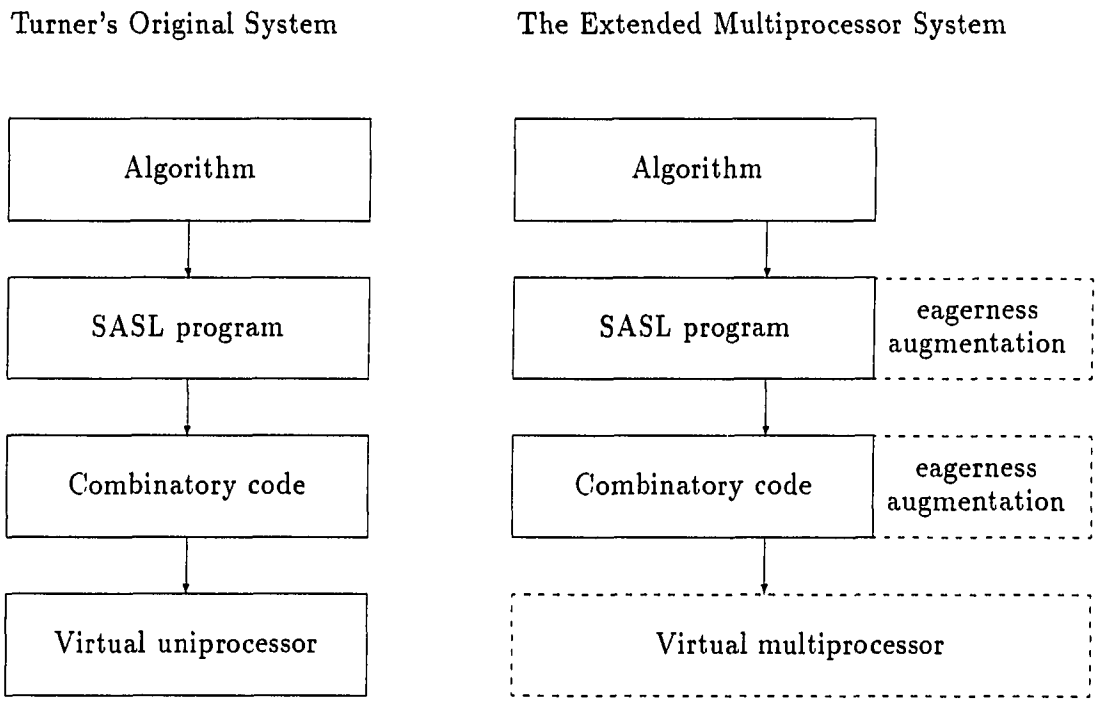


Figure 2.5: The Layer Model for the Original and the Extended System

original system and the proposed extensions are shown. Turner's original scheme can be thought as a programming system composed of four layers: problem description (or algorithm), high-level language (or program), combinatory code, and the virtual machine. The program layer is the functional language SASL, and the combinatory code layer consists of Turner's basic S, K, I, B, C combinators and built-in operators. The combinatory code is executed in an idealized sequential reduction machine represented by the virtual machine layer. Turner's compiling algorithm is used to translate SASL to combinatory machine code.

In the extended system, the program layer and the combinatory code layer will be similar to those of Turner's original system except that they are augmented with parallelism-controlling constructs that guide the concurrent execution. The virtual multiprocessor layer represents an idealized system which contains an unlimited number of reduction processors and a global memory with infinite bandwidth.

Although the combinator paradigm is not limited to a special class of computations, our studies will concentrate only on symbolic-oriented applications; i.e., those applications whose major computations are dedicated to sophisticated searching and matching algorithms, and to the construction and manipulation of list-structured data. This class of applications differs dramatically from the conventional numerical-oriented applications in that while the later is more structured and predictable and involves only uniform array structures, the former shows less regularity, tends to be data-driven and contains lots of "environment manipulation" (such as function call/return, suspension/resumption, and recursion). Because of these run-time characteristics, extracting and controlling the fine-grain parallelism in symbolic applications is difficult.

CHAPTER 3. EXTENDING THE SASL SYSTEM FOR PARALLEL PROCESSING

As discussed in the previous section, the goal of extending the SASL system was to develop a control mechanism that can initiate eager evaluation on selected expressions. To achieve this goal, we added control information to the original SASL system to override the lazy semantics. This new information was added to both the program layer and the combinator layer. Also, the compiling algorithm was expanded so that it can propagate this information to the remainder of the system. In summary, to expand the original SASL system so that it utilizes multiple processes, three major extensions were made:

- add extra control information to selectively initiate eager evaluation by annotating SASL programs.
- employ a tagged-node scheme in the combinators to incorporate this control information.
- extend Turner's compiling algorithm to propagate this control information from SASL programs to the combinatory code.

Because list manipulation is an essential ingredient of functional programs, these modifications cover both function applications and list constructions. The following

sections describe the three extensions in detail.

Annotating SASL Programs

The extended SASL language needs a new construct to specify selective initiation of eager evaluation. Because our system targets fine-grain parallelism, this construct should be kept simple and should not introduce any complicated synchronization or communication. Although this is difficult for conventional multiprocessor systems, it can be easily accomplished in a functional language-based multiprocessor system because the execution order is largely irrelevant. According to the Church-Rosser Theorem, expressions evaluated under different orders will converge to the same normal form provided the evaluations terminate. Thus, instead of specifying detailed control information, functional languages need only to provide a notation to indicate how the corresponding expressions are evaluated.

In our extended system, the symbol "@" is employed in SASL to specify the additional control information. The modified SASL language is the same as the standard version except that an application may be annotated as being eager by placing "@" in front of it. Note that list constructions can be annotated as well as function applications.

Annotating function application

The annotation of a function application is similar to the scheme proposed by Burton [7], although our scheme is modified to fit the syntax of SASL. Here, each function application in the expression will be eager (using a "'@") or lazy (by de-

fault). Eager notation will force the simultaneous evaluation of the applicator (function body) and applicant (argument), and hence we can speed up the execution of the program provided the applicant value is eventually required. Consider the evaluation of $f\ x$. In the lazy mode, the evaluation of x will not be initiated unless the evaluation of f is completed and the value of x is required. On the other hand, if the application $f\ x$ is annotated as eager (i.e., @ $(f\ x)$), the evaluations of f and x will be initiated simultaneously. Because expressions in SASL are written in curried form, this scheme can be easily incorporated into SASL's original framework. For example, consider $f\ 3\ 4$, where

$$f\ x\ y\ =\ ((if\ x > y\ then\ *\ else\ +)@(*\ x\ x))@(*\ y\ y)$$

The applications inside the expression $(if\ x > y\ then\ *\ else\ +)$, $(*\ x\ x)$ and $(*\ y\ y)$ are lazy, but the applications between them are eager. The effect of the eager annotations change the evaluation order from

$$(f\ 3\ 4)\ \Rightarrow\ ((if\ FALSE\ then\ *\ else\ +)\ (*\ 3\ 3))\ (*\ 4\ 4) \Rightarrow (+\ (*\ 3\ 3))\ (*\ 4\ 4)$$
$$\Rightarrow\ (+\ 9)\ (*\ 4\ 4) \Rightarrow (+\ 9)\ 16 \Rightarrow 25$$

to

$$(f\ 3\ 4) \Rightarrow ((if\ FALSE\ then\ *\ else\ +)\ (9)\ (16) \Rightarrow (+\ 9)\ 16 \Rightarrow 25$$

Because the annotation specifies that some expressions can be eagerly evaluated, their evaluations are invoked earlier and thereby speed up the entire computation.

Annotating list construction

Evaluating a function over non-flat domains (such as list) differs from that over flat domains (such as integer, boolean, etc.). While the later always returns a value in normal form, the former can be evaluated to several different "degrees" [2]. Therefore, the evaluation of a list expression needs to specify not only *how* the expression should be evaluated (i.e., eager or lazy), but *how much* of the expression should be evaluated. Consider the following example:

```
map f () = ()
map f l  = f (hd l) : map f (tl l)
sq x = x*x
testlist = map sq (2,3,4,5)
```

According to the "degrees" of evaluation, `testlist` can be evaluated to various forms:

- $sq\ 2 : map\ sq\ (3,4,5)$
- $4 : map\ sq\ (3,4,5)$
- $(sq\ 2, sq\ 3, sq\ 4, sq\ 5)$
- $(4,9,16,25)$

The possible results are ordered from fully-lazy to eager so that the first is the laziest, the second evaluates only the head, the third constructs the entire list but does not evaluate its elements, and the last evaluates everything. The required degree of evaluation of `testlist` is determined by the function that eventually uses its value. For example, consider the following functions:

```
islist x
|| return TRUE if x is a list

hd x
```

```
|| return the head of the list x

length () = 0
length x  = 1 + length (tl x)
|| return the length of the list x

sumlist () = 0
sumlist x  = (hd x) + sumlist (tl x)
|| return the summation of the elements of list x
```

During the evaluation, the four functions require a different amount of information from the input list `x`. While `islist` needs only to find out whether `x` is a list, `sumlist` needs to know the value of every element of `x`. If we apply `testlist` to the four functions, the previous four forms of `testlist` just provide a proper amount of information.

Our annotation scheme for list operations is to specify the control information when the list is constructed. This information contains two major components: whether the head should be eagerly evaluated, and whether the tail should be eagerly constructed. This is denoted by using "@" in the head and tail fields respectively. We can now control the list evaluation to the desired extent by using the proper combination of the annotations. Using the previous `testlist` as an example, its four results can be generated as follows:

- `map f l = f (hd l) :  map f (tl l)`
- `map f l = @(f (hd l)) :  map f (tl l)`
- `map f l = (f (hd l)) :  @(map f (tl l))`
- `map f l = @(f (hd l)) :  @(map f (tl l))`

Using the annotating scheme

The proposed annotation scheme provides only a mechanism to identify specific expressions to be eagerly evaluated; it is the programmer's task to decide how to use it to achieve the desired effect. There are two important concerns that must be considered, namely the correctness of the final result, and the resources required for evaluation. According to the Church-Rosser Theorem, an expression evaluated in different orders returns the same normal form provided the evaluations terminates. Thus, changing an expression from lazy to eager evaluation will still yield the correct if the computation terminates. However, it should be noted that it is possible that under eager evaluation, a computation that would not otherwise be needed will terminate in error or will not terminate. The annotation is known to be *safe* if it does not introduce non-termination.

The second issue concerns the computational resources needed to evaluate an expression. Eager evaluation may waste computation resources by introducing computations whose results are not needed. The annotation is known as *conservative* if it does not introduce any unwanted computation; otherwise, the annotation is known as *speculative.*

Depending on the correctness and resources used, the effect of annotating a computation can be classified as being unsafe, speculatively safe and conservatively safe. The most desirable annotation is conservatively safe because it initiates the required evaluation earlier and therefore may decrease the total computation time. A good application of this is to annotate strict arguments as eager since their values are eventually required. For example, $f\ x\ =\ g\ x\ +\ h\ x$ can be annotated as

$f\ x\ =\ @\ (g\ x)\ +\ @\ (h\ x)$.

In some cases, it may be desirable to use extra resources in exchange for increased computation speed. To do this, the program is annotated as speculatively safe. Consider the following example

$$f\ x\ y\ =\ if\ @\ (x > y)\ then\ @\ (*\ x\ x)\ else\ @\ (*\ y\ y)$$

With no annotation, the execution first evaluates $x > y$ and then, depending the result of $x > y$, initiates the evaluation either $*\ x\ x$ or $*\ y\ y$. The annotated version forces $x > y$, $*\ x\ x$ and $*\ y\ y$ to be evaluated simultaneously, and will return the result once the evaluation of $x > y$ is completed, assuming that the evaluations of $>$ and $*$ take same time. The annotated version is clearly faster; however, it introduces some unnecessary calculation since both branches of the if are evaluated but only one result is required. Thus, there is a trade-off between the speedup and computational resources consumed.

The unsafe annotation means the eager evaluation may introduce non-termination in an otherwise well-behaved expression. Furthermore, it completely changes the behavior of the program. Although it is possible to obtain maximum speedup, introducing an unsafe annotation in program is dangerous unless the programmer has complete control over the range of input data. Consider the following two examples:

$$f\ x = if\ @\ (x > 0)\ then\ @\ (1/x)\ else\ @\ (x * x)$$

$$g\ n = n\ :\ @\ (g\ (n + 1))$$

Both annotated versions alter the original lazy semantics. In the first example, $f\ 0$

fails to return a value because of the division 1/0. The second example never terminates under any input data because tries to construct an infinite list.

There is no optimal way to annotate a program. It depends on the desired speedup, the available resources, and even the range of input data. Although some abstract interpretation techniques [2] can be applied in pre-processing (e.g., strictness analysis) to assist the annotation, it can only detect partial information because of the undecidability of programs. It is still up to the programmer to find the "best" annotation.

Annotating Combinators

Since additional control information is added to SASL programs, the run-time evaluator should be modified to incorporate this information. Our approach is to add a control tag to the regular combinators that can override the original lazy semantics and eagerly initiate some specified expressions. The modified evaluator becomes:

```
while TRUE
  begin
    while (the node is not tip)
       begin
          make left branch the current node;
       end
    if (tip is not a redex)
       return current expression; EXIT;
    if (the strict arguments have not been initiated)
       evaluate the corresponding ribs;
    if (the evaluation of the strict arguments have not been completed)
       wait for completion;
    modify graph by corresponding rewrite rules or built-in functions;
    if (tip is a combinator with control tag)
       initiate the evaluation of desired expressions accordingly
  end
```

The major modification over the original evaluator is the extra `if` step in the end. It may initiate one or more new processes and distribute the computation to different processors. Also, an extra `if` statement (the third one) is added to test the possibility of early initiation.

The role of the control tag can be best explained by an example. Consider the combinator S, where $S\ f\ g\ x\ =\ f\ x\ (g\ x)$. For demonstration purposes, we denote application by an explicit "." and present the expression in a fully parenthesized fashion. In this notation, S can be rewritten as $S\ f\ g\ x\ =\ ((f.x).(g.x))$. We first explain the operation of the original lazy semantics and then show how the tag can be added to force the desired eager evaluation.

Note that the reduction of $S\ f\ g\ x$ introduces three new applications, namely the application of x to f, or $(f.x)$, the application of x to g or $(g.x)$, and the application of $(g.x)$ to $(f.x)$, or $((f.x).(g.x))$. Using lazy semantics, the evaluation of $S\ f\ g\ x$ proceeds as follows:

1. initiate the evaluation of the third application $((f.x).(g.x))$.

2. since the value of function body $(f.x)$ is unknown, suspend the current evaluation and initiate the evaluation of $(f.x)$.

3. $(f.x)$ returns a result, say h; resume the evaluation of the third application, now $h.(g.x)$.

4. if h is a non-strict function, continue the evaluation of $h.(g.x)$ and return the final answer.

5. if h is a strict function (i.e., the value of $(g.x)$ is required), suspend the evaluation

of $h.(g.x)$ and initiate the evaluation of $(g.x)$; after $(g.x)$ returns its result, say y, resume the evaluation of the third application, now $(h.y)$, and return the result.

In summary, the evaluation can be divided into three steps: first, evaluate $f.x$ (and return h); second, if h is strict, evaluate $g.x$ (and return y); third, evaluate $h.y$ or $h.(g.x)$.

Now let us discuss the opportunity for eager evaluation. The first step is eagerly evaluated by the original lazy semantics and there is no need to add a new tag. The third step can not be evaluated immediately since its initiation depends on the first step. Thus, the only possibility is the second step. If we know (or guess) that the result of $(f.x)$ is strict, we can initiate the evaluation of $(g.x)$ earlier by marking it as eager and thereby evaluate the first and second steps in parallel. In other words, the operating semantics of S is now changed from $((f.x).(g.x))$ to $((f.x).@(g.x))$ We will use a tag to represent this change. It will be denoted as a subscript to the original combinator. For convenience, we will also employ angle brackets "$\langle\rangle$" to denotes the eager evaluation of the expression in the bracket. For example, the "eager" S can now be written as S^r, and its rewrite rules as $S^r\ f\ g\ x\ =\ (f\ x)\ \langle g\ x\rangle$.

We can apply the same idea to other combinators by attaching proper control tags. The detailed definitions these combinators are shown in Figure 3.1. Like the tag in S, they instruct how to evaluate the newly created applications after the corresponding combinator reduction. Because some combinators are more complicated and introduce more applications in their rewrite rules, there may be multiple expressions that can be eagerly evaluated. For example, consider $S1$, where

Regular Combinators

$$
\begin{array}{ll}
I\ x = x & B\ f\ g\ x = f\ (g\ x) \\
K\ x\ y = x & B^r\ f\ g\ x = f\ \langle g\ x \rangle \\
S\ f\ g\ x = f\ x\ (g\ x) & B1\ k\ f\ g\ x = k\ (f\ (g\ x)) \\
S^r\ f\ g\ x = f\ x \langle g\ x \rangle & B1^r\ k\ f\ g\ x = k\ (f\ \langle g\ x \rangle) \\
S1\ k\ f\ g\ x = k\ (f\ x)\ (g\ x) & B1^l\ k\ f\ g\ x = k\ \langle f\ (g\ x) \rangle \\
S1^r\ f\ g\ x = k\ (f\ x)\ \langle g\ x \rangle & B1^{lr}\ k\ f\ g\ x = k\ \langle f\ \langle g\ x \rangle \rangle \\
S1^l\ f\ g\ x = k\ \langle f\ x \rangle\ (g\ x) & C\ f\ g\ x = f\ x\ g \\
S1^{lr}\ f\ g\ x = k\ \langle f\ x \rangle\ \langle g\ x \rangle & C1\ k\ f\ g\ x = k\ (f\ x)\ g \\
 & C1^r\ k\ f\ g\ x = k\ \langle f\ x \rangle\ g
\end{array}
$$

List-Oriented Combinators

$$
\begin{array}{ll}
Sp\ f\ g\ x = (f\ x) : (g\ x) & Bp\ f\ g\ x = f : (g\ x) \\
Sp^h\ f\ g\ x = \langle f\ x \rangle : (g\ x) & Bp^t\ f\ g\ x = f : \langle g\ x \rangle \\
Sp^t\ f\ g\ x = (f\ x) : \langle g\ x \rangle & Cp\ f\ g\ x = (f\ x) : g \\
Sp^{ht}\ f\ g\ x = \langle f\ x \rangle : \langle g\ x \rangle & Cp^h\ f\ g\ x = \langle f\ x \rangle : g
\end{array}
$$

Figure 3.1: Definitions of the Extended Combinators

$S1\ k\ f\ g\ x = k\ (f\ x)\ (g\ x)$. Since there are two applications that can be eagerly evaluated, the tag must contain two bits of information, and the corresponding rewrite rule is expanded to four rules:

$$S1\ k\ f\ g\ x = k\ (f\ x)\ (g\ x)$$
$$S1^{r}\ k\ f\ g\ x = k\ (f\ x)\ \langle g\ x\rangle$$
$$S1^{l}\ k\ f\ g\ x = k\ \langle f\ x\rangle\ (g\ x)$$
$$S1^{lr}\ k\ f\ g\ x = k\ \langle f\ x\rangle\ \langle g\ x\rangle$$

The subscripts l and r represent that the "left" and "right" expressions can be eagerly evaluated respectively.

The list-oriented combinators can be extended in similar way. In SASL, a list structure contains two major fields: the head, which is the first element of the list, and the tail, which is also a list (more precisely, a pointer to a list) containing the rest of the list. The list-oriented combinators construct a list by specifying the list's head and tail respectively. The combinator in conjunction with the control tags determines how to construct the corresponding fields. For example, consider Sp, where $Sp\ f\ g\ x = (f\ x) : (g\ x)$. By the lazy semantics, the reduction of Sp only constructs the list and does not evaluate its head or tail. As we have seen in previous examples, it is sometimes advantageous to evaluate the list to certain degree during construction if it is known that the the values will be needed. The new control tags allow us to override the default lazy semantics and to force the specified field to be

eagerly constructed. The rewrite rules for *Sp* can now be expanded into four rules:

$$Sp\ f\ g\ x = (f\ x) : (g\ x)$$
$$Sp^h\ f\ g\ x = \langle f\ x\rangle : (g\ x)$$
$$Sp^t\ f\ g\ x = (f\ x) : \langle g\ x\rangle$$
$$Sp^{ht}\ f\ g\ x = \langle f\ x\rangle : \langle g\ x\rangle$$

Here, h denotes the evaluation of the head, and t the tail. The extended rewrite rules of other list-oriented combinators are shown in Figure 3.1.

Extended Compiling Algorithm

The previous two sections introduce mechanisms to add extra control information into SASL programs and combinators respectively. In this section, we will discuss how the compiling algorithm was extended to propagate the new control information from SASL programs to combinatory code.

Recall that Turner's original compiling algorithm consists of the abstraction phase and optimization phase, each of which contains a set of transforming rules as shown in the Figure 2.4. To propagate the new control information, we add a set of optimization rules for the tagged combinators. The goals of our extension are twofold: first, to keep Turner's original optimization rules (to minimize the number of reductions); second, to preserve all the control information from SASL programs. The complete extended abstraction algorithm is shown in Figure 3.2. Note that whereas there is only one extra rule in the abstraction phase, there is substantial modification in the optimization phase.

Abstraction Rules

$$\mathbf{A}\ x\ [@\ f] = @\ (\mathbf{A}\ x\ [f])$$
$$\mathbf{A}\ x\ [hd : tl] = \mathbf{O}\ [Sp\ (\mathbf{A}\ x\ [hd])\ (\mathbf{A}\ x\ [tl])]$$
$$\mathbf{A}\ x\ [f\ g] = \mathbf{O}\ [S\ (\mathbf{A}\ x\ [f])\ (\mathbf{A}\ x\ [g])]$$
$$\mathbf{A}\ x\ [x] = I$$
$$\mathbf{A}\ x\ [y] = K\ y \qquad \text{if } x \text{ does not occur in } y$$

Optimization Rules for Regular Combinators

$$\mathbf{O}\ [S\ (K\ f)\ (K\ g)] = K\ f\ g$$
$$\mathbf{O}\ [S\ (K\ f)\ @\ (K\ g)] = K\ f\ @\ g$$
$$\mathbf{O}\ [S\ (K\ f)\ I)] = f$$
$$\mathbf{O}\ [S\ (K\ f)\ @\ I)] = f$$
$$\mathbf{O}\ [S\ (K\ f)\ (B\ g\ h)] = B1\ f\ g\ h$$
$$\mathbf{O}\ [S\ (K\ f)\ (B^r\ g\ h)] = B1^r\ f\ g\ h$$
$$\mathbf{O}\ [S\ (K\ f)\ @\ (B\ g\ h)] = B1^l\ f\ g\ h$$
$$\mathbf{O}\ [S\ (K\ f)\ @\ (B^r\ g\ h)] = B1^{lr}\ f\ g\ h$$
$$\mathbf{O}\ [S\ (K\ f)\ g] = B\ f\ g$$
$$\mathbf{O}\ [S\ (K\ f)\ @\ g] = B^r\ f\ g$$
$$\mathbf{O}\ [S\ (B\ f\ g)\ (K\ h)] = C1\ f\ g\ h$$
$$\mathbf{O}\ [S\ (B^r\ f\ g)\ (K\ h)] = C1^r\ f\ g\ h$$
$$\mathbf{O}\ [S\ (B^r\ f\ g)\ @\ (K\ h)] = C1^r\ f\ g\ @\ h$$
$$\mathbf{O}\ [S\ (B\ f\ g)\ @\ (K\ h)] = C1\ f\ g\ @\ h$$
$$\mathbf{O}\ [S\ f\ (K\ g)] = C\ f\ g$$
$$\mathbf{O}\ [S\ f\ @\ (K\ g)] = C\ f\ @\ g$$
$$\mathbf{O}\ [S\ (B\ f\ g)\ h)] = S1\ f\ g\ h$$
$$\mathbf{O}\ [S\ (B^r\ f\ g)\ h)] = S1^l\ f\ g\ h$$
$$\mathbf{O}\ [S\ (B\ f\ g)\ @\ h)] = S1^r\ f\ g\ h$$
$$\mathbf{O}\ [S\ (B^r\ f\ g)\ @\ h)] = S1^{lr}\ f\ g\ h$$
$$\mathbf{O}\ [S\ f\ @\ g] = S^r\ f\ g$$

Optimization Rules for List-Oriented Combinators

$$\mathbf{O}\ [Sp\ (K\ f)\ (K\ g)] = K\ (f : g)$$
$$\mathbf{O}\ [Sp\ @\ (K\ f)\ (K\ g)] = K\ (@\ f : g)$$
$$\mathbf{O}\ [Sp\ (K\ f)\ @\ (K\ g)] = K\ (f : @\ g)$$
$$\mathbf{O}\ [Sp\ @\ (K\ f)\ @\ (K\ g)] = K\ (@\ f : @\ g)$$
$$\mathbf{O}\ [Sp\ (K\ f)\ g] = Bp\ f\ g$$
$$\mathbf{O}\ [Sp\ (K\ f)\ @\ g] = Bp^t\ f\ g$$
$$\mathbf{O}\ [Sp\ @\ (K\ f)\ g] = Bp\ @\ f\ g$$
$$\mathbf{O}\ [Sp\ @\ (K\ f)\ @\ g] = Bp^t\ @\ f\ g$$
$$\mathbf{O}\ [Sp\ f\ (K\ g)] = Cp\ f\ g$$
$$\mathbf{O}\ [Sp\ @\ f\ (K\ g)] = Cp^h\ f\ g$$
$$\mathbf{O}\ [Sp\ f\ @\ (K\ g)] = Cp\ f\ @\ g$$
$$\mathbf{O}\ [Sp\ @\ f\ @\ (K\ g)] = Cp^h\ f\ @\ g$$
$$\mathbf{O}\ [Sp\ @\ f\ g] = Sp^h\ f\ g$$
$$\mathbf{O}\ [Sp\ f\ @\ g] = Sp^t\ f\ g$$
$$\mathbf{O}\ [Sp\ @\ f\ @\ g] = Sp^{ht}\ f\ g$$

Figure 3.2: Extended Abstraction Algorithm

The extra rule in the abstraction phase is

$$\mathbf{A}\ x\ [@\ f]\ =\ @\ (\mathbf{A}\ x\ [f])$$

The only purpose of this rule is to preserve the "@" so that it can be used later in optimization phase. The optimization rules are discussed in the following subsections, including rules for regular combinators, rules for list-oriented combinators, and the correctness of these optimization rules. The last subsection gives several illustrated examples.

Optimization rules for regular combinators

There are fourteen additional optimization rules for tagged regular combinators. Their design is based on three principles:

- translate the eagerness notation "@" to a proper tagged combinator.
- translate one tagged combinator to another tagged combinator.
- propagate "@".

Rules following the first principle "consume" the "@" notation by converting a "@" in the SASL expression into a control tag in the combinator. The most straightforward example is the rule

$$\mathbf{O}\ [S\ f\ @\ g]\ =\ S^r\ f\ g$$

The only purpose of this rule is to change the "@" into the control tag of S. The expressions in the left side and right side are essentially the same except that they

are written in a different notation. The other rules following this principle are more complicated since they not only convert the "@", but also perform combinator optimization at the same time. For example, the rule to generate B^r is

$$\mathbf{O}\ [S\ (K\ f)\ @\ g] = B^r\ f\ g$$

It simplifies the expression $S\ (K\ f)\ g)$ as well as converts the "@" into proper a tag.

Rules following the second principle are concerned with the transfer of information between combinators and do not involve any "@" notation. In the original SASL system, an optimization rule always converts a complicated expression into a simpler one, therefore eliminating combinators. For example, S and B are eliminated in the optimization rule of $S1$

$$\mathbf{O}\ [S\ (B\ f\ g)\ h] = S1\ f\ g\ h$$

However, combinators in the extended system may carry extra information in its tag, and simply discarding the combinator will lose the control information. Rules following the second principle preserve this information by transferring it from the tag of the old combinator to the tag of newly created combinators. For example, the rule to generate $S1^l$ is

$$\mathbf{O}\ [S\ (B^r\ f\ g)\ h] = S1^l\ f\ g\ h$$

Note that in this rule the r tag in B becomes l tag in $S1$. Although the B has been eliminated, the control information is preserved.

The third principle is the complement of the first. It is applied when a left expression contains a "@" but no proper tagged combinator can match the expression's pattern. In other words, there is no way to consume the "@" in the current

abstraction step. Instead of discarding it, this method provides a way to preserve the "@" and pass it on to future abstraction steps. For example, one rule for the K optimization is

$$\mathbf{O}\ [S\ (K\ f)\ @\ (K\ g)] = K\ f\ @\ g$$

Note that "@" notation appears in the both sides. This principle comes from the fact that the compiling algorithm can only abstract one variable at a time, and there is no way to convert the annotated expression if it does not contain the variable currently being abstracted. For example, consider the abstraction of $f\ x\ y\ =\ g\ y\ @\ (h\ x)$. In the first step, y will be abstracted form the expression. In this step, "@" cannot be properly used since y does not occur in $(h\ x)$ and the whole expression $@\ (h\ x)$ is treated as a constant. If we discard @ in this step, the result after the first abstraction is $C'\ g\ (h\ x)$ and the control information can never be recovered. If we employ the third principle, the result will be $C'\ g\ @\ (h\ x)$ and the control information can be used in the next abstraction step (which abstracts x and will give $B^r\ (C'\ g)\ h$, the desired answer).

Note that some complicated optimization rules may involve more than one principle. For example, the optimization rule for $S1^{lr}$ employs the first and the second principles:

$$\mathbf{O}\ [S\ (B^r\ f\ g)\ @\ h)] = S1^{lr}\ f\ g\ h$$

And one optimization rule for $C1^r$ uses the second an the third principles:

$$\mathbf{O}\ [S\ (B^r\ f\ g)\ @\ (K\ h)] = C1^r\ f\ g\ @\ h$$

Optimization rules for list-oriented combinators

The optimization rules for list-oriented combinators are based on concepts similar to those of regular combinators. The major difference is that the list-oriented combinators may need two separate tags to control the the evaluation of the head and tail respectively. Because list-oriented combinators do not have "long-reach" combinators, like $S1$ and $B1$, their optimization rules employ only the first and the third principles in the previous section. However, because the list-oriented combinators have two control fields, these optimization rules may involve two "@" notations at the same time. For example, both of the following rules involve two "@".

$$\mathbf{O}\ [Sp\ @\ f\ @\ (K\ g)] = Cp^{h}\ f\ @\ g$$

$$\mathbf{O}[Sp\ @\ f\ @\ g] = Sp^{ht}\ f\ g$$

The detailed rules are shown in Figure 3.2.

The correctness of the optimization rules

In order to show that an optimization rule is correct, we need to prove that the expressions on the left-hand side and right-hand side of the rule are equivalent. In the extended system, equivalence means that the two expressions will converge (or be reduced) to a common expression, including the control information (lazy or eager) on each function application. This is more strict than the conventional definition since it involves some control information. For example, under this definition $f\ (g\ x)$ and $f\ \langle g\ x\rangle$ are not considered to be equivalent since the control information on the application $g\ x$ are not identical.

The equivalence of the expressions can be shown in the *extensional* sense; i.e., f and g are considered to be equivalent if and only if $\forall x$, $f\ x$ and $g\ x$ are equivalent. We use the rule $\mathbf{O}\ [S\ (B^r\ f\ g)\ @\ h] = S1^{lr}\ f\ g\ h$ to demonstrate the proof since this is a relatively complicated rule and employs both first and second principles. The other rules can be proved in similar manner. Note that in our extended system, every application must be either lazy (by default, denoted by $(\cdots)$ or blank) or eager (denoted by $\langle\cdots\rangle$ or by $@\ (\cdots)$).

Claim: The two expression $S\ (B^r\ f\ g)\ @\ h$ and $S1^{lr}\ f\ g\ h$ are equivalent (in extensional sense).

Proof:

apply x to $S\ (B^r\ f\ g)\ @\ h$:

$$\begin{aligned}
S\ (B^r\ f\ g)\ @\ h\ x &= (B^r\ f\ g\ x)\ @\ (h\ x) && \text{by the definition of } S\\
&= f\ \langle g\ x\rangle @\ (h\ x) && \text{by the definition of } B^r\\
&= f\ \langle g\ x\rangle\langle h\ \rangle && \text{by our convention of } \langle\rangle\\
&= S1^{lr}\ f\ g\ h\ x && \text{by the definition of } S^{lr}
\end{aligned}$$

□

Illustrated examples

The effect of the extended compiling algorithm can be best demonstrated by some simple examples. Three examples are given in this subsection: one for a one-variable function, one for a two-variable function and the other for list operation.

The first example involves only one variable. Let $test1\ x = f\ (g\ x)\ (h\ x)$, where f, g and h are functions defined elsewhere. If the two arguments of f are eagerly

evaluated, then the function *test*1 can be annotated as $f \mathbin{@} (g\ x) \mathbin{@} (h\ x)$. Some key steps of the abstraction of *test*1 are listed below.

$$\begin{aligned}
\mathbf{A}\ x\ [test1] &= \mathbf{A}\ x\ [f \mathbin{@} (g\ x) \mathbin{@} (h\ x)] \\
&= \mathbf{O}\ [S\ (\mathbf{A}\ x\ [f \mathbin{@} (g\ x)])\ (\mathbf{A}\ x\ [@\ (h\ x)])] \\
&= \mathbf{O}\ [S\ (\mathbf{A}\ x\ [f \mathbin{@} (g\ x)]) \mathbin{@} (\mathbf{A}\ x\ [h\ x])] \\
&\vdots \\
&= \mathbf{O}\ [S\ (\mathbf{O}\ [S\ (K\ f) \mathbin{@} g]) \mathbin{@} \mathbf{O}\ [S\ (K\ h)\ I]) \\
&= \mathbf{O}\ [S\ (B^r\ f\ g) \mathbin{@} h]) \\
&= S^r\ (B^r\ f\ g)\ h
\end{aligned}$$

The effect of the tagged combinatory code can be shown by examining the evaluation of an application, say *test*1 2. The first few steps are given below. Note that, as we desired, both evaluations of $h\ 2$ and $g\ 2$ are initiated earlier.

$$\begin{aligned}
test1\ 2 &= S^r\ (B^r\ f\ g)\ h\ 2 \\
&= (B^r\ f\ g)\ 2\ \langle h\ 2\rangle \\
&= f\ \langle g\ 2\rangle \langle h\ 2\rangle \\
&\vdots
\end{aligned}$$

The second example involves two variables. Let $test2\ x\ y = f\ (g\ y)\ (h\ x)$, where f, g and h are defined elsewhere and the two arguments of f are to be eagerly evaluated. The annotated function is $test2\ x\ y = f \mathbin{@} (g\ y) \mathbin{@} (h\ x)$. Since there are two variables in the function definition, abstraction needs to be performed twice (first on y, then on x). Some key steps of the abstraction are:

$$\begin{aligned}
\mathbf{A}\ y\ [test2\ x] &= \mathbf{A}\ y\ [f \mathbin{@} (g\ y) \mathbin{@} (h\ x)] \\
&= \mathbf{O}\ [S\ (\mathbf{A}\ y\ [f \mathbin{@} (g\ y)])\ (\mathbf{A}\ y\ [@\ (h\ x)])] \\
&= \mathbf{O}\ [S\ (\mathbf{A}\ x\ [f \mathbin{@} (g\ x)]) \mathbin{@} (\mathbf{A}\ y\ [h\ x])]
\end{aligned}$$

$$
\begin{aligned}
&\vdots \\
&= \mathbf{O}\ [S\ (\mathbf{O}\ [S\ (K\ f)\ @\ g])\ @\ (K\ (h\ x))] \\
&= \mathbf{O}\ [S\ (B^r\ f\ g)\ @\ h] \\
&= C1^r\ f\ g\ @\ (K\ (h\ x))
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{A}\ x\ [test2] &= \mathbf{A}\ x\ [C1^r\ f\ g\ @\ (K\ (h\ x))] \\
&= \mathbf{O}\ [S\ (\mathbf{A}\ x\ [C1^r\ f\ g])\ (\mathbf{A}\ x\ [@\ (K\ (h\ x))] \\
&= \mathbf{O}\ [S\ (\mathbf{A}\ x\ [C1^r\ f\ g])\ @\ (\mathbf{A}\ x\ [K\ (h\ x)])] \\
&\vdots \\
&= \mathbf{O}\ [S\ (K\ (C1^r\ f\ g))\ @\ h] \\
&= B^r\ (C1^r\ f\ g)\ h
\end{aligned}
$$

Note that only one "@" has been consumed in the abstraction on x, and the other "@" is propagated to the abstraction on y. The effect of the tagged combinatory code is shown below, where the evaluation of say *test*2 3 4 is given. Again, as we desired, both the evaluations of h 3 and g 4 are initiated earlier.

$$
\begin{aligned}
test2\ 3\ 4 &= B^r\ (C1^r\ f\ g)\ h\ 3\ 4 \\
&= C1^r\ f\ g\langle h\ 3\rangle\ 4 \\
&= f\ \langle g\ 4\rangle\langle h\ 3\rangle \\
&\vdots
\end{aligned}
$$

The abstraction processes for list operations is similar to that of scalar operation except the function application is replaced by list construction. However, the annotation on list operations may drastically change its lazy property. The third example shows how this effect can be achieved. Consider $g\ n = n : (g\ (inc\ n))$, which generates an infinite list starting from n (*inc* is a function that increases its argument by 1). To eagerly construct the entire list, the function g can be annotated as

$g\ n = n\ :\ @\ (g\ (inc\ n))$. Of course, this example is only for demonstration purposes since the evaluation will never terminate. The key steps of the abstraction are:

$$\begin{aligned}
\mathbf{A}\ x\ [g] &= \mathbf{A}\ n\ [n : @(g\ (inc\ n))] \\
&= \mathbf{O}\ [Sp\ (\mathbf{A}\ n\ [n])\ (\mathbf{A}\ n\ [@(g\ (inc\ n))])] \\
&= \mathbf{O}\ [Sp\ (I)\ @\ (\mathbf{A}\ n\ [g\ (inc\ n)])] \\
&= \cdots \\
&= \mathbf{O}\ [Sp\ I\ @\ (B\ g\ inc)] \\
&= Sp^t\ I\ (B\ g\ inc)
\end{aligned}$$

The execution of $g\ 1$ is:

$$\begin{aligned}
g\ 1 &= Sp^t\ I\ (B\ g\ inc)\ 1 \\
&= (I\ 1) : \langle (B\ g\ inc)\ 1 \rangle \\
&= (I\ 1) : \langle g\ 2 \rangle \\
&= (I\ 1) : (I\ 2) : \langle (B\ g\ inc)\ 2 \rangle \\
&= (I\ 1) : (I\ 2) : \langle g\ 3 \rangle \\
&= (I\ 1) : (I\ 2) : (I\ 3) : \langle g\ 4 \rangle \\
&= (I\ 1) : (I\ 2) : (I\ 3) : (I\ 4) : \cdots
\end{aligned}$$

As we expected, the evaluation of $g\ 1$ continues building the list and will never terminate since the list is infinite. Note that the list's elements have not been evaluated because the head has not been annotated as eager.

Summary

In this chapter, we have developed a scheme to add control information that can override the original lazy semantics of SASL system. The scheme first annotates SASL programs to specify which expression should be eagerly evaluated, and then transfers the annotation to combinator's control tags. The propagation is performed by the extended compiling algorithm whose optimization phase has been expanded to handle the new control information. Because sequencing is largely irrelevant in functional languages, this scheme fits well into the original framework of SASL system and can

be treated as a multiprocessor extension rather than a new language.

In the next two chapters, we will examine the effectiveness of this scheme by simulating program execution on an idealized multiprocessor system.

CHAPTER 4. SIMULATION OF THE MULTIPROCESSOR REDUCTION SYSTEM

The effectiveness of the proposed annotation scheme is evaluated by simulating program execution on a virtual shared-memory multiprocessor system. The purpose of using a virtual system is to shield the influences from the physical system. Since the virtual system represents an idealized multiprocessor system, the execution speed will not be degraded by communication delays or resource contention, and hence the performance can rely only on the inherent parallelism and the efficiency of the annotation scheme. The design of the simulator is described in this chapter, and the benchmark programs and their simulation results will be discussed in the next chapter.

There are two major modules in the SASL implementation, namely the compiling module and the execution module. The major task of the compiling module is to compile the annotated SASL programs into tagged combinatory code. This module includes two sub-modules which perform parsing and abstraction respectively. The task of the execution module is to simulate the execution of the combinatory code on an idealized system. It contains a reducer module that implements the template for a single instance of a processor, and a process management module that coordinates the operation between processors. Detailed descriptions of the two modules is discussed

in the following sections.

The Design of the Compiling Module

The compiling module is based on the original SASL interpreter. Both the parsing sub-module and the abstraction sub-module are extended to incorporate the annotation scheme. Although the annotation scheme drastically changes the operational semantics of the standard SASL language, there are only minor modifications in the syntactic level and therefore the major part of the two sub-modules is kept intact.

In the parsing sub-module, two syntax rules are modified to reflect the inclusion of the annotation. To include the annotated function application, the rule for regular expressions is changed from

```
<simple> ::= <name> | <constant> | <zfexpr> | (<expr>)
```

[57, p. 23] into

```
<simple> ::= <name> | <constant> | <zfexpr> | (<expr>) | @ (<expr>)
```

To include eager list construction, the rule for the structure is changed from

```
<struct> ::= <formal> | <formal> : <struct>
```

[57, p. 23] into

```
<struct> ::= <formal> | <formal> : <struct> | @ (<formal>) : <struct> |
             <formal> : @ (<struct>) | @ (<formal>) : @ (<struct>)
```

In the abstraction sub-module, the extensions are manifested as extra optimization rules. In the original implementation, the optimization sub-module is implemented as case analysis and each rule is represented by a case with a specific pattern.

Thus, the major modification in this sub-module is to expand the patterns in the case analysis part, specially:

- add extra cases to match expressions with the "@" symbol.
- add an extra level case analysis in each case clause to handle the pattern matching of the control tags.

The output of the extended compiling module is similar to that of the original SASL compiler except that each node now has an extra field containing the combinator's control tag.

The Design of the Execution Module

The execution module simulates the operation of an idealized shared-memory multiprocessor system. The purpose of the simulation is to study the performance of the proposed parallelism-control mechanism. The two major parts of the execution module are the reducer and process manager. Although the two parts are discussed separately, their operations are tightly-coupled. The following subsections describe the operation of the multiprocessor system, process management and the reducer, and give the pseudo code of the execution module.

The idealized combinator-based multiprocessor system

The idealized multiprocessor system can be thought as a shared-memory system containing a large number of processors and a global memory with infinite bus and memory bandwidth. Under this assumption, there is no communication delay or

resource contention and hence all physical constraints are lifted. The processor is similar to a traditional processor except that it needs to perform combinator reduction and to do some basic process manipulation. Each processor contains three basic units:

- ALU unit: perform basic arithmetic and logic operations (the δ-rules).
- reduction unit: perform the combinator reduction and modify the corresponding graph (β-reduction).
- process management unit: perform the process management including initiating the new processes and updating the status of the residing process.

The global memory is shared by all processors and can be accessed simultaneously. Each memory cell corresponds to a node in the combinator graph and includes the following fields:

- *type*: indicate the type of this node; it can be *atom*, *list*, or *application*.
- *left*: depending on the type of the node, it contains:
  - the pointer to the applicator, if the type is application.
  - the pointer to the head, if the type is list.
  - a constant symbol, including combinators, built-in functions, numbers, etc., if the type is atom.
- *right*: depending on the type of the node, it contains:
  - the pointer to the applicant, if the type is application.
  - the pointer to the tail, if the type is list.

- NIL, if the type is atom.

- *status*: node's current execution status, which can be either *reducible, reducing* or *reduced.*

- *tag*: contains the control tag, if the node is a combinator; otherwise undefined.

The left and right fields represent the two branches of the binary tree and are self-explanatory. The status field is included to assist the synchronization of the multi-processor operation. It indicates whether the graph pointed by this node is reducible, is under evaluation, or has been evaluated. The process checks, and modifies, if required, the status field during the initiation and termination. To avoid racing or deadlock, operations on this field are *atomic.* The tag and type fields are also self-explanatory.

Process management

Because there are more than one processor working on the same task, a process manager is needed to coordinate and synchronize the operation. A process can be defined as the procedure that reduces a combinator graph to the normal form. All processes maintain, in addition to general statistics-collecting and housekeeping fields, three pieces of information:

- *root*: the current root of the graph to be reduced.

- *state*: the current execution status of the process, which can be either *running, waiting* or *completed.*

- *argument list*: a list containing the roots of the strict arguments which are currently being evaluated.

During every clock cycle, one of the following actions may take place in a process:

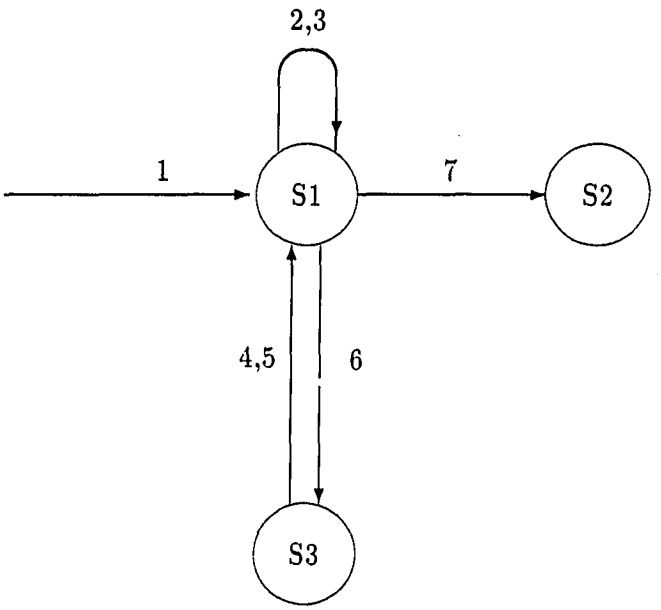
- terminating: if the graph is in normal form.
- reducing: if the graph is reducible and all the required strict arguments have been evaluated.
- waiting: if the strict arguments are being evaluated by other process.
- suspending: if the evaluation of the strict argument(s) has not been initiated; the process sets up a new process for the first strict arguments.

The "reduction" and "suspending" actions may create new processes. If the reduction is performed on a tagged combinator, the control tag will create new processes for the designated expressions. These processes are normally independent of their parent process and can be executed concurrently. Since these processes can be distributed to separate processors, they contribute to the actual speed-up of the multiprocessor system. In the "suspending" action case, a new process is created to evaluate the strict argument. Note that creating a new process for this purpose is not necessary since the operation of old process is always suspended and hence control can be transferred to its child process, as in a traditional procedure call. Although it is possible to let some strict operators such as + and * create a new child process for their strict arguments, we choose not to do so for the uniformity of the design and treat them as lazy. If strictness is desired, these operators can be easily overridden by an extra function; for example, eager + can be written as

```
eager_plus x y = @ x + @ y
```

The operation of the multiprocessor system is coordinated by using a combination of the status field in the node and the process state, as shown in Figure 4.1. When a process is created, its state is set to *running* since there is always a processor available in the idealized system. Depending on the status field of the root, one of the following actions may take place:

- if the status is *reduced*, the process changes its state to *terminate* and signals its completion by broadcasting its root so that processes waiting for that subgraph can then use the new value.
- if the status is *reducing*, which means another process initiated the evaluation earlier, the process' state changes to *waiting* and the root is put root into process' argument list.
- if the status is *reducible*, the process changes it to *reducing* to prevent duplicated computation and starts the evaluation. During the evaluation, there are three possibilities:
  - if the graph is in normal form, the process changes the status to *reduced* and the state to *terminated*, and signals its termination by broadcasting its root.
  - if all strict arguments are available, the process performs a reduction and stays in the *running* state.
  - if the evaluation of the strict arguments has not been initiated, the process sets up new process to evaluate the corresponding subgraph, puts the



Transitions:

1: initialization
2: combinator reduction
3: all strict arguments *reduced*
4: strict argument *reducible*
5: strict argument *reducing*
6: empty argument list
7: graph in normal form

States:

S1: Running
S2: Terminated
S3: Waiting

Figure 4.1: State Diagram for the Process Control

subgraph's root into the argument list and changes its state to *waiting*.

- if any of the strict arguments is under evaluation since the corresponding node's status is *reducing*, the process' state changes to *waiting* and puts the roots into its argument list.

After the process enters the *waiting* state, it continues monitoring the broadcasting roots and eliminates the matched root from its argument list. Once the elements of the argument list has decreased to zero (i.e., all the strict arguments have been reduced), the process can change its state to *running* and continue the evaluation.

The reducer

The reducer simulates the operation of the ALU and reduction unit of the processor. Its basic algorithm is similar to the reducing module of the standard SASL; however, two major extensions are made for the multiprocessor environment:

- before initiating the evaluation of an argument, it will test the node's status to check whether the evaluation has already started. If so, it will wait for completion rather than evaluating the argument again.
- after completing the reduction of a combinator, it will check the control tag and may initiate eager evaluation of some new expressions composed during the reduction.

Outline of the execution module

The execution module is responsible for simulating the execution of tagged combinatory code. Its pseudo code is shown below:

```
create a process for the expression;
while (there are RUNNING process){
   increase clock tick by 1;
   for (each process with active status){
      unwinding the spine;
      if (the tip's status is REDUCED}{
         change the process' state to TERMINATED;
         continue:}
      if (the tip's status is REDUCING){
         change the process' state to WAITING;
         put the current root into argument list;
         continue;}
      if (the graph is in normal form){
         change the process root's status to REDUCED;
         change the process' state to TERMINATED;
      }else{
         change the process root's status to REDUCING;
         switch(tip){
           case COMBINATOR:
             modify the graph accordingly;
             if (control tag exists)
               create a process for the corresponding expression;
             break;
           case OPERATOR:
             if (all the strict arguments has been evaluated){
               perform the operation;
               break;}
             for (every strict arguments under evaluation)
               put its root into argument list
             if (there is an unevaluated strict argument){
               create a new process for the argument;
               change the new process' status to REDUCING;}
             change the process' state to WAITING
         } /* end of switch */
      } /* end of else */
   } /* end for loop */
   for (each process with state TERMINATED){
     broadcasting its root;
     for (each process)
```

```
        if (the root match one in argument list)
           delete the matched node from list
      delete the terminated process;}
    for (each process){
      if (the list is empty)
        change the state to RUNNING;}
} /* end while loop */
dump the relevant statistics;
```

The top-level **while** loop simulates the operation performed in one clock cycle. The two top-level **for** loops inside the **while** perform the synchronization of the processes. While the first loop does the initial checking, the second loop performs the final updating. The actual reduction is simulated by the inner **switch** construct, which is further divided into the reduction of combinators and regular operators.

The actual program is coded in the C language and takes about 2500 lines. Its design emphasizes in the clarity and correctness rather than efficiency and compactness. Rough estimation shows that it takes about twenty hours of CPU time on an AT&T 3B15 to simulate an eight processor system with a clock rate one million reductions per second for one second.

The next chapter will show the actual simulation results of some selected programs.

CHAPTER 5. THE EFFECTIVENESS OF THE ANNOTATION SCHEME

One of the major design criteria for multiprocessor systems is maximizing the potential speedup. In an idealized system, this criterion becomes more significant since the utilization of physical resources are not a major concern. Thus, in the virtual system level, the effectiveness of the proposed annotation scheme can be measured by the parallelism extracted from the application programs. In this chapter, we analyze a set of programs using the simulator described in the previous chapter, and study their run-time behavior and potential speedup.

The extractable parallelism depends on a wide variety of factors. From the system point of view, making effective use of parallelism relies on hardware aspects, such as bus and memory organization, as well as programming language features, such as the parallelism control constructs. From the application point of view, the parallelism depends on the nature of the problem domain, the algorithm used, the programming style, and the size and pattern of input data. Although it is desirable to control every aspect of the program, achieving this goal is almost impossible. First, there are too many factors that may influence the program execution in a multiprocessor environment. Sometimes a minor change in one sensitive factor may greatly affect the outcome. Moreover, some factors such as the programming style and the pattern of

the input data are only vaguely defined and have no exact metrics to measure them. Second, many factors are not independent but are highly interrelated and cannot be easily isolated and analyzed. In other words, we can only run the program and observe the collective behavior of many interwoven factors.

Our approach to this problem is to employ two different sets of benchmark programs. Programs in the first set represent some typical scenarios encountered in the multiprogramming environment. Because of their simplicity, we can study their run-time behavior in more detail and examine the characteristics of the proposed scheme. Programs in the second set are more "realistic" and represent some real-world applications. Because these programs are generally larger and have many factors influencing their execution, explaining their detailed behavior is extremely difficult, if not impossible. Thus, in this set, we only observe the overall effects rather than trying to explain their run-time behavior in detail. The simulation results of the two sets of programs will be examined and discussed in the following sections.

Effect of Annotation on Some Representative Benchmarks

In this section, we investigate the characteristics of the proposed annotation scheme by carefully examining some selective benchmark programs. Because these programs are simple and their run-time behavior is well understood, the effect of the annotation mechanism can be studied in more detail. The benchmark programs include `fib`, which calculates Fibonacci numbers, `map`, which performs a homogeneous operation on all the elements of a list, and `isort`, which sorts a list by the insertion-sort method. These programs were chosen because they contain some characteristics

that are frequently encountered in the non-numeric programming environment, and each of them represents a particular kind of strategy commonly used to extract parallelism.

Benchmark 1: Fibonacci numbers

Recall that the definition of the Fibonacci sequence is:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ f(n-1) + f(n-2) & \text{otherwise} \end{cases}$$

It can be easily coded in SASL:

```
fib n = n<3 -> 1; fib (n-1) + fib (n-2)
```

This program is a widely used benchmark because it has several interesting properties. First, this program is highly recursive and generates numerous function calls, so it can be used to test the efficiency of function call implementation. Second, in a parallel environment, this program can naturally apply the divide-and-conquer strategy since the computation of the "else" branch can be divided into two parallel subtasks, namely `fib (n-1)` and `fib (n-2)`. Although `fib (n-1)` and `fib (n-2)` are not identical, they have similar granularity. In other words, when `fib n` is invoked, the load can be distributed into two evenly divided subtasks. Thus, this program serves as an ideal divide-and-conquer example.

Simulated results In our system, the divide-and-conquer scheme can be easily achieved by annotating the two branches of + as eager:

```
fib n = n<3 -> 1; @(fib (n-1)) + @(fib (n-2))
```

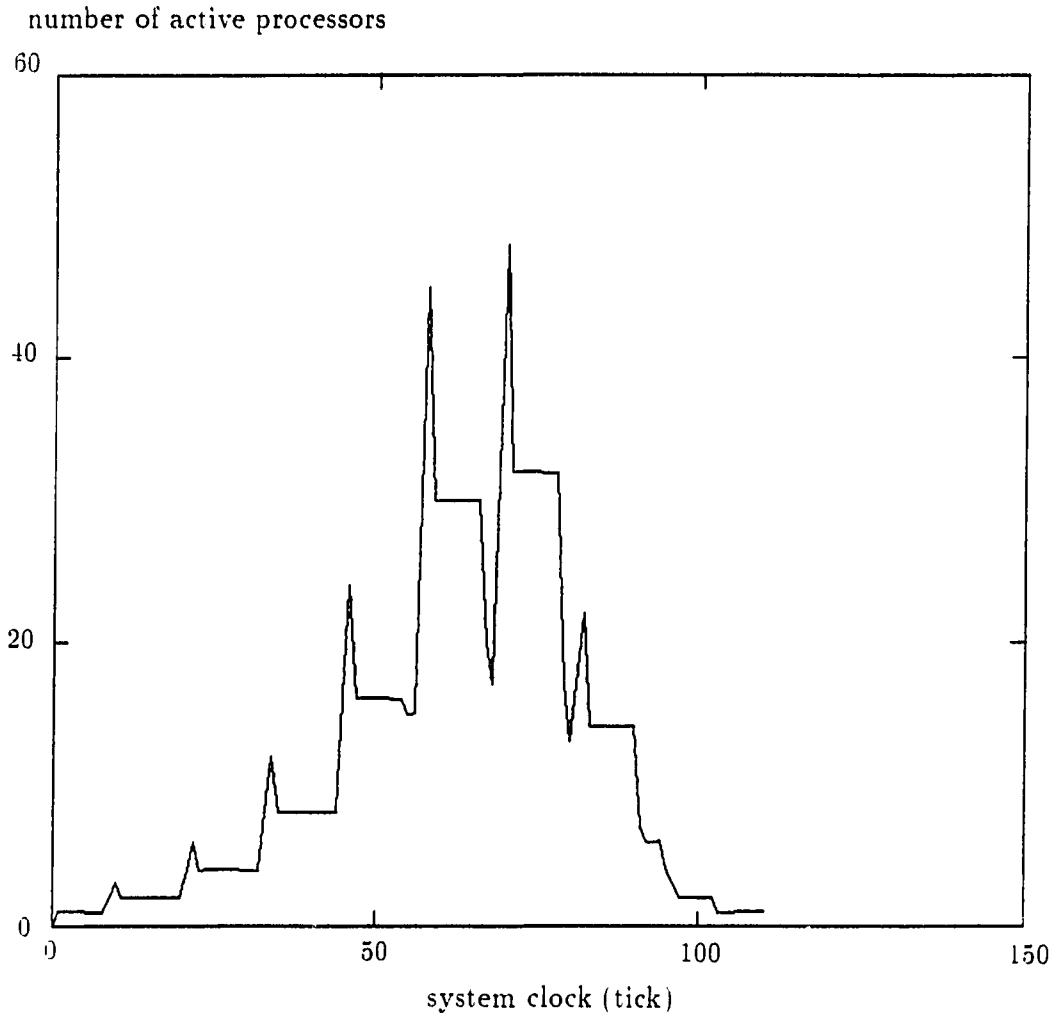
The corresponding compiled combinator code of `fib` is

$$fib = S\ (C1\ cond\ (<\ 3)\ 1)\ (S^r\ (B1^l + fib\ (C - 1))\ (B\ fib\ (C - 2)))$$

The tags of S and $B1$ are due to the annotation in the right branch and left branch of + respectively.

The extractable parallelism is represented by the profile of active processors, which is a plot with the vertical axis as the number of active processors and the horizontal axis as the virtual system clock. The execution of `fib 10` is shown in Figure 5.1. The trend in the plot is similar to what we expect from a divide-and-conquer program. It can be roughly divided into the spreading phase and the resolving phase. In the spreading phase, initiation of the new processes is the dominant activity. Since the parent process recursively invokes child processes and distributes its work into smaller pieces, the number of active processors continuously increases. When the execution reaches the leaves in the divide-and-conquer computation tree, the child processes start to return the results and the execution gradually enters the resolution phase. In the resolution phase, the termination of the processes becomes the dominant activity. Since a large number of child processes return their results and terminate, the number of active processors declines rapidly.

To explain the profile in more detail, we need to look at the trace of the execution. Some key steps in the execution of the initial recursion cycle of `fib 10` are highlighted in the following paragraph. The underlined symbols in the trace represent the operators under reduction and the number to the right denotes the system clock. Note that the node containing a constant symbol needs one tick to be marked as *reduced*, and some nodes may need to be visited twice because of the unevaluated




```
fib 10 where fib n = n<3->1; @(fib (n-1)) + @(fib (n-2))
      total time elapsed: 110 ticks
      total work done: 1304 reductions
      speedup: 11.84
```

Figure 5.1: Profile with **fib n = n<3->1; @(fib (n-1)) + @(fib (n-2))**

strict arguments; i.e., the first visit initiates the evaluation of some strict arguments, and the second visit, which occurs when the value of the argument has been returned, performs the actual reduction. The trace is:

$$
\begin{array}{lll}
 & fib\ 10 & \\
= & \underline{S}\ (C1\ cond\ (<\ 3)\ 1)\ (S^r\ (B1^l + fib\ (C-1))\ (B\ fib\ (C-2)))\ 10 & 1 \\
= & \underline{C1}\ cond\ (<\ 3)\ 1\ 10\ (S^r\ (B1^l + fib\ (C-1))\ (B\ fib\ (C-2))\ 10) & 2 \\
= & \underline{cond}\ (<\ 3\ 10)\ 1\ (S^r\ (B1^l + fib\ (C-1))\ (B\ fib\ (C-2))\ 10) & 3 \\
= & cond\ (\underline{\leq\ 3\ 10})\ 1\ (S^r\ (B1^l + fib\ (C-1))\ (B\ fib\ (C-2))\ 10) & 4 \\
= & \underline{cond}\ false\ 1\ (S^r\ (B1^l + fib\ (C-1))\ (B\ fib\ (C-2))\ 10) & 5 \\
\vdots & & \\
= & \underline{S^r}\ (B1^l + fib\ (C-1))\ (B\ fib\ (C-2))\ 10 & 8 \\
= & (\underline{B1^l} + fib\ (C-1)\ 10)\ \langle \underline{B}\ fib\ (C-2))\ 10 \rangle & 9 \\
= & \pm\ \langle \underline{fib}\ ((C-1)\ 10) \rangle\ \langle \underline{fib}\ ((C-2))\ 10) \rangle & 10 \\
\vdots & &
\end{array}
$$

The execution of every recursion level can be roughly divided into the evaluation of the conditional test and the evaluation of the "else" branch. The conditional test part is evaluated sequentially, as in the original SASL system. The "else" part distributes its work into two subtasks by initiating two new processes to evaluates the arguments of the addition operator. These new processes recursively call `fib` and repeat the similar procedure. This procedure continues until the termination condition of `fib` is met. The initial task distribution can be observed from the spreading phase of the plot. If we ignore the spikes, the number of active processors increases like a "geometric ladder" for five steps; i.e., the number is doubled for a fixed amount of time, repeating five times. The increment comes from the fact that every process initiates two new child processes, which have similar form but with different arguments. Because of the recursion, the total number of processors grows

geometrically. The length of each step (except for the first) is about 12 ticks, which represents the time of setting up child process, performing the conditional test and evaluating actual parameters. This geometrical trend continues until some processes start to return their values and terminate. At this moment, the increment trend levels off because the increment introduced by the newly initiated processes is canceled by the terminating processes. This can be observed in the sixth step, in which the number of processors increases only slightly. Since a large number of processes can be terminated at same time, there are notches in the resolution phase.

Now let us examine the source of the spike in each step. Although the intention of the previous annotation is to initiate the concurrent execution of `fib (n-1)` and `fib (n-2)`, it may initiate three evaluations, namely `fib (n-1)`, `fib (n-2)` and +, and therefore the number of active processors may be tripled. This can be better explained by changing the program to prefix form:

```
fib n = n<3 -> 1; + @(fib (n-1)) @(fib (n-2))
```

When the evaluation of the "else" branch starts, + is eagerly evaluated by default and `(fib (n-1))` and `(fib (n-2))` by eager annotation. Thus, there is a possibility that three processes execute concurrently. The actual invocation of two branches is completed in two steps. First, reduction on S' (at tick 8) initiates the concurrent evaluation of $(B1^l + fib\ (C' - 1))\ 10$ and $(B\ fib\ (C' - 2))\ 10$, which correspond to `(+ (fib (n-1)))` and `(fib (n-2))` respectively. Then, the reduction on $B1^l$ (at tick 9) initiates + and $fib\ ((C' - 1)\ 10)$, which correspond to + and `(fib (n-1))`. Since + is a built-in function, its evaluation is completed in one tick and then suspended to wait for the values of its two arguments. Thus, for a very short time, there are three

active processes, which are responsible for the initial spike in every step.

Discussion There are several interesting properties that can be observed from this example. First is the asymmetry of the spreading phase and the resolution phase. Compared to the resolution phase, the spreading phase has a relatively slow start. In a conventional multiprocessor system, this scenario would normally indicate that there is a substantial overhead in initiating a new process. However, this is not the case in our combinator-based system since the initiation of a new process only needs to pass one pointer that points to the root of the expression and does not involve any complicated context-switching or environment manipulation. Instead, our results can be attributed to the initial parameter distribution and sequential conditional test. The trace shows that it takes 7 ticks to perform the condition test, but only 3 ticks to complete the initiation of `fib (n-1)` and `fib (n-2)`. Although it is partially due to the internal implementation of *cond*, the slow start of the spreading phase primarily comes from the large overhead in parameter distribution of combinator-based systems. This is particularly true for the systems based on Turner's combinator set since they can only distribute the actual parameter one level at a time. The effect of the combinator's control tag is to force the concurrent execution of the two branches, but it will not to accelerate the parameter distribution in the "vertical dimension". Therefore, the distribution still takes place level by level, and the spreading phase is relatively sluggish. Once the parameters have been distributed to the proper places, there are relatively few "real" computations, such as + and function initiation, and therefore the process terminates very quickly. Thus, the resolution phase is much faster than the spreading phase.

Another important observation is the effect of annotation. To study this, we annotated the program in several different ways. All the annotations initiate the concurrent execution of `fib (n-1)` and `fib (n-2)`, though the variations introduce some subtle differences. For convenience, the prefixed expression is used for the rest of this section. The original program can be rewritten as

```
fib n = n<3 -> 1; + @(fib (n-1)) @(fib (n-2))
```

The first variation is to drop the first "`@`"; i.e.,

```
fib n = n<3 -> 1; + (fib (n-1)) @(fib (n-2))
```

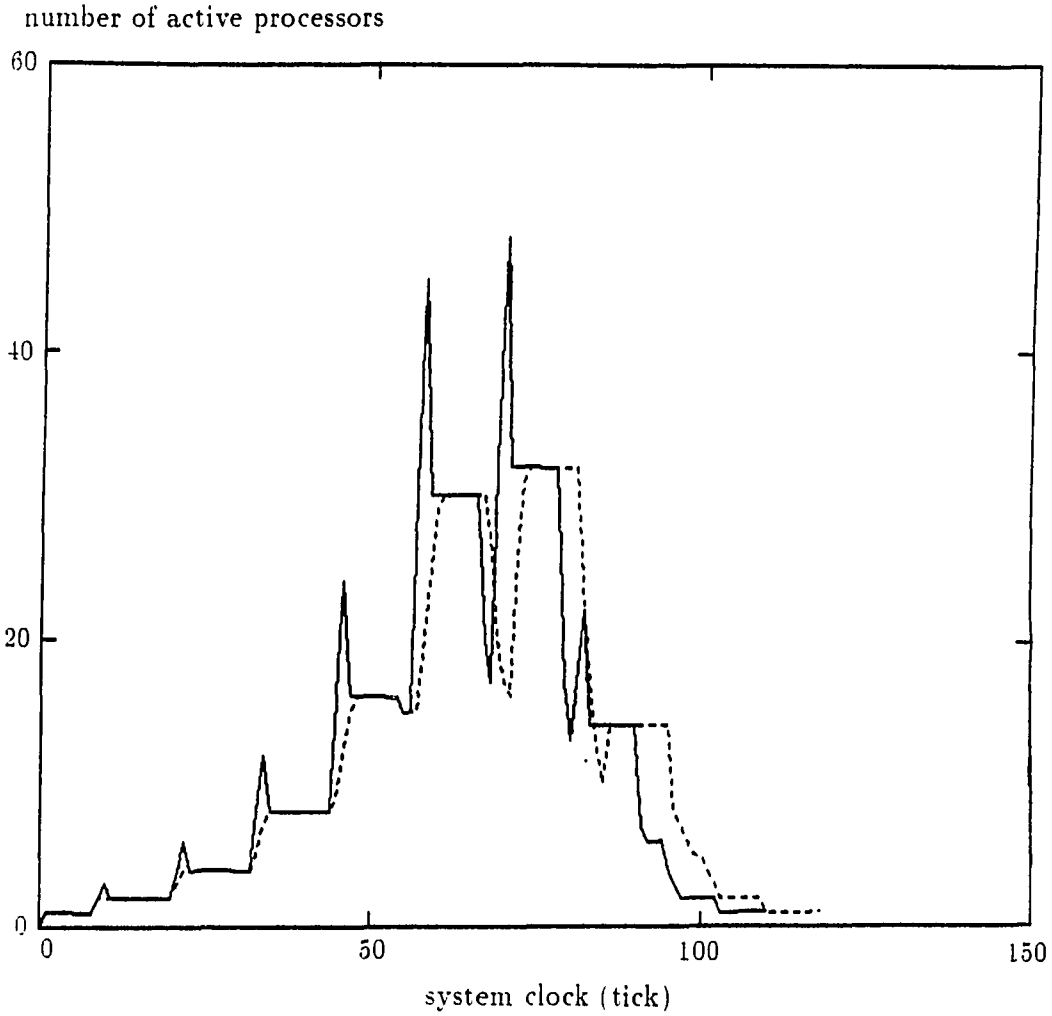
The compiled code becomes

$$fib \;=\; S\ (C1\ cond\ (<\ 3)\ 1)\ (S^r\ (B1 + fib\ (C-1))\ (B\ fib\ (C-2)))$$

Note that the corresponding tag on $B1$ is not used. This change makes the evaluation of `+ (fib (n-1))` sequential and postpones the initiation of the evaluation of `fib (n-1)` for one tick. The trace shows this delay:

$$\begin{array}{lll}
 & fib\ 10 & \\
= & \underline{S}\ (C1\ cond\ (<\ 3)\ 1)\ (S^r\ (B1 + fib\ (C-1))\ (B\ fib\ (C-2)))\ 10 & 1 \\
\vdots & & \\
= & \underline{S^r}\ (B1 + fib\ (C-1))\ (B\ fib\ (C-2))\ 10 & 8 \\
= & (\underline{B1} + fib\ ((C-1)\ 10))\ \langle \underline{B}\ fib\ ((C-2)\ 10)\rangle & 9 \\
= & \pm\ (fib\ ((C-1)\ 10))\ \langle \underline{fib}\ ((C-2)\ 10)\rangle & 10 \\
= & +\ (\underline{fib}\ ((C-1)\ 10))\ \langle\cdots\rangle & 11 \\
\vdots & &
\end{array}$$

Note that initiation of the evaluation of $fib\ ((C-1)\ 10)$ is postponed from tick 10 to tick 11. The run-time profile is shown in Figure 5.2, accompanied by the



solid line: `fib 10 where fib n = n<3->1; + @(fib (n-1)) @(fib (n-2))`
total time elapsed: 110 ticks total work done: 1304 reductions
speedup: 11.84

dash line: `fib 10 where fib n = n<3 -> 1; + (fib (n-1)) @(fib (n-2))`
total time elapsed: 118 tick total work done: 1304 reductions
speedup: 11.03

Figure 5.2: Profile with **fib n = n<3->1; + (fib (n-1)) @(fib (n-2))**

profile of the original version. Because the task is now divided into two subtasks (i.e., `+ (fib (n-1))` and `fib (n-2)`), rather than three (i.e., `+`, `fib (n-1)` and `fib (n-2)`), the spikes in the previous profile disappear. The sequential execution of `+ (fib (n-1))` introduces a small period of delay, which makes each step shift to the right a little. Although the amount of shift is fairly small, it is accumulated in every step and its effect becomes more apparent as the execution progresses. There is a noticeable shift in the last step and the execution time is prolonged from 110 ticks to 118 ticks.

The second variation shows the effect of nested annotation. Since the eager annotation affects only one level, the expressions nested inside the annotated expression still maintain their lazy semantics. This is the case for `(n-1)` and `(n-2)` since they are nested inside `fib (n-1)` and `fib (n-2)`. We can continue tracing the execution of one branch of `fib 10`, say `fib (10-1)`:

$$
\begin{aligned}
& (\underline{B1^{l}} + fib\ (C-1)\ 10) && 9\\
= \ & \pm\ \langle \underline{fib}\ ((C-1)\ 10)\rangle && 10\\
= \ & +\ \underline{S}\ (C1\ cond\ (<\ 3)\ 1)(S^{r}\ (B1^{l} + fib\ (C-1))(B\ fib\ (C-2)))((C-1)\ 10)\\
= \ & +\ \underline{C1}\ cond\ (<\ 3)\ 1\ ((C-1)\ 10)(S^{r}\ (B1^{l} + fib\ (C-1))\\
& (B\ fib\ (C-2))((C-1)\ 10) && 11\\
= \ & +\ \underline{cond}\ (<\ 3\ (C-1)\ 10))\ 1\ (S^{r}\ (B1^{l} + fib\ (C-1))\\
& (B\ fib\ (C-2))((C-1)\ 10) && 12\\
= \ & +\ cond\ (\leq\ 3\ (C-1)\ 10))\ 1\ (S^{r}\ (B1^{l} + fib\ (C-1))(B\ fib\ (C-2))\ 10) && 13\\
= \ & +\ cond\ (<\ 3\ \underline{C}-1)\ 10))\ 1\ (S^{r}\ (B1^{l} + fib\ (C-1))\ (B\ fib\ (C-2))\ 10) && 14\\
= \ & +\ cond\ (<\ 3\ \underline{-}\ 10\ 1))\ 1\ (S^{r}\ (B1^{l} + fib\ (C-1))\ (B\ fib\ (C-2))\ 10) && 15\\
= \ & +\ cond\ (\leq\ 3\ 9))\ 1\ (S^{r}\ (B1^{l} + fib\ (C-1))\ (B\ fib\ (C-2))\ 10) && 16\\
& \vdots
\end{aligned}
$$

The trace shows that the evaluation of $(C-1)\ 10$ is postponed until its value is required by the boolean expression in the conditional test, $<\ 3\ ((C-1)\ 10)$. Because

we know that the argument of `fib` is strict and its value is eventually required, it can be annotated as eager to override the original lazy semantics; i.e.,

```
fib n = n<3 -> 1; + @(fib @(n-1)) @(fib @(n-2))
```

The compiled code becomes

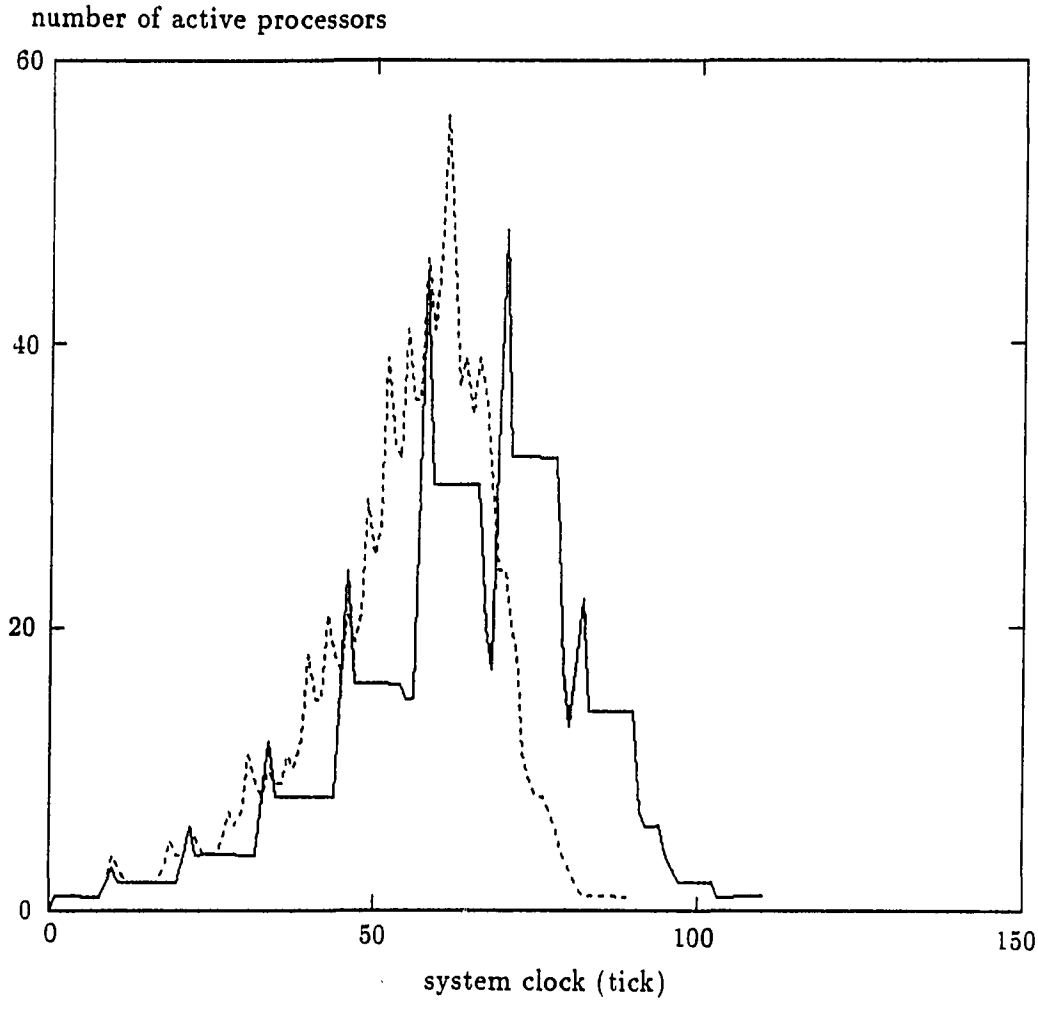
$$fib \;=\; S\;(C1\;cond\;(<\;3)\;1)\;(S^r\;(B1^{lr} + fib\;(C-1))\;(B^r\;fib\;(C-2)))$$

The annotation introduces more parallelism since, during the function application, both the body and the argument of `fib` are evaluated simultaneously . For comparison, we can trace the execution of `fib (10-1)` again:

$$\begin{array}{lll}
 & (\underline{B1^{lr}} + fib\;(C-1)\;10) & 9 \\
= & \pm\;\langle \underline{fib}\;\langle(\underline{C}-1)\;10\rangle\rangle & 10 \\
= & +\;\underline{S}\;(C1\;cond\;(<\;3)\;1)(S^r\;(B1^{lr} + fib\;(C-1))(B^r\;fib\;(C-2)))\;\langle(\underline{C}-1)\;10\rangle & \\
= & +\;\underline{C1}\;cond\;(<\;3)\;1\;\langle(\underline{-}\;10\;1)\rangle\;(S^r\;(B1^{lr} + fib\;(C-1)) & \\
 & (B^r\;fib\;(C-2))\langle(\underline{-}\;10\;1)\rangle & 11 \\
= & +\;\underline{cond}\;(<\;3\;9)\;1\;(S^r\;(B1^{l} + fib\;(C-1))\;(B\;fib\;(C-2))\;9) & 12 \\
\vdots & &
\end{array}$$

Because of the eager evaluation of the argument, the conditional test takes four fewer ticks, which considerably shortens the spreading phase. The run-time profile is shown in Figure 5.3, accompanied by the profile of the original version. There are no clear steps in this plot since the initiation of new processes becomes irregular. Comparing to the original annotation, the new scheme can extract a significant amount of parallelism in addition to the simple divide-and-conquer scheme. The total execution time decreases from 110 ticks to 89 ticks and the speedup increases from 12.0 to 14.5.

The third observation concerns the order of initiation. At the first glance, the annotation `@(fib (n-1)) + @(fib (n-2))` seems to initiate the two branches at



solid line: `fib 10 where fib n = n<3 -> 1; + @ (fib (n-1)) @(fib (n-2))`
total time elapsed: 110 ticks total work done: 1304 reductions
speedup: 11.84

dash line: `fib 10 where fib n = n<3->1; + @(fib @(n-1)) @(fib @(n-2))`
total time elapsed: 89 ticks total work done: 1304 reductions
speedup: 14.02

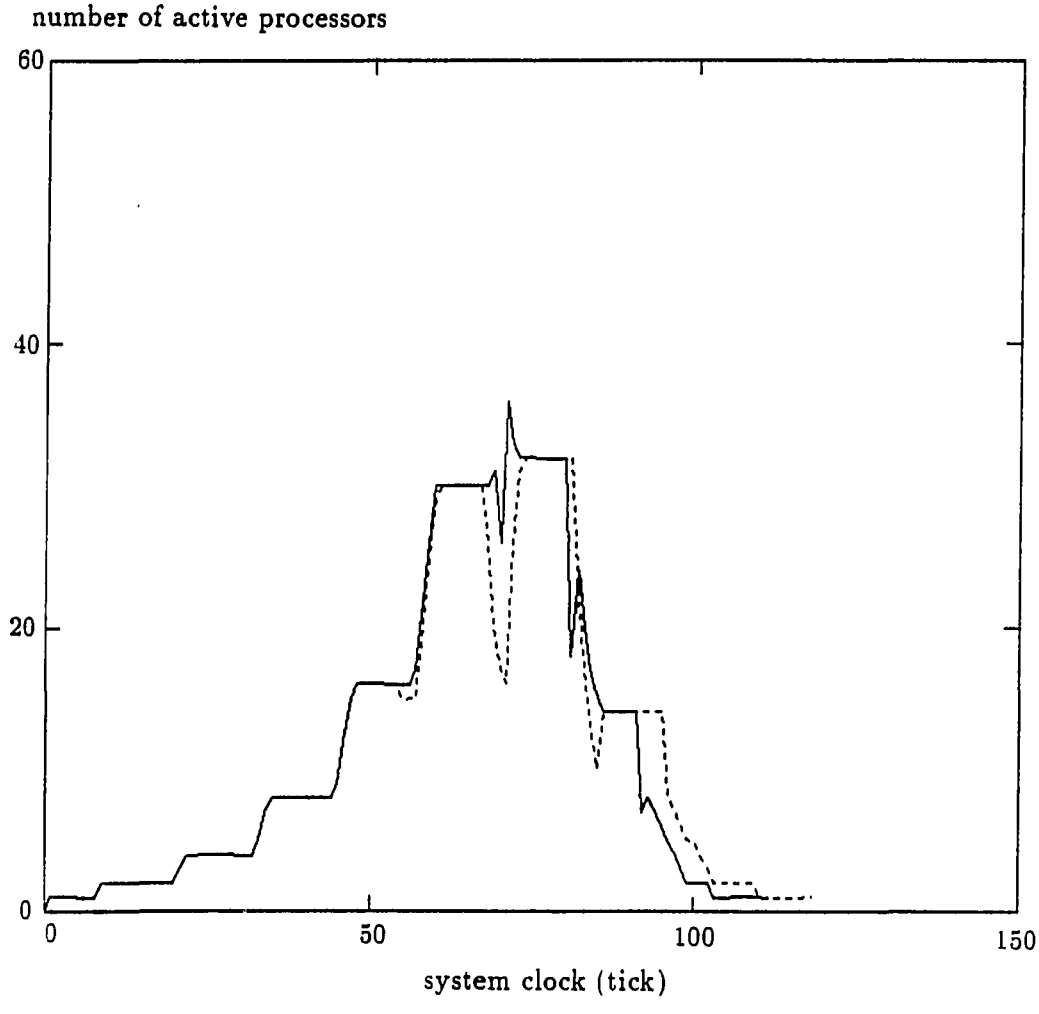
Figure 5.3: Profile with **fib n = n<3->1; + @(fib @(n-1)) @(fib @(n-2))**

the same time. However, this is not the case. From the trace, we know that the two branches are initiated by different combinators tags, and that the right branch is initiated one tick earlier. This is due to the inherent design of SASL, namely the fully-curried expression. That is, while expressions like `@(fib (n-1)) + @(fib (n-2))` are written in infix form for the convenience of programmers, internally they are represented in curried form, `(+ @(fib (n-1))) @(fib (n-2))`. Since the left branch is nested one extra level, its initiation is delayed. The asymmetry in initiation raises a subtle point about determining the best way to distribute the load. Because the + operator is commutative, its two branches can be swapped. Since the `fib (n-1)` branch contains more work than `fib (n-2)` branch, it may be better to initiate the evaluation of `fib (n-1)` first; i.e., switch it to the right branch. To highlight the difference, consider the two following versions:

```
fib n = n<3 -> 1; + (fib (n-1)) @(fib (n-2))
fib n = n<3 -> 1; + (fib (n-2)) @(fib (n-1))
```

Although the two versions are semantically equivalent, the second version seems to be better since `fib (n-2)`, which contains less work, is grouped with + and `fib (n-1)`, which contains more work, is invoked earlier. The run-time profiles of the two versions are shown in Figure 5.4, where we can see that our assumption is correct. The resolution phase in the second version is shorter, and the total execution time decreases from 118 ticks to 111 ticks.

The last observation concerns about the initiation of the arguments of the strict operators. As mentioned in the previous chapter, one alternative scheme to invoke the eager evaluation of the arguments of strict operators is to change the internal



solid line: `fib 10 where fib n = n<3 -> 1; + (fib (n-2)) @(fib (n-1))`
total time elapsed: 118 ticks total work done: 1304 reductions
speedup: 11.03

dash line: `fib 10 where fib n = n<3 -> 1; + (fib (n-1)) @(fib (n-2))`
total time elapsed: 111 ticks total work done: 1304 reductions
speedup: 11.73

Figure 5.4: Comparisons between the two commutative versions

evaluation mechanism so that the evaluation of the arguments will be automatically initiated. We compared this mechanism with the previous annotated example. Since there is no annotation, all the tags in the combinatory code are dropped:

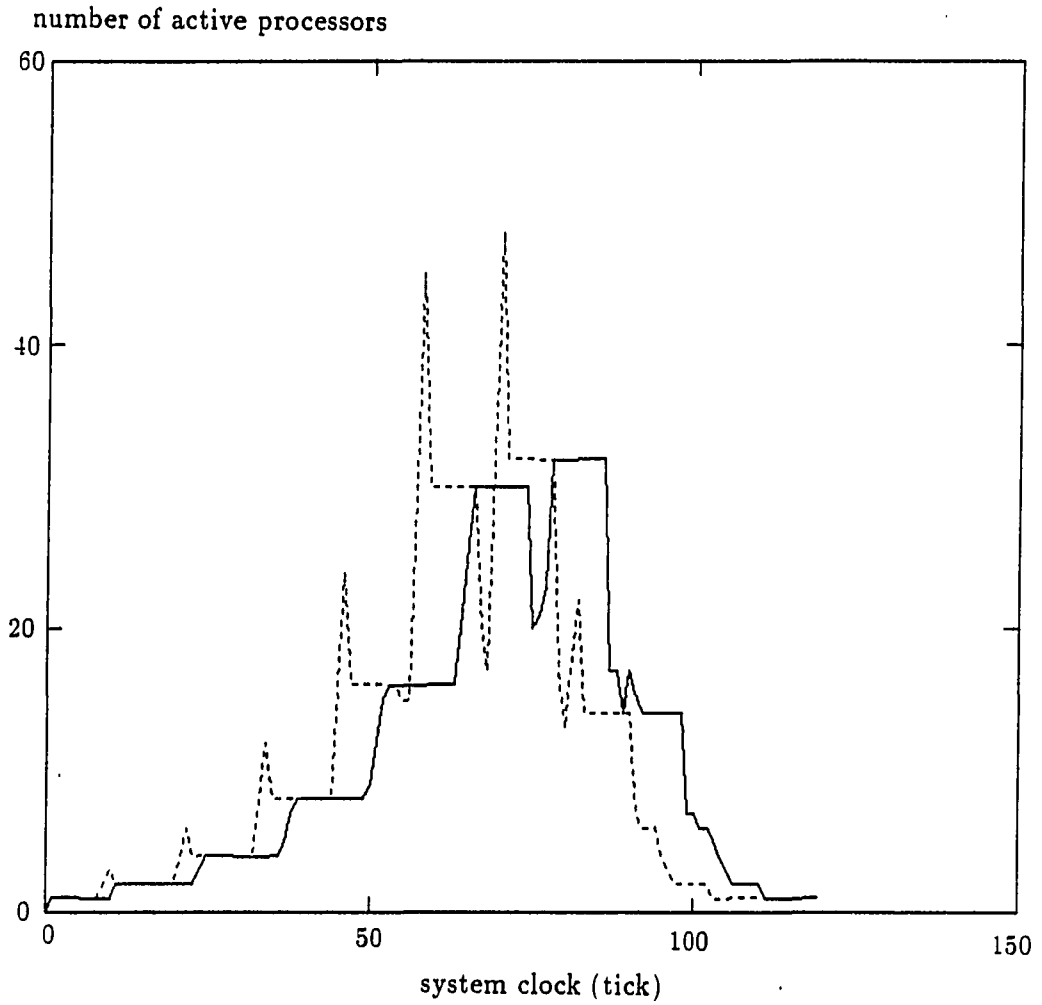
$$fib \;=\; S\;(C1\;cond\;(<\;3)\;1)\;(S\;(B1+fib\;(C-1))\;(B\;fib\;(C-2)))$$

The execution trace becomes:

$$\begin{array}{lll}
 & fib\ 10 & \\
= & \underline{S}\;(C1\;cond\;(<\;3)\;1)\;(S\;(B1+fib\;(C-1))\;(B\;fib\;(C-2)))\;10 & 1 \\
\vdots & & \\
= & \underline{S}\;(B1+fib\;(C-1))\;(B\;fib\;(C-2))\;10 & 8 \\
= & (\underline{B1}+fib\;(C-1)\;10)\;(B\;fib\;(C-2))\;10) & 9 \\
= & \pm\;(fib\;((C-1)\;10))\;(B\;fib\;(C-2))\;10) & 10 \\
= & +\;\langle\underline{fib}\;((C-1)\;10))\rangle\;\langle\underline{B}\;fib\;(C-2))\;10\rangle & 11 \\
\vdots & &
\end{array}$$

While compared to the original trace, both the initiation of the evaluation of $fib\;((C-1)\;10)$ and $fib\;((C-2)\;10)$ are delayed until the evaluation of $+$ is done. While the former is postponed from tick 10 to tick 11, the later is postponed from tick 8 to tick 11. The run-time profile is shown in Figure 5.5, accompanied by the profile of the annotated version. The initiation process becomes even slower and the major computation curve shifts further to the right. This example shows that the annotated version is better because the evaluation of the arguments can be started before the evaluation of operator is completed.

Summary `fib` demonstrates many interesting properties of the proposed annotation scheme. It shows the annotation scheme is fairly general and flexible. By proper annotation, we can achieve the desired effects such as the concurrent evaluation of two branches and the eager evaluation of the strict arguments. On the



solid line: `fib 10 where fib n = n<3->1; + (fib (n-2)) (fib (n-1))`
total time elapsed: 119 ticks total work done: 1304 reductions
speedup: 10.94

dash line: `fib 10 where fib n = n<3->1; + @ (fib (n-2)) @ (fib (n-1))`
total time elapsed: 110 ticks total work done: 1304 reductions
speedup: 11.84

Figure 5.5: Profile with **fib n = n<3->1; + (fib (n-2)) (fib (n-1))**

other hand, `fib` also shows some subtlety of the annotation, as demonstrated in the various variations. To "fine-tune" a program, the programmer needs an in-depth understanding of the characteristics of the program and the internal curry mechanism of the SASL system. However, because the execution order of code segments written in a functional language is not significant, any annotation will usually lead to substantial performance improvement, even when the annotation is not optimal. We therefore claim that this scheme is much simpler and more effective than other conventional techniques.

In summary, this benchmark has shown that a considerable amount of parallelism can be extracted by our proposed annotation scheme and the overall computation can be speeded up by a factor of fourteen, even for a small input number. The major limitation of this scheme is the slow parameter distribution, which contributes to the relatively slow spreading phase of the `fib`. This comes from the fact that whereas the proposed system invokes the concurrent evaluation of function body and function argument, it still distributes parameters one level at a time and does not speed up the execution in the "vertical" dimension. To further speed up the execution, we may need to employ more complicated combinator sets, such as Abdali's combinator and super combinator, to decrease the overhead in the parameter distribution and to shorten the spreading phase.

Benchmark 2: `map`

`Map` is a function that applies a second function to every element in a list. The SASL code for `map` is:

```
map f   ()  = ()
```

```
map f (a:x) = (f a) : (map f x)
```

Map has several characteristics that are frequently encountered in symbolic programming. First, it represents a widely-used style to code list-oriented operations. In this kind of code, a new list is constructed by first defining how to generate its head and tail (`(f a)` and `(map f x)` respectively here), and then combining them by the constructor ":". Because the list is the only available data structure in SASL, the role played by the list operation becomes more significant. Second, the first parameter to map is a higher-order function (i.e., functional argument `f`), which is common in functional programming. Therefore, map can be used to test the efficiency of the implementation of second-ordered structures.

Simulated results To obtain potential parallelism, map can be annotated to evaluate the head and tail simultaneously:

```
map f    ()  = ()
map f (a:x) = @(f a) : @(map f x)
```

Although this is legitimate SASL code, it employs the internal pattern-matching mechanism and introduces a new dimension of complexity because of the new combinators such as TRY, $MATCH$, U, etc. To avoid this, we rewrite it in a more "traditional" form:

```
map f l = l eq () -> (); @(f (hd l)) : @(map f (tl l))
```

In this version, pattern-matching is replaced by a conditional test and by `tl` and `hd` operators. In our implementation, both the pattern-matching and the conditional

test are performed sequentially; therefore, the two versions are essentially the same. The compiled code **map** corresponding to the above annotation is:

$$map \;=\; B\ (S\ (C1\ cond\ (C\ eq\ nil)\ nil))\ (S1\ Sp^{ht}\ (C\ B\ hd)\ (C1\ B\ map\ tl))$$

The effect can be best explained by the following demonstration. In this example, several "dummy" functions, **d1**, **d2**, **d3** and **d4**, are used to generate sequential computations of various lengths, and four test functions (**maptestx**) force every element in the list to be evaluated.

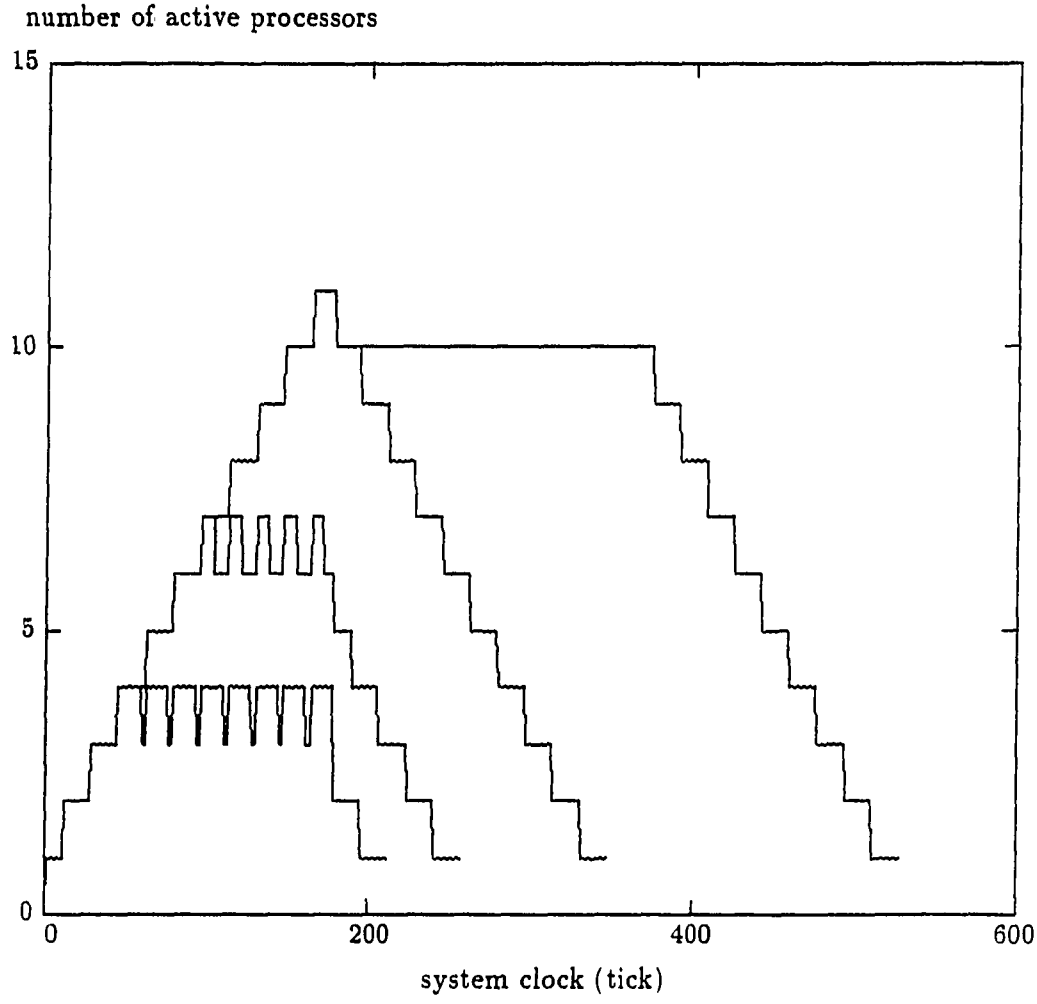
```
d1 x = x & x & x & x & x & x & x & x & x & x & x & x & x & x & x
d2 x = d1 (d1 x)
d3 x = d2 (d2 x)
d4 x = d3 (d3 x)
dlist = (true, true, true, true, true, true, true, true, true, true)
maptest1 = map d1 dlist
maptest2 = map d2 dlist
maptest3 = map d3 dlist
maptest4 = map d4 dlist
```

The profile of this test program is shown in Figure 5.6. We first study **maptest4** and then explain the remaining functions. The profile of **maptest4** has a trapezoidal shape with staircase-like sides. Unlike the previous divide-and-conquer example, interpretation for this plot is not obvious. The best explanation comes from the detailed trace of the execution:

$$
\begin{array}{lll}
 & map\ d4\ dlist & \\
= & \underline{B}\ (S\ (C1\ cond\ (C\ eq\ nil)\ nil))\ (S1\ Sp^{ht}\ (C\ B\ hd)\ (C1\ B\ map\ tl))\ d4\ dlist & 1 \\
= & \underline{S}\ (C1\ cond\ (C\ eq\ nil)\ nil))\ ((S1\ Sp^{ht}\ (C\ B\ hd)\ (C1\ B\ map\ tl))\ d4)\ dlist & 2 \\
= & \underline{C1}\ cond\ (C\ eq\ nil)\ nil)\ dlist\ (((S1\ Sp^{ht}\ (C\ B\ hd)\ (C1\ B\ map\ tl))\ d4)\ dlist) & 3 \\
= & \underline{cond}\ (C\ eq\ nil\ dlist)\ nil)\ (((S1\ Sp^{ht}\ (C\ B\ hd)\ (C1\ B\ map\ tl))\ d4)\ dlist) & 4 \\
\vdots & & \\
= & \underline{S1}\ Sp^{ht}\ (C\ B\ hd)\ (C1\ B\ map\ tl)\ d4\ dlist & 10 \\
= & \underline{Sp^{ht}}\ ((C\ B\ hd)\ d4)\ ((C1\ B\ map\ tl)\ d4)\ dlist & 11 \\
= & \langle \underline{C}\ B\ hd\ d4\ dlist \rangle\ :\ \langle \underline{C1}\ B\ map\ tl\ d4\ dlist \rangle & 12 \\
= & \langle \underline{B}\ d4\ hd\ dlist \rangle\ :\ \langle \underline{B}\ (map\ d4)\ tl\ dlist \rangle & 13 \\
= & \langle \underline{d4}\ (hd\ dlist) \rangle\ :\ \langle (\underline{map}\ d4)\ (tl\ dlist) \rangle & 14 \\
\vdots & & \\
= & \langle \cdots \rangle\ :\ \langle \underline{d4}\ (hd\ (tl\ dlist)) \rangle\ :\ \langle (\underline{map}\ d4)\ (tl\ (tl\ dlist)) \rangle & 31 \\
\vdots & &
\end{array}
$$

The first 10 ticks perform the conditional test where the execution is sequential as in the standard SASL system. The key step is tick 11, where the h and t tags of Sp initiate the eager evaluation of the head and the tail respectively. While the old process is inherited by the evaluation of the head, a new process is created for the evaluation of the tail. The two processes are executed concurrently from tick 12 and are responsible for the step in Figure 5.7 at tick 12. Since the tail part uses map recursively, its execution is similar to the previous process except that the input list is one element shorter. This procedure repeats itself for several times until the list is empty.

The exact run-time behavior can be best demonstrated in Figure 5.7. Every slot in the figure represents an independent process, including the initial conditional test and the evaluation of the head. As the execution progresses, each element becomes the head of the list and its evaluation is initiated accordingly. This initiation explains



Four plots from left to right: `maptest1`, `maptest2`, `maptest3`, `maptest4`

	work done	time elapsed	speedup
`maptest1`	658	212	3.10
`maptest2`	1108	257	4.31
`maptest3`	2008	347	5.79
`maptest4`	3808	527	7.23

Figure 5.6: Profile of **map** with four different input functions

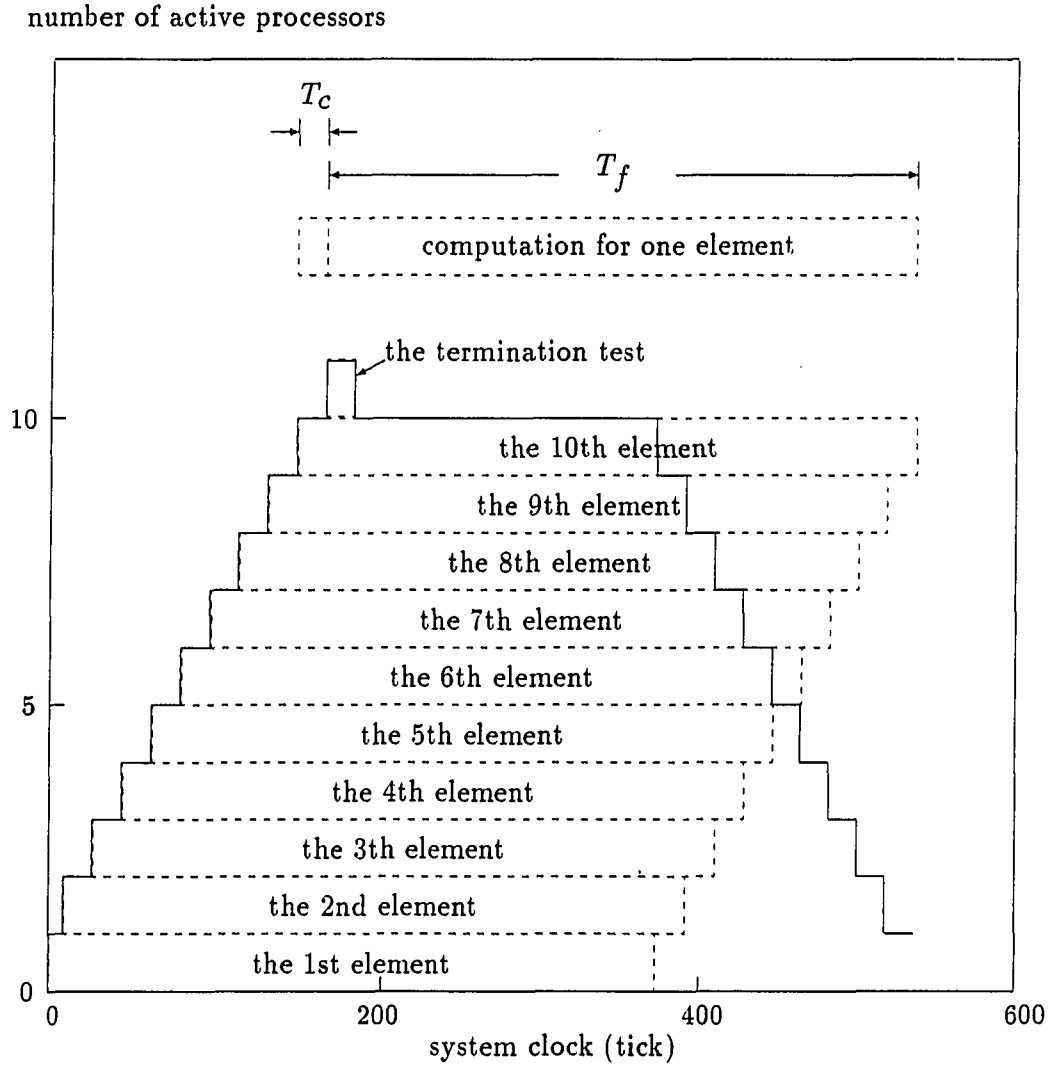


Figure 5.7: The process chart of **maptest4**

the regular upward staircase in the plot. The first ten steps represent the initiation of the ten elements and the last step represents the terminal condition test. Since the evaluation of the element takes the same time, the length of the slots is the same except for the last slot, which is considerably shorter since it performs the termination test and does not include the evaluation of an element.

The three remaining test functions, `maptest3`, `maptest2` and `maptest1`, can be interpreted in a similar way. However, because the function performed on the element is much simpler, the computation time decreases and hence the slot becomes shorter. In `maptest3`, the plateau disappears. In `maptest2` and `maptest1`, a process may terminate before the last process is initiated and hence the computation can never reach to the point where all ten processes are active.

Discussion There are several interesting observations that can be made about this benchmark program. First, the speedup of `map` depends primarily on the complexity of the function `f`, which is performed on every element of the list. If `f` is complicated, each process will take more time to complete and increase the potential for concurrent execution, which contributes to the total speedup. On the other hand, if `f` is simple, each process will terminate quickly, even before the other processes has been initiated, and introduce very little parallelism. This scenario is clearly shown in the four previous test functions.

A more detailed explanation can be formalized as follows. Note that each process can be divided into the execution of the conditional test, T_c, and the evaluation of applying the function to a element, T_f, as shown in the Figure 5.7. The T_f part is overlapped with another process since at that time the evaluation of tail is also

initiated. If we ignore the last termination-testing process, the time spent on the evaluation of a list with N elements in a sequential system, T_s, and in a multiprocessor system, T_m, are:

$$T_s = N(T_c + T_f)$$

$$T_m = NT_c + T_f$$

And the speedup, S, is:

$$S = \frac{T_s}{T_m} = \frac{NT_c + NT_f}{NT_c + T_f}$$

As the value of T_f increases, S will increase because of the large N in the numerator.

An interesting point about speedup is the case where T_f is large. If the $T_f \gg T_c$, the S can be approximated by

$$S = \frac{NT_c + NT_f}{NT_c + T_f} \simeq \frac{NT_f}{T_f} = N$$

This speedup is just what we can expect from an idealized vector processor. In other words, if the function is complicated enough, evaluating map in a combinator-based multiprocessor can achieve the results similar to those of a vector processor.

The second observation is on the source of parallelism. At the first glance, eager evaluation and lazy structures seem to be two incompatible concepts; however, they can be nicely incorporated into the combinator-based multiprocessor system. The operation of the lazy list can be divided into two basic parts, namely the evaluation of the head and the construction of a closure. The original purpose of the closure is to save the necessary information so that the evaluation of the tail can be resumed later. Since the closure contains all the necessary information, it is possible to move the closure to another processor and to start the evaluation of the tail. If the overhead

for constructing the closure is small, the evaluation of the head and tail can be performed concurrently. Furthermore, if the evaluation of the tail recursively invokes the evaluation of the new head, as in map, several processes' execution can be overlapped and a considerable amount of parallelism can be extracted. Although this concept can be applied to all kinds of multiprocessor systems, the complexity and the high overhead involved in the closure construction make it difficult for environment-based systems. On the contrary, the closure construction can be done by assigning a pointer and introduces virtually no overhead in a combinator-based system; thus, this scheme is particularly effective for our proposed system.

Benchmark 3: insertion sort

`Isort` sorts a list of numbers in descending order by inserting each element sequentially:

```
isort    () = ()
isort (a:x) = insert a (isort x)
insert a    () = a,
insert a (b:x) = a < b -> (a:b:x); b:(insert a x)
```

Unlike the previous programs, `isort` does not fit well into any "conventional" multiprogramming paradigm. If we treat operations on a list as undivisible and atomic, `isort`'s operation is completely sequential since in the `isort (a:x)`, the insertion of `a` cannot start until the `(isort x)` is done. There is no opportunity to divide the process into smaller child processes or to initiate homogeneous computation on every element of the list. Thus, it is very hard to extract parallelism from this program by using conventional techniques. However, because of the lazy structure, there is still a

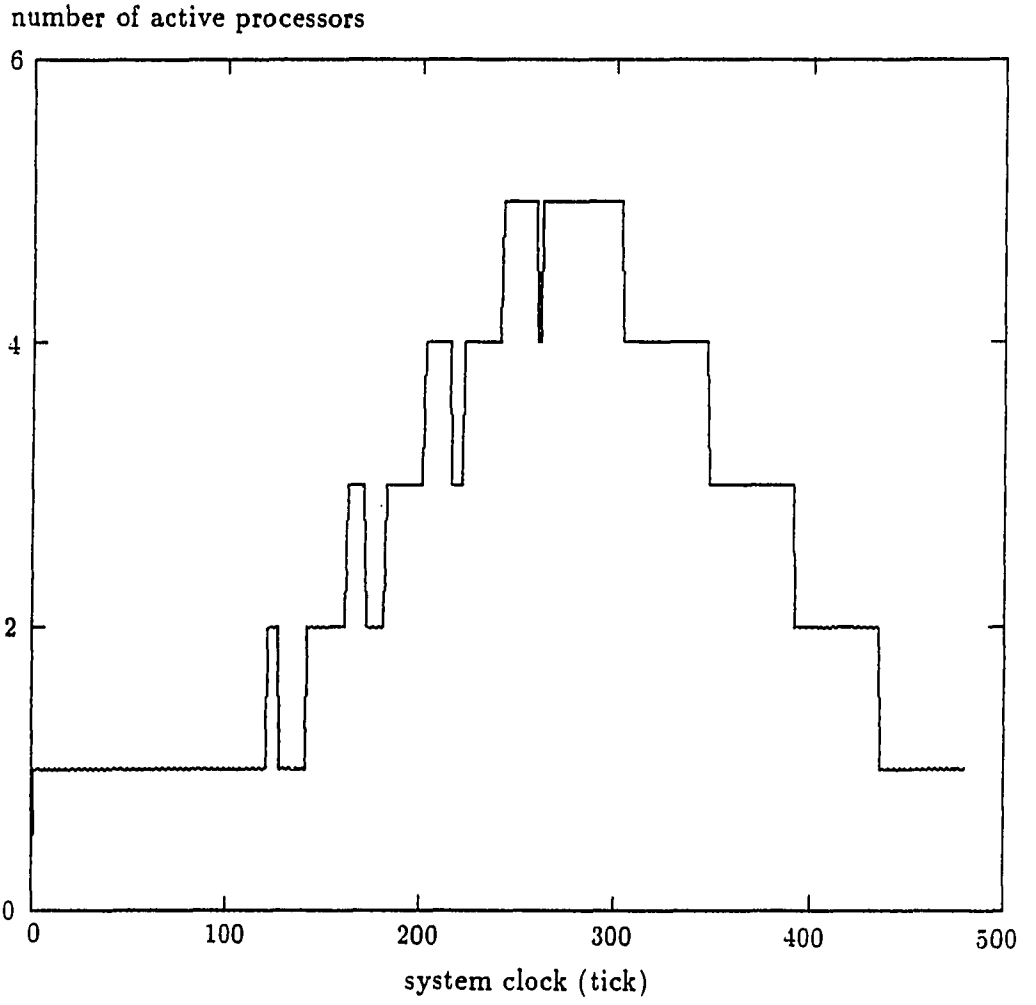
chance to exploit some hidden parallelism in the combinator-based system. To speed up the execution, we can annotate the **insert** part to force eager construction of the list:

```
isort    () = ()
isort (a:x) = insert a (isort x)
insert a    () = a,
insert a (b:x) = a < b -> (a:b:x); b:@(insert a x)
isorttest = isort (10:9:8:7:6:5:4:3:2:1)
```

The profile of the **isorttest** is shown in Figure 5.8, which indicates that the execution is accelerated by a factor of 2.5. The source of the speedup comes from the concurrent evaluation of the head and the tail, as in the execution of **map**. However, this parallelism is more implicit and is automatically formed during the list construction. Since the operation of **isort** in the combinatory level is similar to that of **map**, we focus on the operation of a more abstract level in this subsection. Its effect can be best explained by tracing the execution. For simplicity, a shorter sequence, (5:4:3:2:1), is considered:

$$
\begin{array}{lll}
 & \mathit{isort}5:4:3:2:1 & \\
= & \underline{\mathit{insert}}\ 5\ (\mathit{isort}\ 4:3:2:1) & 1 \\
= & \mathit{insert}\ 5\ (\underline{\mathit{insert}}\ 4\ (\mathit{isort}3:2:1)) & 2 \\
 & \vdots & \\
= & \mathit{insert}\ 5\ (\mathit{insert}\ 4\ (\mathit{insert}\ 3\ (\mathit{insert}\ 2\ (\mathit{insert}\ 1\ (\underline{\mathit{isort}}\ \mathit{NIL}))))) & 3 \\
= & \mathit{insert}\ 5\ (\mathit{insert}\ 4\ (\mathit{insert}\ 3\ (\mathit{insert}\ 2\ (\underline{\mathit{insert}}\ 1\ \mathit{NIL})))) & 4 \\
= & \mathit{insert}\ 5\ (\mathit{insert}\ 4\ (\mathit{insert}\ 3\ (\underline{\mathit{insert}}\ 2\ (1,)))) & 5 \\
= & \mathit{insert}\ 5\ (\mathit{insert}\ 4\ (\underline{\mathit{insert}}\ 3\ (1:@(\mathit{insert}\ 2\ \mathit{NIL})))) & 6 \\
= & \mathit{insert}\ 5\ (\mathit{insert}\ 4\ (\underline{\mathit{insert}}\ 3\ (1:(\underline{\mathit{insert}}\ 2\ \mathit{NIL})))) & \\
 & \vdots &
\end{array}
$$

Initially, *isort* recursively invokes *insert* until *isort* reaches the termination condition, as in the step 3. After *isort* returns NIL, the innermost *insert* becomes active



total time elapsed: 480 ticks
total work done: 1004 reductions
speed-up: 2.50

Figure 5.8: Profile of **isort (10:9:8:7:6:5:4:3:2:1)**

again and returns a list of one element, (1,), as in step 4. In step 5, another *insert* becomes active and returns a partial result, (1 : @(*insert* 2 *NIL*)). In step 6, the evaluation of *insert* 3 (1 : (*insert* 2 *NIL*)) is initiated because the first element of the list, namely 1, is available. At the same time, because of the eager annotation, the evaluation of *insert* 2 *NIL* continues and therefore the execution is overlapped. As the execution progresses, the evaluation of other *insert* can be initiated as soon as the first element of the input list is available and may introduce new overlapped operation. As shown in the profile, the number of active processes can reach five for the ten-element list. In this benchmark, the execution implicitly builds a "pipeline" for the list construction so that the head can be piped to the required function even when the value of the tail is still under evaluation.

The two previous benchmarks show that eagerly evaluating lazy structures is a very powerful mechanism. It can automatically overlap list operations and introduce considerable parallelism. Because the overhead associated with closure construction is negligible in combinator-based systems, this mechanism can be applied in small granularity and extract a considerable amount of fine-grain parallelism.

Effects of Annotation on Some Realistic Benchmarks

In this section, we study the effectiveness of the proposed annotation scheme by observing the run-time profile of some "realistic" benchmark programs. Unlike the programs in the previous section, the execution of these programs is less regular and their run-time behavior represents the interaction of various features. Thus, instead of analyzing one specific feature and tracing the program execution, we only observe

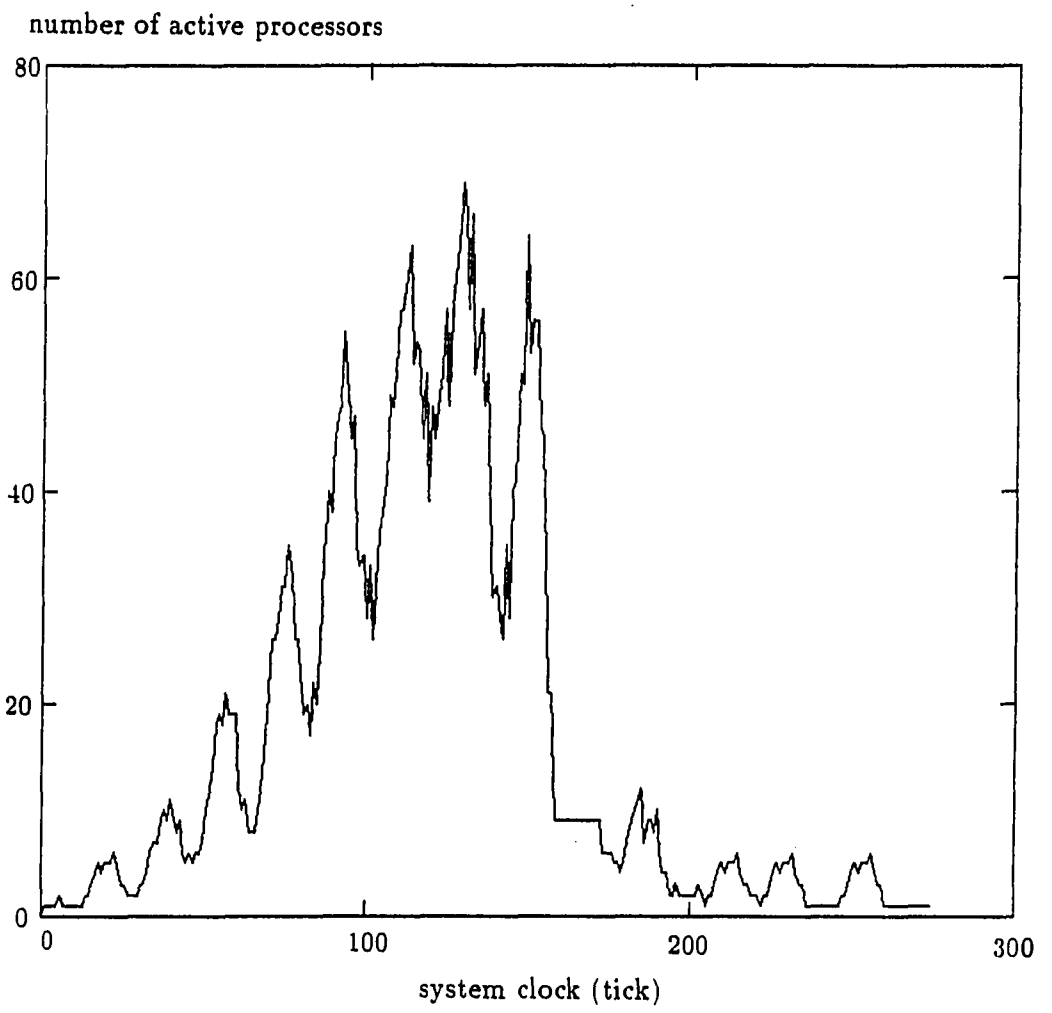
the overall effect of the programs in this section.

Since SASL is a newly developed language, there are relatively few existing benchmarks. The major part of our benchmark programs are adapted from Lisp [15] [60]. The major difference between Lisp and SASL is that real-world Lisp is only a quasi-functional language, which includes "procedural" constructs, such as assignment (e.g., *setq*), destructive structure operation (e.g., *conc*) and sequential execution (e.g., **let*). Thus, some programs needed to be extensively modified to accommodate the framework of SASL. Despite the modification, we tried to keep the "structure" of the original programs intact. In other words, we did not modify or fine-tune the programs to fit our system; instead, we just faithfully transformed the original programs and added proper annotation.

Because of the time and space limitations of the simulator, the benchmark programs are fairly small. Many of them are only the "toy version" of the actual programs. However, despite their size, these programs are representative and employ a variety of symbolic programming techniques. The detailed programming listing is attached in the Appendix.

Benchmark 4: `tak`

`Tak` is a widely used benchmark program in the Lisp environment. It generates irregular function calls and is used to test the function call and primitive arithmetic operations. The profile of `tak 9 6 3` is shown in Figure 5.9.



total time elapsed: 274 ticks
total work done: 4648 reductions
speedup: 16.96

Figure 5.9: Profile of tak_test

Benchmark 5: tautology checking

`Taut` checks whether the input logical function is a tautology. It is used to test higher-order structures and primitive logical operations. The eagerness annotation changes the lazy OR and AND operators into strict operators and may introduce some redundant computation. This is the only speculative annotation in our benchmark programs. The profile of a four-argument predicate function is shown in Figure 5.10.

Benchmark 6: insertion sort revisited

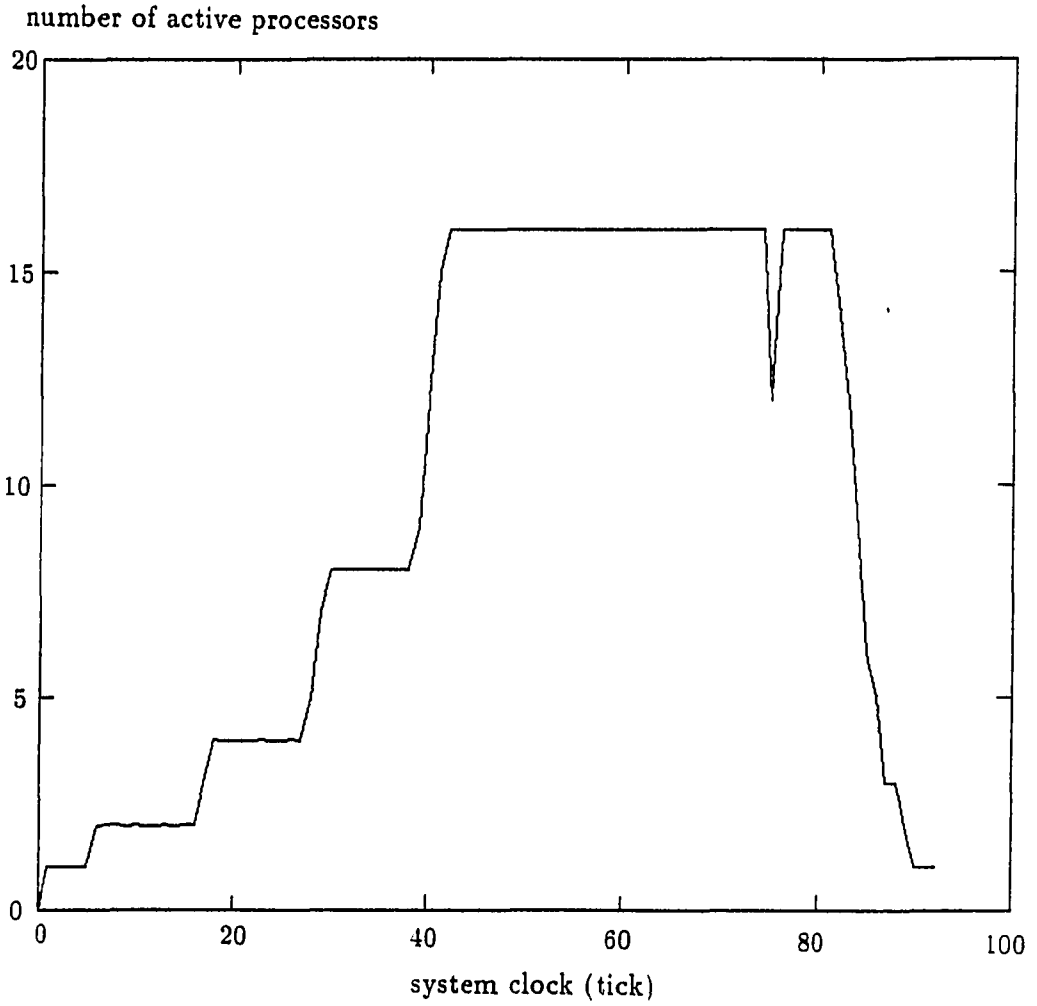
We employ the same `isort` program but with a more realistic input list. The input list now contains 100 randomly generated numbers. The profile of this input is shown in Figure 5.11.

Benchmark 7: quick sort

`Qsort` sorts by the quick sort method. Quick sort can apply divide-and-conquer strategy since the input list is divided into two sublists recursively. However, because the division depends on the value of the pivot, the two sublists may not be well-balanced and thus the load may not be evenly distributed. The profile of sorting 100 randomly generated numbers is shown in Figure 5.12.

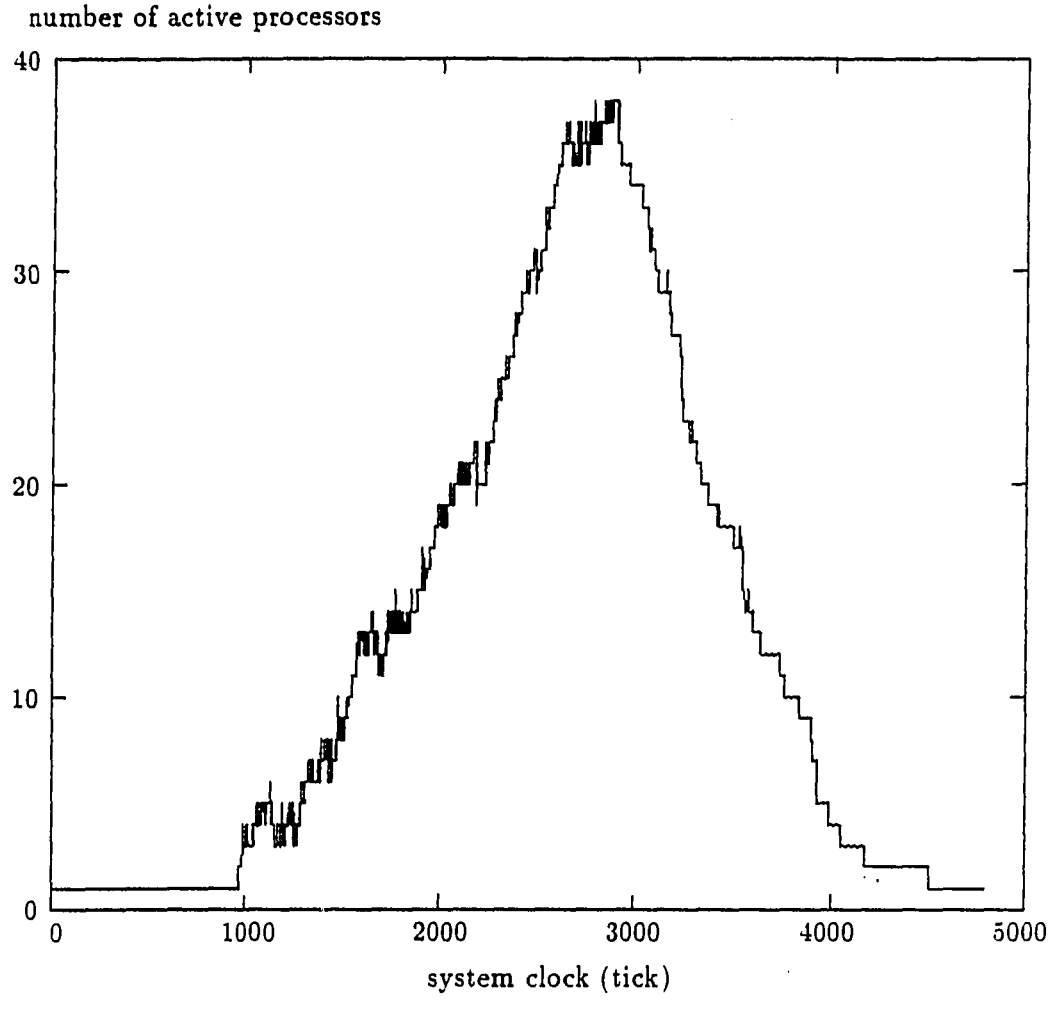
Benchmark 8: matrix multiplication

`Matrix` performs matrix multiplication. Although it is a classic benchmark for numerical computation, it is used here to test the list operations here since SASL



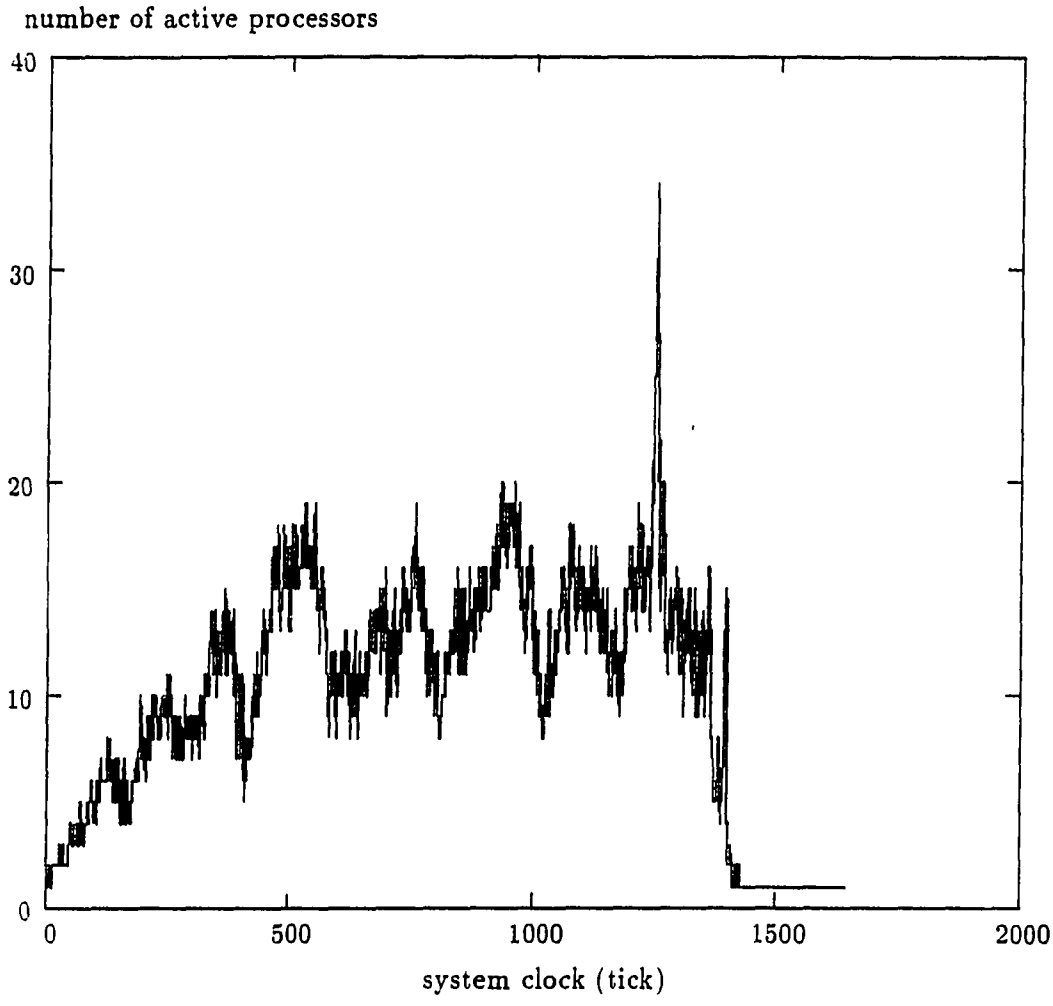
total time elapsed: 92 ticks
total work done: 883 reductions
speedup: 9.60

Figure 5.10: Profile of **taut_test**



total time elapsed: 4794 ticks
total work done: 58771 reductions
speedup: 12.26

Figure 5.11: Profile of **isort_test2**



total time elapsed: 3283 ticks
total work done: 33042 reductions
speedup: 10.06

Figure 5.12: Profile of **qsort_test**

does not provide any indexed structures or vector capability. The major part of the computation is spent in complicated list manipulation which transforms the input list into a form needed for the inner-product function. Therefore, in SASL, `matrix` is more or less a list manipulation benchmark. The profile of the multiplication of two eight by eight matrices is shown in Figure 5.13.

Benchmark 9: sparse matrix multiplication

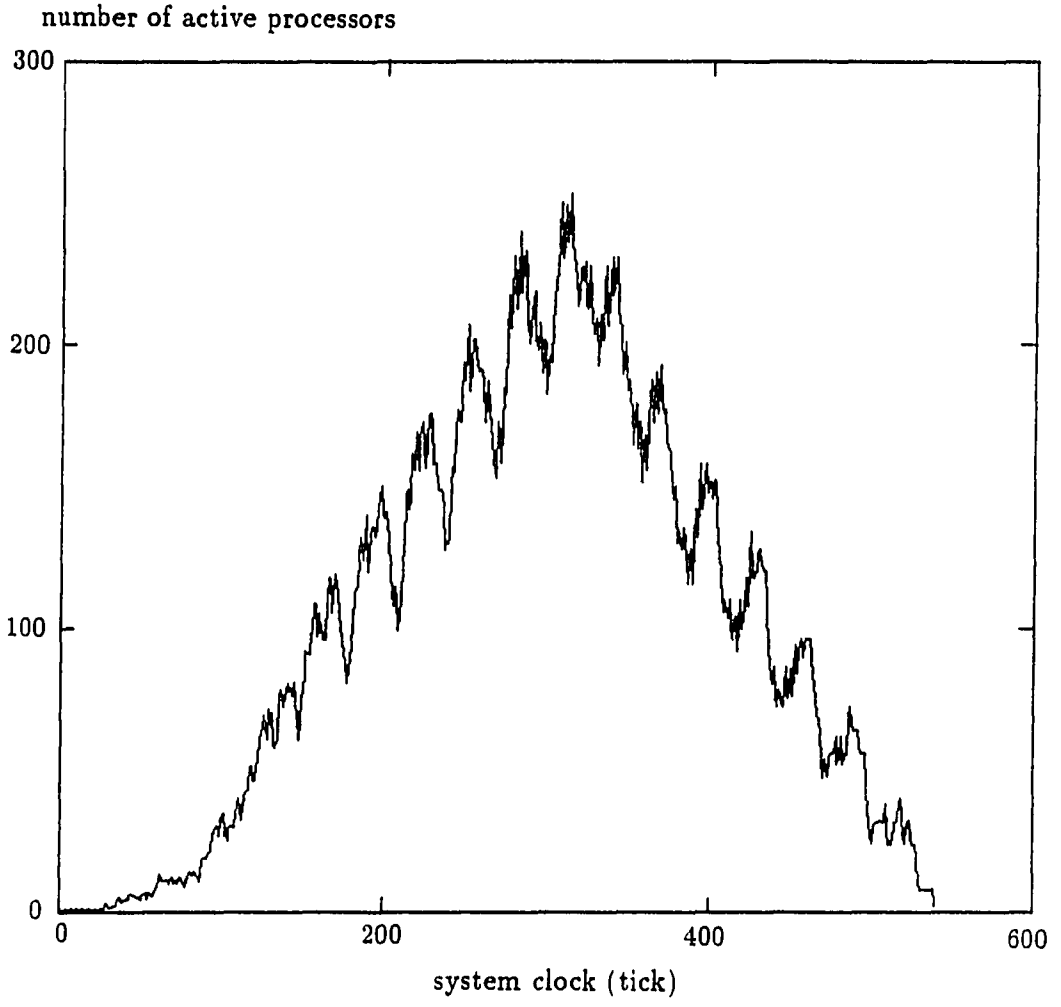
`Smatrix` performs sparse matrix multiplication. It can be efficiently represented by an adjacent list containing only non-zero entries which are represented by a tuple containing the index and value. For example, consider $(0,1,0,0,0,0,0,3)$, a single row from an 8 by 8 matrix. The new representation can be written as $((2,1),(8,3))$, which is more efficient and compact than the original 8-entry vector. This program also tests list operations. However, unlike `matrix`, the pattern of data accesses is highly irregular. The profile of the multiplication of two eight by eight matrices is shown in Figure 5.14.

Benchmark 10: convolution

`Conv` performs the convolution of two vectors. The definition of convolution can be described as follows: let $\vec{a} = (a_0,\ldots,a_{n-1})$ and $\vec{b} = (b_0,\ldots,b_{n-1})$ be two n dimension vectors, and $\vec{c} = (c_0,\ldots,c_{2n-2})$ be their convolution, then

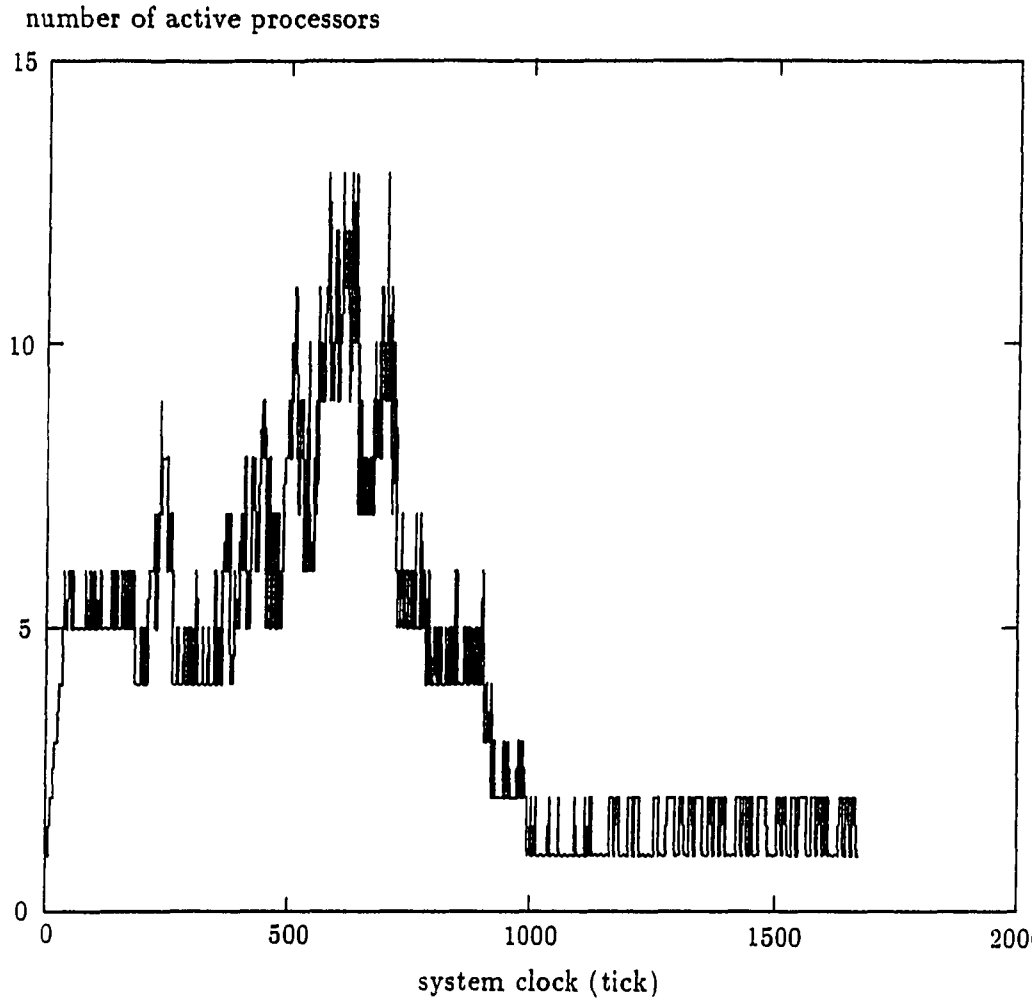
$$c_i = \sum_{j=0}^{n-1} a_j b_{i-j} \qquad\qquad a_k = b_k = 0, \text{ if } k < 0 \text{ or } k \geq n$$

Since the two lists are accessed in a regular pattern, this program is written as an operation between two streams $\vec{a}, \vec{b}$. The profile of the convolution of two eight-



total time elapsed: 539 ticks
total work done: 54574 reductions
speedup: 101.25

Figure 5.13: Profile of **matrix_test**



total time elapsed: 1667 ticks
total work done: 6639 reductions
speedup: 3.98

Figure 5.14: Profile of **smatrix_test**

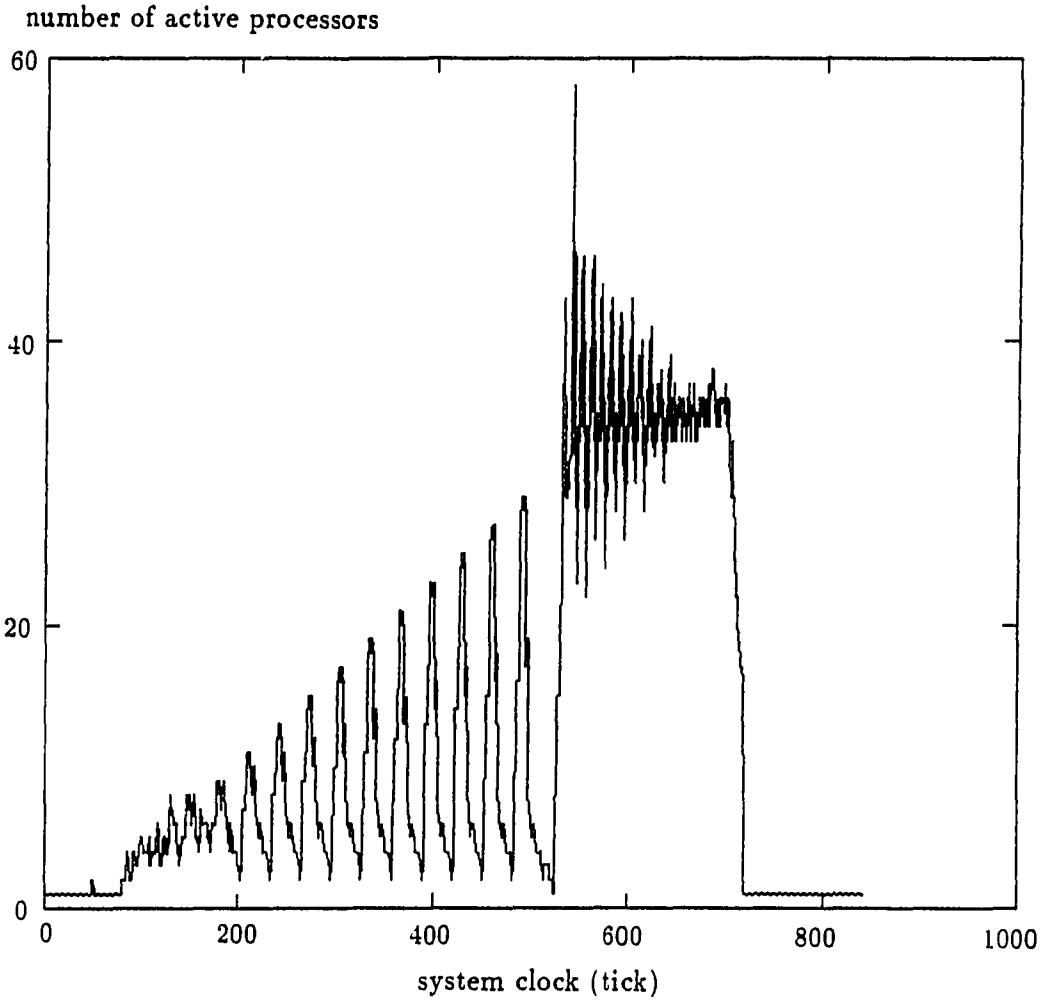
element lists is shown in Figure 5.15.

Benchmark 11: symbolic RLC impedance

Impedance finds symbolic impedance of a RLC circuit in the s-domain. This program includes two major parts: the manipulation of a polynomial and the interpretation of the input symbols. The first part is implemented as a special abstract data type, *rational polynomial*, which is represented as a numerator-polynomial and denominator-polynomial pair. Five operations, including initiation, addition, subtraction, multiplication and division, are defined on it. Previous convolution is used to find the coefficients of polynomial multiplication. The second part employs two implicit dispatchers based on the circuit type (parallel or serial) and element type (R, L or C) respectively. According to the specific type, the dispatcher invokes the corresponding routine, which is composed by the operations defined on the rational polynomial type. The profile of a twelve-element RLC circuit is shown in Figure 5.16.

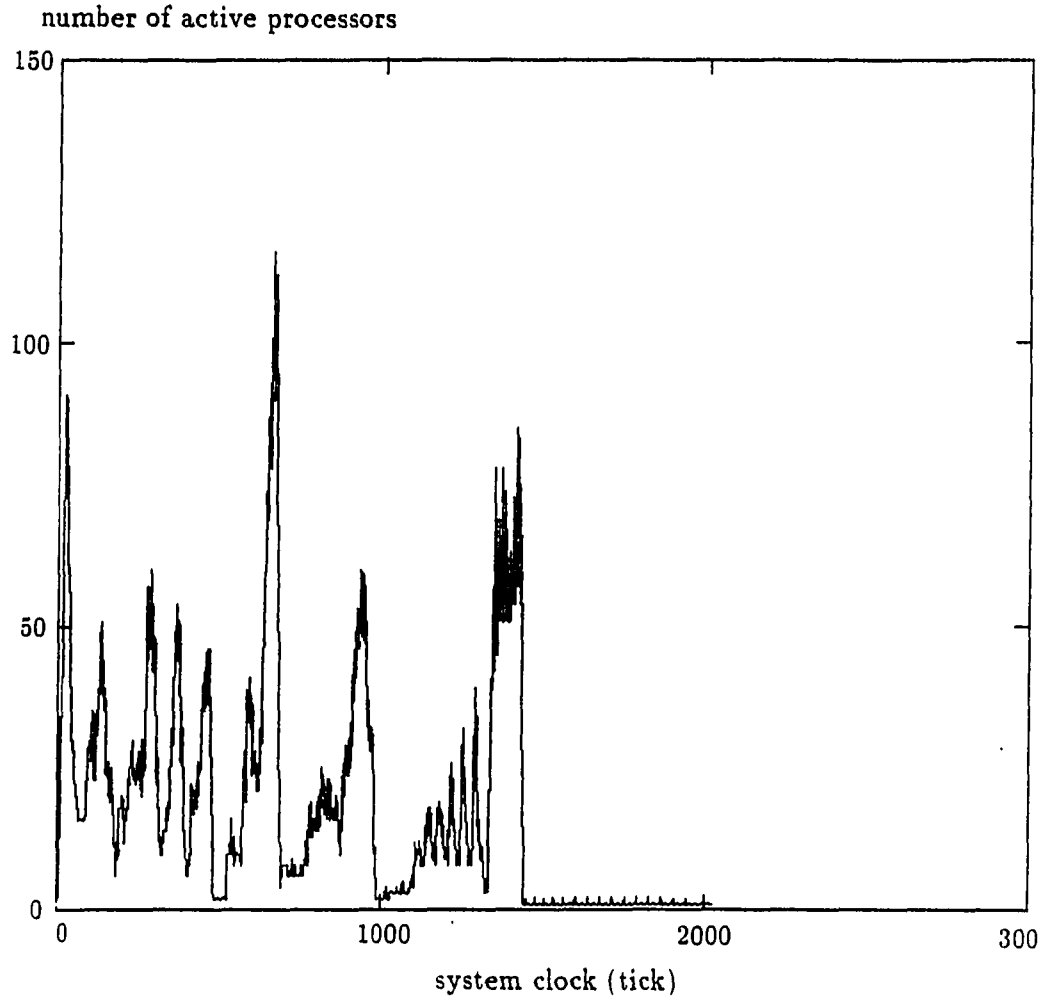
Benchmark 12: ZF set

set is a simplified version of SASL's ZF set, which provides a simple way to manipulate a list. The set first filters the input list according to a designated predicate function and then produces the new list according to a designated generating function. The profile of the operation of a ten-element list is shown in Figure 5.17.



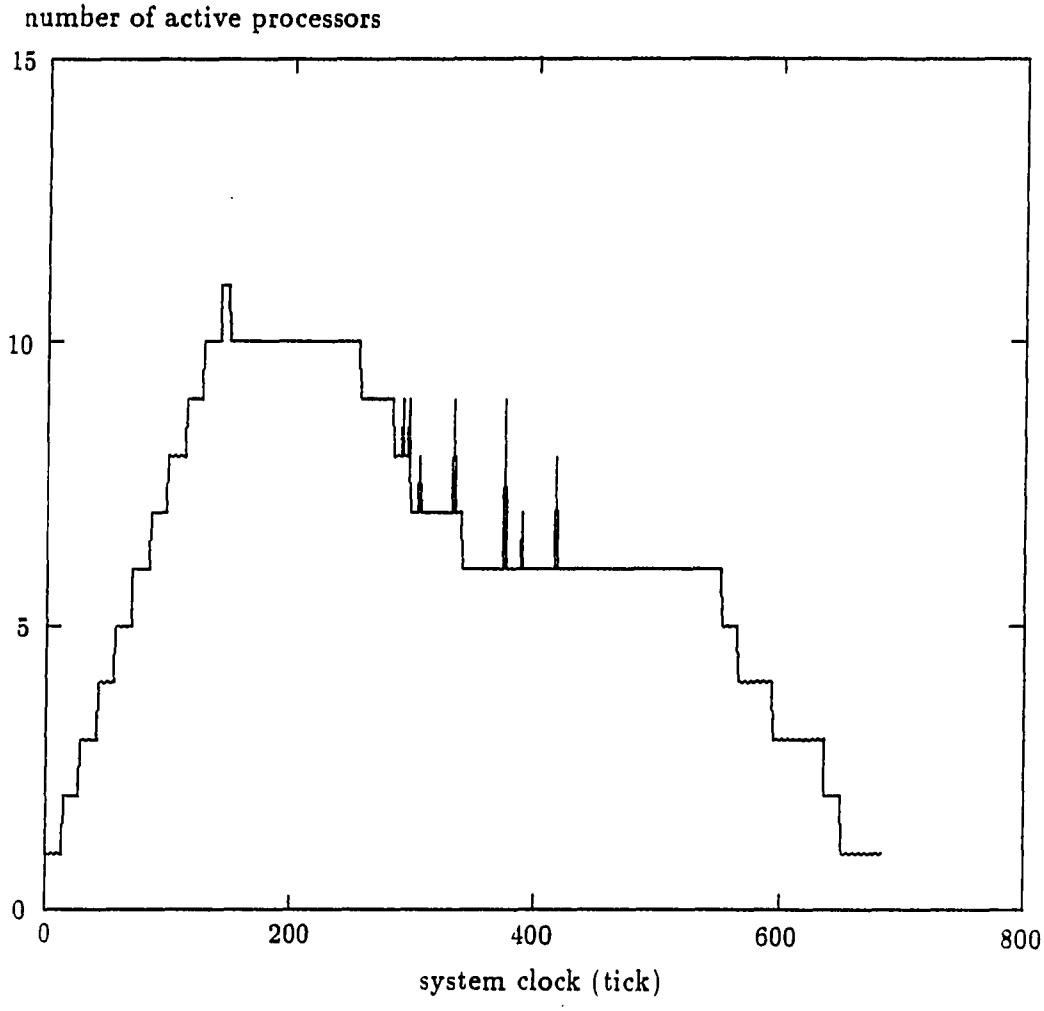
total time elapsed: 448 ticks
total work done: 3100 reductions
speedup: 6.92

Figure 5.15: Profile of **conv_test**



total time elapsed: 2022 ticks
total work done: 35933 reductions
speedup: 17.77

Figure 5.16: Profile of **circuit_test**



total time elapsed: 684 ticks
total work done: 4235 reductions
speedup: 6.19

Figure 5.17: Profile of set_test

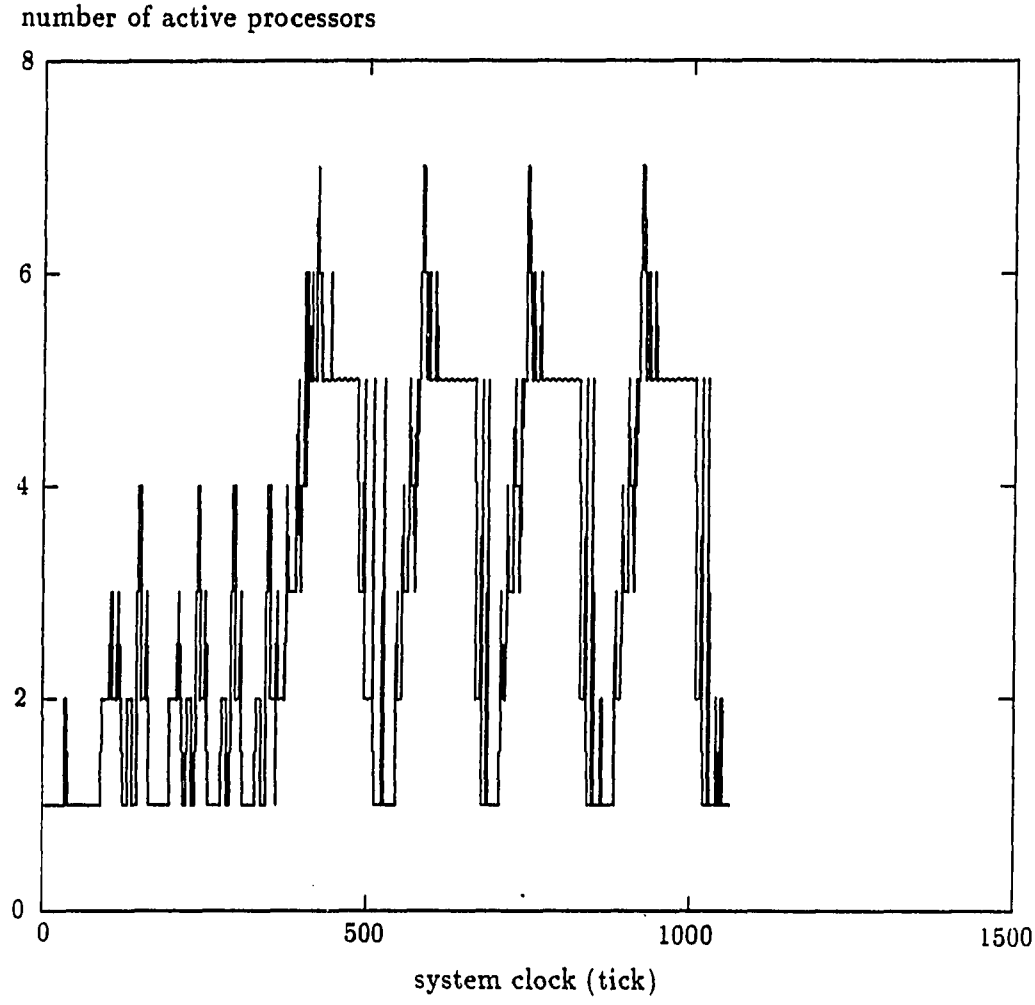
Benchmark 13: production system

expert is a toy forward-chaining production (expert) system. The input includes an assertion database, which states some initial facts, and an IF-THEN rule database, which contains a set of relevant rules. The internal control mechanism will derive all the implicit facts. Inferring new facts from a rule is the key operation of this program. It includes two major phases: the IF part will first be tested by a series of pattern matching, filtering, and variable binding; if all conditions are satisfied, then the THEN part will be bound to the corresponding value and be added into assertion database. An upper-level control will thread the individual rule and cycle the assertion database to obtain the final answer. The test database is used to guess the type of a animal, which includes four rules and five facts. The run-time profile is shown in Figure 5.18.

Summary

In this section, we have shown that the annotation scheme can extract a considerable amount of parallelism from a wide range of application programs. Most of these programs have irregular and unstructured run-time behavior and are difficult, if not impossible, to be parallelized by conventional techniques. Moreover, instead of writing new parallel versions, we keep the original sequential programs intact and just augment them with proper annotation.

Because of the complex run-time behavior, the source of the parallelism is hard to analyze. The **taut** and **qsort** benchmarks explicitly apply divide-and-conquer strategy, though the division is not well-balanced. The other benchmarks exploit



total time elapsed: 1061 ticks
total work done: 3145 reductions
speedup: 2.96

Figure 5.18: Profile of **expert_test**

parallelism from the eager evaluation of the strict arguments and from the eager evaluation of the lazy structures. The eager evaluation of lazy structures plays an important role in these programs. If the annotation can construct one or more "implicit pipelines", as in `isort` `matrix` and `set`, a significant number of operations can be overlapped and a great speedup can be achieved. On the other hand, if the pipeline operation cannot be continued, as in the `expert` and `circuit`, the number of active processors grows up and down, which is reflected in the various peaks in the plot. Thus, even though the annotation scheme can be applied to the sequential code directly, it may be sometimes necessary to rewrite programs to construct the pipeline operation.

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

Conclusions

In this dissertation, we have developed a scheme to extend the SASL system so that it can be incorporated into a multiprocessor environment. This scheme introduces extra control information which can selectively initiate the eager evaluation of the designated expressions. This information is specified by adding annotation to the original sequential SASL programs and then transferring it to the control tags of the combinators. The transfer is performed by the extended compiling and optimization algorithm, which can translate the annotation to proper control tags and propagate this information when the combinatory code is simplified.

To measure the effectiveness of our scheme, we have simulated a set of programs on an idealized shared-memory multiprocessor system. The results show that this scheme can be applied to a wide variety of applications and can extract a considerable amount of parallelism, even when the execution of the programs is irregular and unstructured. Moreover, we have found that the eager evaluation of the lazy structures is a very powerful concept. It can overlap the list operation and extract parallelism which otherwise cannot be exploited. Because of the environment-free nature of combinator-based systems, this concept introduces very little overhead in

our proposed system and therefore it can be applied in fine granularity, which in turn significantly speeds up the overall computation.

Although our scheme can be readily applied to the sequential programs, in-depth understanding of the program behavior and the internal mechanism of this scheme is still essential to exploit maximal parallelism. While some programs, like the `fib` benchmark, need to be slightly modified to increase efficiency, the other, like the `circuit` and `expert` benchmarks, may need to be redesigned to construct a pipeline so that more parallelism can be extracted. Thus, to certain degree, this scheme is still not "transparent" to the programmer and imposes some requirements on the program design.

It is concluded that our annotation scheme provides a simple but effective way to extract the fine-grain parallelism, at least at the virtual level, for non-numeric applications. It is also concluded that combinator-based systems can easily be extended so that their programs can be evaluated in parallel.

Future Work

This dissertation has investigated only one aspect of combinator-based multiprocessor systems. Future research can be continuously conducted in several areas. One major topic is whether there is a physical architecture that can efficiently support the parallel graph reduction. As in any other multiprocessor system, the major bottleneck will be the global memory and bus bandwidth. During the course of performing this research, we have collected data and analyzed subsets of many of the above problems. Our intuition would lead us to believe that this scheme would not re-

quire more hardware resources than other multiprocessor systems, and, with a proper cache organization [14] and naming scheme, the virtual system can be mapped to a real system containing a small number (e.g., 4 to 16) of processors without losing too much efficiency.

One limitation of our scheme is the slow parameter distribution, which prolongs the initial spreading phase and hinders the overall speedup. This problem is carried from the Turner's combinators, which can only distribute the parameters one level at a time in the expression tree, and cannot be accelerated in the "vertical dimension". However, we believe this limitation can be avoided if a more powerful combinator set, such as Abdali's set, is employed.

In summary, our current work can be furthered in following areas:

- Memory reference characteristics of the benchmark programs.
- Space requirement of the eagerly evaluated lazy structures.
- Design of an architecture, especially the memory organization part, to support the proposed scheme.
- Annotation to specify other control information, such as scheduling, load balancing and graph distribution.
- Control of the AND/OR parallelism from the pattern matching.
- Control tags and optimization algorithm for other combinator sets.

ACKNOWLEDGEMENTS

I would like to express my most sincere appreciation to Dr. James A. Davis, my major professor, for his guidance, support, and friendship throughout my entire Ph.D. program and all the hours devoted to the discussion of the topics related to this study. I am also grateful to Dr. Arthur V. Pohm, Dr. Vijay Vittal, Dr. Gurpur M. Prabhu and Dr. Doug W. Jacobson for their support as members of my committee.

BIBLIOGRAPHY

[1] S. Kamal Abdali. An abstraction algorithm for combinatory logic. *J. of Symbolic Logic*, 41(1)(1976):222–224.

[2] Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Language*. Addison Wesley, Reading, MA, 1987.

[3] John Allen. *Anatomy of Lisp*. McGraw-Hill, New York, NY, 1978.

[4] Paul Anderson, Chris Hankin, Paul Kelly, Peter Osmon, and Malcolm Shute. Cobweb–2: structured specification of a wafer-scale supercomputer. pages 51–67. In *Proc. of Parallel Architecture and Language Europe*, Eindehoven, Netherlands, June 1987.

[5] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8)(1978):613–641.

[6] A. J. Beaven, G. L. Burn, and R. J. Karia. Overview of a parallel reduction machine project. pages 394–413. In *Proc. of Parallel Architecture and Language Europe*, Eindehoven, Netherlands, June 1987.

[7] F. Warren Burton. Annotations to control parallelism and reduction order in the distributed evaluation of functional programs. *ACM Trans. Prog. Lang. and Systems*, 6(2)(1984):159–174.

[8] M. Castan, M. H. Durand, and M. Lemaitre. A set of combinators for abstraction in linear space. *Inf. Proc. Letters*, 24(2)(1987):183–188.

[9] M. Castan, M. H. Durand, G. Durrieu, B. Lecussan, and M. Lemaitre. MARS: a multiprocessor machine for parallel graph reduction. pages 113–121. In *The 19th Annual Hawaii International Conference on System Science*, Honolulu, Hawaii, Jan. 1984.

[10] M. Castan, G. Durrieu, B. Lecussan, M. Lemaitre, A. Contessa, E. Cousin, and P. Ng. Toward the design of a parallel graph reduction machine. pages 160–180. In *Workshop on Graph Reduction*, Santa Fe, New Mexico, Sep. 1986.

[11] T. J. W. Clarke, P. J. S. Gladstone, C. D. MacLean, and A. C. Norman. SKIM—The S, K, I Reduction Machine. pages 128–135. In *LISP Conf.*, Stanford, CA, Aug. 1980.

[12] H. K. Curry and R. Feys. *Combinatory Logic.* North-Holland, New York, NY, 1965.

[13] John Darlington and Mike Reeve. Alice—a multiprocessor reduction for the parallel evaluation of applicative languages. pages 65–75. In *Conf. on Functional Programming Language and Computer Architecture*, Portsmouth, NH, Oct. 1981.

[14] James A. Davis, William J. Armstrong, and Arthur V. Pohm. A hierarchical memory system for a fine-grain parallel processor. Unpublished report. Dept. Electrical Engineering and Computer Engineering, Iowa State Univ., Ames, Iowa.

[15] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, Cambridge, MA, 1985.

[16] Daniel Gajski and Jih-Kwon Peir. Essential issues in multiprocessor systems. *IEEE Computer*, 18(6)(1985):9–27.

[17] H. Glaser, C. Hankin, and D. Till. *Principles of Functional Programming.* Prentice Hall, Englewood Cliffs, NJ, 1984.

[18] Benjamin Goldberg and Paul Hudak. Alfalfa: distributed graph reduction on a hypercube multiprocessor. pages 94–113. In *Workshop on Graph Reduction*, Santa Fe, New Mexico, Sep. 1986.

[19] Robert H. Halstead Jr. Implementation of multilisp: lisp on a multiprocessor. pages 9–17. In *ACM Symp. on LISP and Functional Programming*, Austin, TX, Aug. 1984.

[20] Robert H. Halstead Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Prog. Lang. and Systems*, 7(4)(1985):501–538.

[21] Robert H. Halstead Jr. Parallel symbolic computing. *IEEE Computer*, 19(8)(1986):35–43.

[22] Robert H. Halstead Jr. and Tetsuya Fujita. MASA: a multithreaded processor architecture for parallel symbolic computing. pages 443–451. In *The 15th Int. Symp. on Computer Architecture*, Honolulu, Hawaii, June 1988.

[23] C. L. Hankin, G. L. Burn, and S. L. P. Jones. A safe approach to parallel combinator reduction (extended abstract). pages 99–110. In *Proc. of European Symp. on Programming*, Saarbrucken, Germany, 1986.

[24] Chris L. Hankin, Peter E. Osmon, and M. J. Shute. Cobweb: a combinator reduction architecture. pages 99–112. In *IFIP Int. Conf. on Functional Language and Computer Architecture*, Nancy, France, Sep. 1985.

[25] P. G. Harrison and M. Reeve. The parallel graph reduction machine, Alice. pages 181–202. In *Workshop on Graph Reduction*, Santa Fe, New Mexico, Sep. 1986.

[26] Pieter H. Hartel and Arthur H. Veen. Statistics on graph reduction of sasl programs. *Software Practice and Experience*, 18(3)(1988):239–253.

[27] Peter Henderson. *Functional Programming: Application and Implementation.* Prentice Hall, Englewood Cliffs, NJ, 1980.

[28] Teruo Hikita. On the average size of Turner's translation to combinator programs. *J. Information Processing*, 7(3)(1984):164–169.

[29] Mark Hill. Design decision in spur. *IEEE Computer*, 19(11)(1986):8–22.

[30] W. Daniel Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[31] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, London, 1986.

[32] Kai Huang, Joydeep Ghosh, and Raymond Chowkwanyun. Computer architecture for artificial intelligence processing. *IEEE Computer*, 20(1)(1987):19–27.

[33] Paul Hudak and Benjamin Goldberg. Serial combinators: "optimal" grains of parallelism. pages 382–399. In *IFIP Int. Conf. on Functional Language and Computer Architecture*, Nancy, France, Sep. 1985.

[34] R. J. M. Hughes. Super-combinators: a new implementation method for applicative language. pages 1–10. In *ACM Symp. on LISP and Functional Programming*, Pittsburgh, PA, Aug. 1982.

[35] Kai Hwang and Faye A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, NY, 1984.

[36] Thomas Johnson. Efficient compilation of lazy evaluation. pages 58–69. In *Proc. of ACM Symp. on Compiler Construction*, June 1984.

[37] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, Englewood Cliffs, NJ, 1987.

[38] M. S. Joy, V. J. Rayward-Smith, and F. W. Burton. Efficient combinator code. *J. Computer Lang.*, 10(3)(1985):211–224.

[39] Robert M. Keller and Frank Lin. Simulated performance of a reduction based multiprocessor. *IEEE Computer*, 17(7)(1984):70–82.

[40] Robert M. Keller, Frank Lin, and Jiro Tanaka. Rediflow multiprocessing. pages 410–417. In *IEEE COMPCON*, San Francisco, California, Feb. 1984.

[41] Robert M. Keller, Jon W. Slater, and Kelvin T. Likes. Overview of Rediflow II development. pages 203–214. In *Workshop on Graph Reduction*, Santa Fe, New Mexico, Sep. 1986.

[42] Richard Kennaway and Ronan Sleep. A new implementation technique for applicative languages. *ACM Trans. Prog. Lang. and Systems*, 10(4)(1988):602–626.

[43] Richard B. Kieburtz. The G-machine: a fast, graph-reduction evaluator. pages 400-413. In *IFIP Int. Conf. on Functional Language and Computer Architecture*, Nancy, France, Sep. 1985.

[44] Richard B. Kieburtz. Performance measurement of a G-machine implementation. pages 275–296. In *Workshop on Graph Reduction*, Santa Fe, New Mexico, Sep. 1986.

[45] Deborah L. Knox. *Combinators as Control Mechanisms in Multiprocessor Systems.* Ph.D. thesis, Iowa State University, 1987.

[46] M. Lemaitre, M. Castan, M. H. Durand, G. Durrieu, and B. Lecussan. Mechanisms for efficient multiprocessor combinator reduction. pages 113–121. In *ACM Symp. on LISP and Functional Programming*, Cambridge, MA, Aug. 1986.

[47] Kohei Noshita. Translation of Turner combinator in o(n log n) space. *Inf. Proc. Letters*, 20(2)(1985):71–74.

[48] Kohei Noshita and Teruo Hikita. The BC-chain method for representing combinators in linear space. *New Generation Computing*, 3(1985):131–144.

[49] Simon L. Peyton Jones, Chris Clack, Jon Salkild, and Mark Hardie. GRIP—a high-performance architecture for parallel graph reduction. pages 98–111. In *The 3rd Conference on Functional Language and Computer Architecture*, Portland, Oregon, Sep. 1987.

[50] John D. Ramsdell. The CURRY Chip. pages 122-131. In *ACM Symp. on LISP and Functional Programming*, Cambridge, MA, Aug. 1986.

[51] S. M. Sarwar, S. J. Hahn, and James A. Davis. Implementing functional languages on a combinator-based reduction machine. *SIGPLAN Notices*, 23(4)(1988):65–71.

[52] Mark Scheevel. NORMA: A Graph Reduction Processor. pages 212–219. In *ACM Symp. on LISP and Functional Programming*, Cambridge, MA, Aug. 1986.

[53] Harold S. Stone. *High-performance Computer Architecture.* Addison Wesley, Reading, MA, 1987.

[54] W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some practical methods for rapid combinator reduction. pages 159–166. In *LISP Conf.*, Stanford, CA, Aug. 1984.

[55] David A. Turner. Another algorithm for bracket abstraction. *J. of Symbolic Logic*, 44(2)(1979):267–270.

[56] David A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9(9)(1979):31–49.

[57] David A. Turner. *SASL Language Manual.* Univ. of Kent, Canterbury, England, 1983.

[58] Steven R. Vegdahl. A survey of proposed architecture for the execution of functional languages. *IEEE Trans. on Computers*, C-33(12)(1984):1050–1071.

[59] Paul Watson and Ian Watson. Evaluating functional programs on the FLAGSHIP machine. pages 80–97. In *The 3rd Conference on Functional Language and Computer Architecture*, Portland, Oregon, Sep. 1987.

[60] Patrick Henry Winston and Berthold Klaus Paul Horn. *Lisp.* Addison Wesley, Reading, MA, 1984.

APPENDIX. BENCHMARK PROGRAM LISTING

```
|| THE BENCHMARK PROGRAMS

|| tak

   tak x y z = ~@(y < x) -> z;
               tak @(tak @(x-1) y z) @(tak @(y-1) z x) @(tak @(z-1) x y)

   tak_test = tak 9 6 3


|| tautology
||    f is a logic predicate with n variables

   taut 0 t = t
   taut n f = taut (n-1) (f TRUE) & @(taut (n-1) (f FALSE))

   test_pred w x y z = (z & w & ~x) | (~y | x & z) | (x & y) | ~x | x
   taut_test = taut_a 4 test_pred


|| insertion sort

  isort () = ()
  isort (a:x) = insert a (isort x)
  insert a () = a,
  insert a (b:x) = a < b ->a:b:x
                   b : @(insert a x)

  isort_test2=isort
              (52,18,31,54,95,18,71,23,50,13,9,39,28,37,99,54,77,65,77,79,
              83,16,63,32,35,92,52,41,61,79,94,87,87,68,76,59,39,36,21,83,
              42,47,98,13,22,96,74,41,79,76,96,3,32,76,25,59,5,96,32,6,45,
```

```
            92,58,12,57,26,50,24,48,41,88,43,36,39,5,17,53,70,10,41,78,
            25,35,23,30,31,89,4,66,40,68,74,94,24,84,97,78,44,68,81)


|| quick sort
   app x y = y ++ x
   qsort () = ()
   qsort (a:x) = app @(a : @(qsort n)) @(qsort m)
                 WHERE   m = smallist a x
                         n = largelist a x
   smallist pivot () = ()
   smallist pivot (a:x) = pivot > a -> a:(smallist pivot x);
                                       (smallist pivot x)
   largelist pivot () = ()
   largelist pivot (a:x) = ~(pivot > a) -> a:(largelist pivot x);
                                           (largelist pivot x)


   qsort_test=qsort
              (52,18,31,54,95,18,71,23,50,13,9,39,28,37,99,54,77,65,77,79,
              83,16,63,32,35,92,52,41,61,79,94,87,87,68,76,59,39,36,21,83,
              42,47,98,13,22,96,74,41,79,76,96,3,32,76,25,59,5,96,32,6,45,
              92,58,12,57,26,50,24,48,41,88,43,36,39,5,17,53,70,10,41,78,
              25,35,23,30,31,89,4,66,40,68,74,94,24,84,97,78,44,68,81)
```

```
|| matrix.s - matrix multiplication

   pmap f () = ()
   pmap f (h:t) = @(f h) : @(pmap f t)
   pzip x = hd x = () -> ()
            pmap hd x: pzip (pmap tl x)
   mult x y = pmap (pmap IP) (pmap distl (distr (x, pzip y)))
   IP x = sum (pmap product (pzip x))
   distl (a,x) =  pmap (pair a) x
   distr (a,x) =  pmap (rpair x) a
   pair a b = a,b
   rpair a b = b,a

   m8 = ((1,2,3,4,5,6,7,8),
         (1,2,3,4,5,6,7,8),
         (1,2,3,4,5,6,7,8),
         (1,2,3,4,5,6,7,8),
         (1,2,3,4,5,6,7,8),
         (1,2,3,4,5,6,7,8),
         (1,2,3,4,5,6,7,8),
         (1,2,3,4,5,6,7,8),
         (1,2,3,4,5,6,7,8),
         (1,2,3,4,5,6,7,8))
    matrix_test = mult m8 m8


|| multiplication of sparse matrices
||
|| matrix is represented as a list of (row-index row-vector*)
|| e.g.  | 0 4 0 |     ( (1,    (2,4)   )
||      | 0 0 0 |  is   (3, (1,5),(2,6))
||      | 5 6 0 |     )

   scalev s vector = pmap (scale s) vector
   scale s (index,val) = (index, s*val)
|| addv - add vector a, b
   addv x () = x
   addv () y = y
   addv (ha:ta) (hb:tb) = (hd ha) < (hd hb) -> ha: @(addv ta (hb:tb))
                          (hd ha) > (hd hb) -> hb: @(addv (ha:ta) tb)
                          @((hd ha),(hd (tl ha)+hd (tl hb))):@(addv ta tb)
|| multvm - multiplication of a vector and matrix (v * m)
```

```
|| multr - constructing a complete row of (v * m)
   multvm m () = ()
   multvm () v = ()
   multvm (hm:tm) (hv:tv) = (hd hv) < (hd hm) -> (multvm (hm:tm) tv)
                            (hd hv) > (hd hm) -> (multvm tm (hv:tv))
                            addv (scalev (hd (tl hv)) (tl hm))
                                 @(multvm tm tv)
   multr m (h:vec) = h:@(multvm m vec)
|| multmm - multiplication of a matrix m, n (m * n)
   mult m n = pmap (multr n) m

   sparse_test  = mult
                  ((2,(1,5),(3,4),(6,2),(7,3)),
                  (3,(2,2),(5,8)            ),
                  (4,(3,2),(4,8)            ),
                  (6,(1,2),(5,8),(6,2),(7,4),(8,1)),
                  (8,(1,2),(2,5),(5,8),(6,2),(7,4),(8,1))
                  )
                  ( (1,(1,2),(3,4),(6,9),(8,2)),
                    (2,(2,3),(3,6)),
                    (5,(1,2),(3,4),(4,7),(5,1),(6,9),(8,2)),
                    (6,(5,1),(6,9),(8,2)),
                    (7,(6,9),(8,2)),
                    (8,(5,9),(7,2))
                  )
```

```
|| conv.s - convolution of two vector

   reverse = foldl cons ()
   foldl op r ()    = r
   foldl op r (a:x) = foldl op @(op a r) x
|| rev c l n - reverse list n with |c|-1 leading 0's; l is for temp storage
   rev ()   l () = l
   rev (a,) l () = l
   rev ()   l (h:t) = rev () (h:l) t
   rev (a,) l (h:t) = rev () (h:l) t
   rev (hc:tc) l () = rev tc (0:l) ()
   rev (hc:tc) l (h:t) = 0:@(rev tc (h:l) t)
|| shift l - generate a list of "shifted list"
||          e.g. shift (1,2,3) = (1,2,3),(2,3),(3,)
   shift (a,)  = (a,),
   shift (h:t) = @(h:t) : @(shift t)
   bimap f () q = ()
   bimap f p () = ()
   bimap f (hp:tp) (hq:tq) = @(f hp hq) : @(bimap f tp tq)
   prod_list m n = pmap (bimap times m) (shift (rev m () n))
   conv m n = reverse (pmap sum (prod_list m n))

   conv_test = conv (1,2,3,4,5,6,7,8) (8,7,6,5,4,3,2,1)


|| circuit

   bimap1 unit f () () = ()
   bimap1 unit f () (hq:tq) = @(f unit hq) : @(bimap1 unit f () tq)
   bimap1 unit f (hp:tp) () = @(f hp unit) : @(bimap1 unit f tp ())
   bimap1 unit f (hp:tp) (hq:tq) = @(f hp hq) : @(bimap1 unit f tp tq)
   addv m n = bimap1 0 plus m n
   diffv m n = bimap1 0 minus m n
               WHERE minus x y = x - y
|| polynomial operation:
   makep d n = (hd d)=0 & (hd n)=0 -> makep (tl d) (tl n); (d,n)
   den p = hd p
   num p = hd (tl p)
   prodp p1 p2 = makep @(conv @(den p1) @(den p2))
                       @(conv @(num p1) @(num p2))
   quotp p1 p2 = makep @(conv @(den p1) @(num p2))
                       @(conv @(num p1) @(den p2))
```

```
   sump  p1 p2 = makep @(conv @(den p1) @(den p2))
                       @(addv @(conv (den p1) (num p2))
                       @(conv (num p1) (den p2)))
   diffp p1 p2 = makep @(conv @(den p1) @(den p2)) @(diffv @(conv @(num p1)
                       @(den p2)) @(conv @(den p1) @(num p2)))

|| circuit simulation - find impedance (in the s-domain) of the circuit
   L h = makep (1,) (0,h)
   R r = makep (1,) (r,)
   C c = makep (0,c) (1,)
   S z1 z2 = sump z1 z2
   P z1 z2 = quotp @(prodp z1 z2) @(sump z1 z2)

|| a RLC with 12 elements
   circuit_test = S (P (S (C 2)
                          (P (R 6)
                             (L 7)))
                       (P (R 4)
                          (S (L 3)
                             (C 2))))
                    (P (S (C 2)
                          (P (R 6)
                             (L 7)))
                       (P (R 4)
                          (S (L 3)
                             (C 2))))
```

```
|| set: simulate SASL's list comprehension (or ZF set)

   app x y = @y ++ @x
   filter p ()    = ()
   filter p (a:x) = app @(filter p x) @(p a -> a,;())
   set f p ()    = ()
   set f p (a:x) = app @(set f p x) @(p a -> @(f a),; ())

   f1 x = x & x & x & x & x & x & x & x & x
   f2 x = f1 (f1 x)
   f3 x = f2 (f2 x)
   f4 x = f3 (f3 x)
   set_test = set f4 f4
              (TRUE,TRUE,FALSE,TRUE,FALSE,FALSE,TRUE,TRUE,FALSE,TRUE)
```

```
|| expert: a forward-chaining production system

   append x y = @x++@y
   reverse = foldl cons ()
   foldl op r ()    = r
   foldl op r (a:x) = foldl op @(op a r) x
   pmap f () = ()
   pmap f (a:x) = @(f a) : @(pmap f x)

   fetch pat dat = (match pat dat) -> dat;()
   pat_ind x = hd x
   pat_var x = tl x

   makea var val = (var,val)
   var elem = hd elem
   val elem = hd (tl elem)

   match alist () () = alist
   match alist () x  = ()
   match alist x  () = ()
   match alist (h:pat) (d:dat)
    = (pat_ind h)=%> -> match (append @alist @((makea (pat_var h) d),))
                        pat dat
      (pat_ind h)=%< -> match alist (@(get_val (pat_var h) alist):@pat)
                        (d:dat)
      (h=d) -> match alist pat dat
      ()

   get_val name () = ()
   get_val name (h:t) = (var h)=name -> val h; get_val name t

   filter_assertion pat a_base init_alist
    = mapc exist @(match init_alist pat) a_base

   exist x = ~(x=())

   filter_a_stream a_base a_stream pat
    = foldr append () @(pmap (filter_assertion pat a_base) a_stream)

   cascade_patterns a_base a_stream () = a_stream
   cascade_patterns a_base a_stream (p:pats)
    = filter_a_stream a_base @(cascade_patterns a_base a_stream pats) p
```

```
update () a_base = a_base
update x  a_base = (x:a_base)

replace_var alist () = ()
replace_var alist (h:t)
  = (pat_ind h)=%< -> (get_val (pat_var h) alist):(replace_var alist t);
                      h : replace_var alist t

spread_actions a_base () alist = a_base
spread_actions a_base (a:actions) alist
 = spread_actions new_base actions alist
   WHERE new_base = update (replace_var alist a) a_base

feed_to_actions actions a_base () = a_base
feed_to_actions actions a_base (a:a_stream)
 = feed_to_actions actions f_base a_stream
   WHERE f_base = spread_actions a_base actions a

use_rule (assertion,action) a_base
 = feed_to_actions action a_base @(cascade_patterns a_base ((),)
                  (reverse assertion))

forward_chain a_base () = a_base
forward_chain a_base (r:rules)
 = forward_chain @(use_rule r a_base) rules

mapc test f ()    = ()
mapc test f (h:t) = append @(test tmp -> tmp,;()) @(mapc test f t)
                    WHERE tmp = f h

animal_rule =
(
 ( ( ('>animal", 'has", 'hair"),),
   ( ('<animal", 'is", 'mammal"),)
 ),
 ( ( ('>animal", 'eats", 'meat"),),
   ( ('<animal", 'is", 'carnivore"),)
 ),
 ( ( ('>animal", 'has", 'pointed", 'teeth"),
     ('<animal", 'has", 'claws"),
     ('<animal", 'has", 'forward", 'eyes")),
```

```
    ( ('<animal", 'is", 'carnivore"),)
  ),
  ( ( ('>animal", 'is", 'mammal"),
      ('<animal", 'is", 'carnivore"),
      ('<animal", 'has", 'tawny", 'color"),
      ('<animal", 'has", 'dark", 'spots")),
    ( ('<animal", 'is", 'cheetah"),)
  )
)

animal_base =
(
   ('rob", 'has", 'dark", 'spots"),
   ('rob", 'has", 'tawny", 'color"),
   ('rob", 'eats", 'meat"),
   ('rob", 'has", 'hair"),
   ('suz", 'has", 'feather")
 )

expert_test = forward_chain animal_base animal_rule
```