Using the Run-Time Sizes of Data Structures to Guide Parallel-Thread Creation

Lorenz Huelsbergen
AT&T Bell Laboratories
lorenz@research.att.com

James R. Larus
University of Wisconsin–Madison
larus@cs.wisc.edu

Alexander Aiken
University of California–Berkeley
aiken@cs.berkeley.edu

Abstract

Dynamic granularity estimation is a new technique for automatically identifying expressions in functional languages for parallel evaluation. Expressions with little computation relative to thread-creation costs should evaluate sequentially for maximum performance. Static identification of such threads is however difficult. Therefore, dynamic granularity estimation has compile-time and run-time components: Abstract interpretation statically identifies functions whose complexity depends on data structure sizes; the run-time system maintains approximations to these sizes. Compiler-inserted checks consult this size information to make thread creation decisions dynamically.

We describe dynamic granularity estimation for a list-based functional language. Extension to general recursive data structures and imperative operations is possible. Performance measurements of dynamic granularity estimation in a parallel ML implementation on a shared-memory machine demonstrate the possibility of large reductions (> 20%) in execution time.

1 Introduction

Functional languages do not overly constrain a program's evaluation order with data dependences. This simplifies automatic parallelization: multiple arguments in a strict function application can evaluate in parallel, for example. Abundant parallelism, however, does not directly lead to effective parallel implementations. Efficient implementation of a functional language on a parallel architecture remains difficult in part because the creation of a parallel thread incurs considerable overhead costs [14, 21, 23, 20].

For an implementation to be efficient, it must decide which parallelism in a program is beneficial; that is, whether parallel evaluation of a given expression will speed program execution. If an expression contains less computation than the cost of creating a thread for the expression, parallel evaluation of the thread will slow program execution (*cf.* [6]). Figure 1 shows the effect that scheduling overheads can have on overall execution times.

In this paper, we present a new technique, ***dynamic granularity estimation*** (**dge**), that uses the run-time sizes of data structures to create parallel threads only when they are known to be beneficial. This technique is based on the observation that a function's time complexity often depends on the size of the dynamic data with which it computes. For simplicity, we describe **dge** for lists—the general scheme can, however, be applied to programs that manipulate other data structures (*e.g.*, trees, DAGs, and arrays).

In a list-based language, **dge** conservatively determines, for a program function f applied to a list parameter l, the lengths of l for which the cost of computing the application $e \equiv (f\ l)$ always exceeds the overhead of creating a thread for e's concurrent evaluation. Initial empirical evidence, gathered in an implementation of **dge** in Standard ML of New Jersey (SML/NJ) [2] on a parallel shared-memory machine, suggests that the run-time costs of **dge** are small and that **dge** can substantially reduce a program's parallel execution time.

Dynamic granularity estimation is a hybrid; it is composed of dynamic and static components [17, 16]. Hybrid techniques are necessary for language parallelization since purely-static analyses are fundamentally limited. Static analysis for **dge** is in the form of an ***abstract interpretation*** [5, 1] that identifies functions whose time complexity is dependent on the sizes of the list data structures passed to them as parameters. The compiler statically identifies program points at which the length of a list always influences the cost of an application expression. When evaluation reaches such a point, compiler-inserted code consults an approximation to the list's length (maintained dynamically) to determine whether it is beneficial to evaluate an application as a separate parallel thread. The dynamic component of **dge** approximates list lengths at run time.

The quicksort function (qs) of Figure 2 provides an example. In qs, the arguments to **append** can evaluate in parallel. Parallel evaluation of these arguments is advantageous if the costs of the recursive applications of qs exceed the cost of creating and scheduling them as parallel threads. However, when the length of a sublist (l or g) is small (*e.g.*, zero), creating a parallel thread to sort the sublist is counterproductive. In this case, the arguments to **append** should evaluate sequentially. The static analysis of **dge** identifies list lengths for which

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

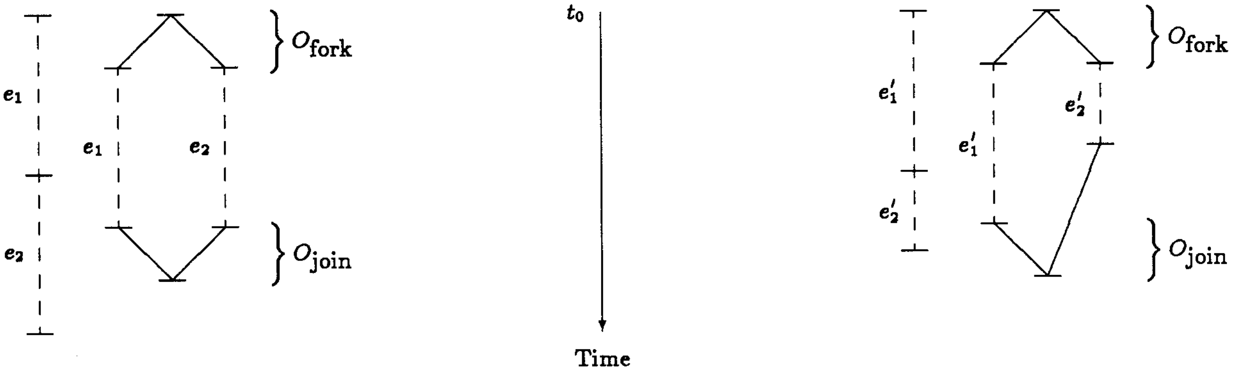

Figure 1: The impact of overhead. Time starts at t_0. The concurrent evaluation of e_1 and e_2, with overhead ($O = O_{\text{fork}} + O_{\text{join}}$) taken into account, completes before their sequential evaluation and is therefore beneficial. Concurrent evaluation of e_1' and e_2', however, slows the program's evaluation since e_2' does not contain enough computation to offset scheduling overheads.

```
fun qs p [] = []
 | qs p (x::xs) =
    let fun split l =
        let fun split' [] less greater = (less,greater)
           | split' (y::ys) less greater =
              if (p y x) then
                split' ys (y::less) greater
              else
                split' ys less (y::greater)
        in
          split' l [] []
        end
      val (l,g) = split xs
    in
      if (l̄ > cutoff) andalso (ḡ > cutoff) then
        append|| (qs p l) (x::(qs p g))
      else
        append (qs p l) (x::(qs p g))
    end
```

Figure 2: Functional quicksort automatically restructured by dynamic granularity estimation. Static analysis determines that the amounts of computation in the arguments to append depend on the lengths (denoted $\bar{l}$ and $\bar{g}$) of the sublists produced by split. The compiler inserts a run-time check (the conditional in qs's body) to examine the lengths of l and r (stored with the list representation). Based on these dynamic lengths, the check decides whether to create parallel threads ($\text{append}_{||}$ evaluates its arguments in parallel). The compiler also deduces the cutoff value.

the cost of applying qs to a list of that length is always greater than the overhead incurred in creating a new thread for the application's concurrent evaluation. At run time, **dge** approximates the lengths of all lists; the length information of the lists bound to the identifiers l and g in the qs function is available for making the final parallelization decision.

Dynamic techniques, like **dge**, that examine the sizes of data structures to conditionally select parallel evaluation are necessary since compile-time expression scheduling is fundamentally limited. This is evident from the qs example. When a statically-unknown list reaches qs, the sublist partition that qs's auxiliary `split` function creates is also unknown. Therefore, the costs of the recursive applications of qs that sort the sublists cannot be known at compile time. In the absence of precise static information about qs's list parameter, it is not possible to statically decide when concurrent evaluation of qs's recursive applications is advantageous.

In languages with explicit constructs for thread creation and synchronization, programmers typically use *cutoff* values to curb parallelism and to ensure that the program only creates large threads [11]. In the qs example, the programmer might explicitly check if the sublist being sorted contains $> k$ elements for some small k before creating parallel threads for append's arguments. Code remains portable with **dge** since the language's implementation—not the programmer—matches a cutoff to the underlying parallel architecture. The granularity of parallel threads is less of a programming issue when thread sizes are determined automatically.

In the next section, we describe the language under consideration for dynamic granularity estimation and introduce terminology. We then describe **dge**'s static (§3.1) and dynamic (§3.2) components, illustrate **dge**'s operation with examples (§4), present possible extensions to general data structures and mutable data (§5), and describe an initial implementation of this new technique (§6) and discuss results (§7).

2 Preliminaries

The language under consideration for dynamic granularity estimation is the λ_v-calculus, a functional[1], call-by-value, higher-order language [24, 26]. The ground terms of λ_v are variables and constants:

$$\begin{array}{lcl} x & \in & \textsc{Var} \\ b & \in & \textsc{Const} = \{\texttt{nil}, \texttt{true}, \texttt{false}\} \end{array}$$

The terms of λ_v are expressions ($e \in \textsc{Exp}$) and values ($v \in \textsc{Val} \subset \textsc{Exp}$):

$$\begin{array}{rcl} e & ::= & v \\ & | & e\ e \\ & | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\ & | & \texttt{cons}\ e\ e \\ & | & \texttt{hd}\ e \\ & | & \texttt{tl}\ e \\ & | & \texttt{isnull}\ e \end{array} \qquad \begin{array}{rcl} v & ::= & b \\ & | & x \\ & | & \lambda x.e \end{array}$$

We assume that λ_v terms are well-typed.

For simplicity, we focus on the ***list*** as the dynamic structure for dynamic granularity estimation. This is because a list's size is simply its length. The syntax of λ_v therefore contains `cons`, `hd`, `tl`, and `isnull` directly. Section 5 describes possible extension of **dge** to general recursive datatypes that give rise to trees, for example.

Denote the time required to evaluate an expression e as $|e|$, the *cost* of e. The cost of a parallel thread to evaluate e is $|e|$ plus the overhead, O, required to create and schedule a parallel thread.[2] Let $T \geq O$ be a machine-dependent cost *threshold* so that if $|e| > T$ then expression e is a candidate for parallel evaluation (*cf.* Figures 1 and 2). Costs are measured in integer *evaluation units* (e-units). An e-unit corresponds to—again for simplicity—the operational notion of function application [7]. For a given implementation, normalization of e-units is necessary since all function applications do not have identical costs (*e.g.*, functions may be compiled in line).

For λ_v, we assume that the evaluation of variables, constants and λ-abstractions incurs no cost (zero e-units) and that the evaluation of the other language terms costs one e-unit. Under these simplifying assumptions, for example, the application $(f\ \ (g\ \ l))$, where f and g are functions and l is a list, incurs a cost of at least two e-units (the applications of f and g each cost one), but complete evaluation of $(f\ \ (g\ \ l))$ may require many more e-units and may depend on the size (length) of l.

The length of list l is written as $\bar{l}$. When i is a natural number, $\bar{i}$ represents any list of length i.

We further use the following notation. If A and B are sets, then $A \cup B$ is their union, $A \cap B$ is their intersection, and $A \backslash B$ is their difference. The empty set is denoted by $\emptyset$, and $\mathrm{Fin}(A)$ denotes the set of finite subsets of A. If f is a map, then the domain and range of f are $\mathrm{Dom}(f)$ and $\mathrm{Rng}(f)$. A finite map from A to B is a partial map with finite domain. Denote the set of finite maps from A to B as $A \overset{fin}{\to} B$ where any $f \in A \overset{fin}{\to} B$ can be written as $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$. The empty map is written $\{\}$. If f and g are maps, then $f \pm g$ is the map with f modified by g and has the domain $\mathrm{Dom}(f) \cup \mathrm{Dom}(g)$ and the values:

$$(f \pm g)(a) = \begin{cases} g(a) & \text{if } a \in \mathrm{Dom}(g) \\ f(a) & \text{otherwise} \end{cases}$$

A *sequent* of the form $A \vdash phrase \to B$ holds, with respect to A, if $phrase \to B$ where $\to$ is some ternary relation between A, *phrase*, and B. An inference rule has the form

$$\frac{P_1 \cdots P_n}{C}$$

[1] Restriction to a functional language allows efficient implementation of dge's dynamic component that must approximate the sizes of dynamic data at run time. In a functional language, a datum d's size can only monotonically increase whereas, in a language with assignment to reference values, d's size can decrease and the efficient propagation of d's new (reduced) size estimate to other data that share d is difficult. Section 5 describes possible methods for estimating data sizes in imperative dynamic languages.

[2] It is assumed that the cost of creating and scheduling a thread is bounded and can be (empirically) determined for a given language implementation and machine architecture.

where $n > 0$. Successful inference of the premises, P_i, infers the conclusion C. The premises are either sequents or mathematical side conditions.

3 The New Technique

Dynamic granularity estimation deduces at compile time for a program function **f** whether **f**'s complexity depends on the sizes of **f**'s list parameters. This information is then used by the compiler to restructure an application $e \equiv$ (**f** l). The compiler inserts a check of l's length that selects parallel evaluation of e only when e contains enough computation to warrant its parallel evaluation. The deduction of a function's evaluation cost relative to its list parameters and subsequent program restructuring (check insertion) constitute **dge**'s static component. The dynamic component of **dge** maintains lengths with lists at run time. This section first describes **dge**'s static component and then its dynamic component.

3.1 Static Component

The idea is to abstractly evaluate, at compile time, an application $e \equiv (f\ l)$ while counting the number of e-units required. The static e-unit count thus obtained is conservative; that is, static estimation of e-units does not overestimate the number of e-units that evaluation of an expression requires. For example, if static analysis of e indicates that $|e| = i$, then actual evaluation of e must require $\geq i$ e-units. Since the aim is to identify functions whose list parameters control their complexity, an abstract semantics that interprets a list l as its length, $\bar{l}$, is used. E-units are (conservatively) counted under this abstract semantics. We first give the standard semantics for the language and then the abstract semantics. To guarantee the termination of abstract evaluation, it is also necessary to bound the number of abstract evaluation steps (§3.1.3). This bound is naturally the threshold T (§2) at which parallel evaluation of a thread becomes beneficial (*i.e.*, overcomes scheduling overheads).

3.1.1 Standard Semantics $\mathcal{S}$

The dynamic objects of the standard semantics $\mathcal{S}$ are in Figure 3. Since the list is the dynamic structure of interest for granularity estimation, it is directly represented with dynamic objects rather than indirectly encoded in λ_v: The constant *nil* is the empty list and a *cons* pair $\langle v, l\rangle$ contains an element v and the list's tail l.

Figure 4 gives a standard semantics for the language. The operational style of the semantics is derived from Tofte's semantics [29]. The semantics given here, however, also contains ***integer time annotations*** that indicate the number of e-units that an expression's evaluation requires. The evaluation relation $E \vdash e \longrightarrow_i v$ (where $E \in$ Env, $e \in$ Exp, $v \in$ DVal, and $i \in \mathbf{Z}$) indicates that the evaluation of expression e to value v with respect to environment E requires i e-units. For example, the **app** rule states that if the evaluation of e_1 to v_1 requires a e-units, the evaluation of e_2 to v_2 requires b e-units, and the application of v_1 to v_2 requires c e-units, then the evaluation of the application $(e_1\ e_2)$ requires $1 + a + b + c$ evaluation units. Similarly, conditional evaluation (**if** rule) counts e-units only in the evaluation of the branch expression selected by the conditional's predicate. Note that the evaluation of λ_v's value terms (*e.g.*, variables and λ-abstractions) requires zero e-units under this relation; a specific implementation would, however, use an e-unit measure and evaluation rules that reflect their concrete costs.

3.1.2 Abstract Semantics $\mathcal{A}$

A non-standard (abstract) semantics $\mathcal{A}$ that abstracts lists as their lengths is used for counting e-units for dynamic granularity estimation. This analysis determines whether an application $(f\ l)$ will always require at least i (where $i \geq 0$) e-units of evaluation for a given length of l. The dynamic objects of the abstract semantics are in Figure 5. Every abstract object V denotes a set of values of the standard semantics $\sigma(V)$:

$$\begin{aligned}
\sigma(\{v_1, \ldots, v_n\}) &= \bigcup_{i=1}^{n} \sigma(\{v_i\}) \\
\sigma(\{\mathit{true}\}) &= \{\mathit{true}\} \\
\sigma(\{\mathit{false}\}) &= \{\mathit{false}\} \\
\sigma(L_k) &= \{l \mid l \text{ is a list of length} \geq\ k\} \\
\sigma(\left\{E^{\mathcal{A}}\right\}) &= \left\{f \mid \forall x \in \mathrm{Dom}(E^{\mathcal{A}}), f(x) \in E^{\mathcal{A}}(x)\right\} \\
\sigma(\left\{\left[x, e, E^{\mathcal{A}}\right]\right\}) &= \left\{[x, e, E] \mid E \in \sigma(E^{\mathcal{A}})\right\} \\
\sigma(\top^{\mathcal{A}}) &= \{v \mid v \in \mathrm{DV{\small AL}}\}
\end{aligned}$$

A list of length k in the abstract semantics is represented by L_k, the set of all lists with at least k elements.[3] An environment (Env$^{\mathcal{A}}$) maps a program variable either to a concrete finite subset of values or to any such subset (denoted $\top^{\mathcal{A}}$).

The upper bound operation $\sqcup$ on dynamic objects X and Y is defined:

$$X \sqcup Y = \begin{cases} \top^{\mathcal{A}} & \text{if } X = \top^{\mathcal{A}} \text{ or } Y = \top^{\mathcal{A}} \\ L_i & \text{if } X = L_i \text{ and } Y = L_j \text{ and } i \leq j \\ X \cup Y & \text{otherwise} \end{cases}$$

The operator $\sqcup$ is set union, except that $\top^{\mathcal{A}}$ absorbs all other values and that list abstractions combine conservatively.

The relation for abstract evaluation, $E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_i V$ (where $E^{\mathcal{A}} \in$ Env$^{\mathcal{A}}$, $e \in$ Exp, $V \in$ DValSet$^{\mathcal{A}}$, and $i \in \mathbf{Z}$), evaluates expression e with respect to (abstract) environment E$^{\mathcal{A}}$ to a ***set*** of values V. This relation is defined such that when $e \xrightarrow{\mathcal{A}}_i V$ and $e \longrightarrow_j v$ then $v \in \sigma(V)$ and $i \leq j$. That is, the set of values computed by the abstract relation always contains e's actual value (as produced by $\mathcal{S}$). Furthermore, the e-unit count produced by the abstract semantics is conservative; standard evaluation of e under $\mathcal{S}$ always requires at least i e-units when abstract evaluation of e under $\mathcal{A}$ requires i e-units.

Figure 6 gives the operational rules for the abstract semantics using the $\xrightarrow{\mathcal{A}}_i$ evaluation relation. Beginning with an unevaluated term, the abstract rules are run backwards in a goal-directed fashion towards the

[3] Note that L_0 describes all lists and $L_i \supset L_{i+1}$, $i \geq 0$.

$$\begin{array}{rcl}
b & \in & \textsc{Bool} = \{\mathit{true}, \mathit{false}\} \\
\langle v, l\rangle & \in & \textsc{Cons} = \textsc{DVal} \times \textsc{List} \\
l & \in & \textsc{List} = \{\mathit{nil}\} + \textsc{Cons} \\
[x, e, E] & \in & \textsc{Clos} = \textsc{Var} \times \textsc{Exp} \times \textsc{Env} \\
v & \in & \textsc{DVal} = \textsc{Bool} + \textsc{List} + \textsc{Clos} \\
E & \in & \textsc{Env} = \textsc{Var} \xrightarrow{fin} \textsc{DVal}
\end{array}$$

Figure 3: Dynamic objects of the standard semantics $\mathcal{S}$.

$$\frac{x \mapsto v \in E}{E \vdash x \longrightarrow_0 v} \qquad \textbf{(var)}$$

$$\frac{}{E \vdash (\lambda x.e) \longrightarrow_0 [x, e, E]} \qquad \textbf{(abs)}$$

$$\frac{\begin{array}{r} E \vdash e_1 \longrightarrow_a [x, e, E'] \\ E \vdash e_2 \longrightarrow_b v \\ E' \pm \{x \mapsto v\} \vdash e \longrightarrow_c v' \end{array}}{E \vdash (e_1\ e_2) \longrightarrow_{1+a+b+c} v'} \qquad \textbf{(app)}$$

$$\frac{E \vdash e_1 \longrightarrow_a \mathit{true} \quad E \vdash e_2 \longrightarrow_b v}{E \vdash (\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3) \longrightarrow_{1+a+b} v} \qquad \textbf{(if-true)}$$

$$\frac{E \vdash e_1 \longrightarrow_a \mathit{false} \quad E \vdash e_3 \longrightarrow_b v}{E \vdash (\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3) \longrightarrow_{1+a+b} v} \qquad \textbf{(if-false)}$$

$$\frac{}{E \vdash \mathtt{nil} \longrightarrow_0 \mathit{nil}} \qquad \textbf{(nil)}$$

$$\frac{E \vdash e_1 \longrightarrow_a v \quad E \vdash e_2 \longrightarrow_b l}{E \vdash (\mathtt{cons}\ e_1\ e_2) \longrightarrow_{1+a+b} \langle v, l\rangle} \qquad \textbf{(cons)}$$

$$\frac{E \vdash e \longrightarrow_a \langle v, l\rangle}{E \vdash (\mathrm{hd}\ e) \longrightarrow_{1+a} v} \qquad \textbf{(hd)}$$

$$\frac{E \vdash e \longrightarrow_a \langle v, l\rangle}{E \vdash (\mathtt{tl}\ e) \longrightarrow_{1+a} l} \qquad \textbf{(tl)}$$

$$\frac{E \vdash e \longrightarrow_a \mathit{nil}}{E \vdash (\mathtt{isnull}\ e) \longrightarrow_{1+a} \mathit{true}} \qquad \textbf{(isnull-true)}$$

$$\frac{E \vdash e \longrightarrow_a \langle v, l\rangle}{E \vdash (\mathtt{isnull}\ e) \longrightarrow_{1+a} \mathit{false}} \qquad \textbf{(isnull-false)}$$

Figure 4: Standard semantics $\mathcal{S}$ with time annotations.

$$
\begin{array}{rcl}
b & \in & \textsc{Bool}^{\mathcal{A}} = \{\mathit{true}, \mathit{false}\} \\
L_k & \in & \textsc{List}^{\mathcal{A}} = \{L_0, L_1, \ldots\} \text{ where } L_k \text{ denotes all lists of length} \geq k \\
\left[x, e, E^{\mathcal{A}}\right] & \in & \textsc{Clos}^{\mathcal{A}} = \textsc{Var} \times \textsc{Exp} \times \textsc{Env}^{\mathcal{A}} \\
v & \in & \textsc{DVal}^{\mathcal{A}} = \textsc{Bool}^{\mathcal{A}} + \textsc{List}^{\mathcal{A}} + \textsc{Clos}^{\mathcal{A}} \\
V & \in & \textsc{DValSet}^{\mathcal{A}} = \mathrm{Fin}(\textsc{DVal}^{\mathcal{A}}) + \top^{\mathcal{A}} \\
E^{\mathcal{A}} & \in & \textsc{Env}^{\mathcal{A}} = \textsc{Var} \xrightarrow{fin} \textsc{DValSet}^{\mathcal{A}}
\end{array}
$$

Figure 5: Dynamic objects of the abstract semantics $\mathcal{A}$.

axioms. When more than one rule may apply (*e.g.*, **isnull**$^{\mathcal{A}}$ versus **isnull-false**$^{\mathcal{A}}$), the more specific rule is chosen.

Foremost, note that the **any**$^{\mathcal{A}}$ rule can always be applied. Rule **any**$^{\mathcal{A}}$ evaluates an expression e to any value and incurs no e-unit cost. Therefore, it is a conservative estimate of values and e-units. Note that abstract evaluation can invoke the **any**$^{\mathcal{A}}$ when the premises of no other rule hold. The rule **any**$^{\mathcal{A}}$ is also applied if the depth of the proof exceeds the parallelization cutoff value T (§2). This is further explained in §3.1.3 below.

The **var**$^{\mathcal{A}}$ rule retrieves the mapping of a variable from an environment at zero cost. The **abs**$^{\mathcal{A}}$ rule evaluates a λ-abstraction term to a singleton set containing its closure at zero cost. Again, in practice, costs must be calibrated to a particular machine and implementation.

Abstract evaluation of an application $(e\ e')$ with the **app**$^{\mathcal{A}}$ rule first abstractly evaluates e and e'. When e produces a set F of closures, each $f \in F$ is applied to the value set V that e' produces. The e-unit cost of an application is one e-unit (for the application proper), the e-units required for (abstractly) evaluating e and e', and the minimum of the e-unit costs incurred in applying each $f \in F$ to V. This gives a conservative e-unit count because the cost of the least expensive function reaching the application is used. The set of values produced by **app**$^{\mathcal{A}}$ is the union of the value sets produced by the applications of the closures F. The **app-**$\top^{\mathcal{A}}$ rule handles the case where F is not known.

The conditional rules (**if-true**$^{\mathcal{A}}$, **if-false**$^{\mathcal{A}}$, **if**$^{\mathcal{A}}$) conservatively approximate a conditional's behavior. If the predicate abstractly evaluates to a singleton set containing either *true* or *false*, the respective conditional branch is abstractly evaluated. However, when the predicate's abstract value set is not precisely known (*e.g.*, when it contains both *true* and *false*), both conditional branches are abstractly evaluated and the minimum e-unit cost of these evaluations is incorporated into the conditional's cost—the set of values produced by the conditional is the union of the value sets produced by both conditional branches.

The rules for list objects and the primitive list functions operate as follows. The **nil**$^{\mathcal{A}}$ rule evaluates the syntactic constant `nil` to the identifier L_0 denoting the set of all lists. Abstract evaluation of the constant `nil` incurs no e-unit cost under this cost model.

A list's size (length) increases when an element is *consed* onto it. List creation with the special cons form (**cons**$^{\mathcal{A}}$ rule)—when the tail of the new list is in the set L_i; *i.e.*, it is a list of at least length i)—produces the set of lists of at least length $i+1$, L_{i+1}. The abstract e-unit cost for this operation is one plus the cost of evaluating the arguments to cons. The **cons-**$\top^{\mathcal{A}}$ rule handles the case where all information about the list being *consed* onto has been lost.

Selecting the head (**hd**$^{\mathcal{A}}$ rule) of any object returns any value ($\top^{\mathcal{A}}$) since a list's contents (its elements) are not maintained in the abstract semantics. Selecting the tail (**tl**$^{\mathcal{A}}$ rule) of a list of at least length i returns L_{i-1}, the set of lists of at least length $i-1$, since the list returned by the tail selector is always one less than the length of its argument list. The **tl-**$\top^{\mathcal{A}}$ rule handles application of `tl` to an unknown list.

Testing for the empty list with `isnull` produces the set $\{\mathit{false}\}$ when `isnull`'s argument is a list of at least length ≥ 1 (**isnull-false**$^{\mathcal{A}}$ rule). Otherwise, this test conservatively returns $\{\mathit{true}, \mathit{false}\}$ under abstract evaluation (**isnull**$^{\mathcal{A}}$ rule).

3.1.3 Termination

Abstract evaluation as described may not terminate. Conditional terms, for example, abstractly evaluate both arms. This termination problem is solved by bounding the number of abstract evaluation steps. Evaluation of an execution path under $\mathcal{A}$ terminates (along that path) when the accumulated e-units exceed the overhead threshold T (§2). In other words, when viewed as a deductive proof, the proof tree of an expression's abstract evaluation never exceeds a depth of T unit-cost deductions; *i.e.*, the **any**$^{\mathcal{A}}$ rule is applied upon reaching this bound. Halting abstract evaluation in this manner avoids the non-termination issue since we only evaluate for a bounded T e-units along any execution path and return the cost of the least-cost path.

3.1.4 Program Restructuring

A compiler can use dynamic granularity estimation to restructure the program as follows. The compiler wraps a conditional around every application expression, `(f l)`, that applies function `f` to a list `l`. The conditional's branches respectively contain code for the sequential and parallel evaluation of the application expression (see, for example, Figure 2). The predicate of the compiler-supplied conditional examines the length of `l` (available at run time) and compares it to a compiler-deduced cutoff value (described below). When `l`'s length is at least equal to this cutoff, the conditional selects parallel evaluation for `(f l)`.

$$\frac{}{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_{0} \top^{\mathcal{A}}} \qquad (\mathbf{any}^{\mathcal{A}})$$

$$\frac{x \mapsto V \in E^{\mathcal{A}}}{E^{\mathcal{A}} \vdash x \xrightarrow{\mathcal{A}}_{0} V} \qquad (\mathbf{var}^{\mathcal{A}})$$

$$\frac{}{E^{\mathcal{A}} \vdash (\lambda x.e) \xrightarrow{\mathcal{A}}_{0} \left\{ \left[x, e, E^{\mathcal{A}} \right] \right\}} \qquad (\mathbf{abs}^{\mathcal{A}})$$

$$\frac{\begin{array}{rcl} E^{\mathcal{A}} & \vdash & e \xrightarrow{\mathcal{A}}_{a} \left\{ \left[x_1, e_1, E_1^{\mathcal{A}} \right], \ldots, \left[x_n, e_n, E_n^{\mathcal{A}} \right] \right\} \\ E^{\mathcal{A}} & \vdash & e' \xrightarrow{\mathcal{A}}_{b} V \\ E_i^{\mathcal{A}} \pm \{x_i \mapsto V\} & \vdash & e_i \xrightarrow{\mathcal{A}}_{c_i} V_i, \ 1 \leq i \leq n \end{array}}{E^{\mathcal{A}} \vdash (e \ e') \xrightarrow{\mathcal{A}}_{(1+a+b+\min(c_1,\ldots,c_n))} \bigsqcup_{i=1}^{n} V_i} \qquad (\mathbf{app}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_{a} \top^{\mathcal{A}} \quad E^{\mathcal{A}} \vdash e' \xrightarrow{\mathcal{A}}_{b} V}{E^{\mathcal{A}} \vdash (e \ e') \xrightarrow{\mathcal{A}}_{1+a+b} \top^{\mathcal{A}}} \qquad (\mathbf{app}\text{-}\top^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_{a} \{\mathit{true}\} \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_{b} V}{E^{\mathcal{A}} \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \xrightarrow{\mathcal{A}}_{1+a+b} V} \qquad (\mathbf{if\text{-}true}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_{a} \{\mathit{false}\} \quad E^{\mathcal{A}} \vdash e_3 \xrightarrow{\mathcal{A}}_{b} V}{E^{\mathcal{A}} \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \xrightarrow{\mathcal{A}}_{1+a+b} V} \qquad (\mathbf{if\text{-}false}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_{a} V_1 \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_{b} V_2 \quad E^{\mathcal{A}} \vdash e_3 \xrightarrow{\mathcal{A}}_{c} V_3}{E^{\mathcal{A}} \vdash (\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \xrightarrow{\mathcal{A}}_{1+a+\min(b,c)} V_2 \sqcup V_3} \qquad (\mathbf{if}^{\mathcal{A}})$$

$$\frac{}{E^{\mathcal{A}} \vdash \text{nil} \xrightarrow{\mathcal{A}}_{0} L_0} \qquad (\mathbf{nil}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_{a} V \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_{b} L_i}{E^{\mathcal{A}} \vdash (\text{cons } e_1 \ e_2) \xrightarrow{\mathcal{A}}_{1+a+b} L_{i+1}} \qquad (\mathbf{cons}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e_1 \xrightarrow{\mathcal{A}}_{a} V \quad E^{\mathcal{A}} \vdash e_2 \xrightarrow{\mathcal{A}}_{b} \top^{\mathcal{A}}}{E^{\mathcal{A}} \vdash (\text{cons } e_1 \ e_2) \xrightarrow{\mathcal{A}}_{1+a+b} L_1} \qquad (\mathbf{cons}\text{-}\top^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_{a} V}{E^{\mathcal{A}} \vdash (\text{hd } e) \xrightarrow{\mathcal{A}}_{1+a} \top^{\mathcal{A}}} \qquad (\mathbf{hd}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_{a} L_i}{E^{\mathcal{A}} \vdash (\text{tl } e) \xrightarrow{\mathcal{A}}_{1+a} L_{\max(0,i-1)}} \qquad (\mathbf{tl}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_{a} \top^{\mathcal{A}}}{E^{\mathcal{A}} \vdash (\text{tl } e) \xrightarrow{\mathcal{A}}_{1+a} L_0} \qquad (\mathbf{tl}\text{-}\top^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_{a} L_0}{E^{\mathcal{A}} \vdash (\text{isnull } e) \xrightarrow{\mathcal{A}}_{1+a} \{\mathit{true}, \mathit{false}\}} \qquad (\mathbf{isnull}^{\mathcal{A}})$$

$$\frac{E^{\mathcal{A}} \vdash e \xrightarrow{\mathcal{A}}_{a} L_i \quad i > 0}{E^{\mathcal{A}} \vdash (\text{isnull } e) \xrightarrow{\mathcal{A}}_{1+a} \{\mathit{false}\}} \qquad (\mathbf{isnull\text{-}false}^{\mathcal{A}})$$

Figure 6: Abstract semantics $\mathcal{A}$ with time annotations.

$$\dfrac{\dfrac{\dfrac{}{E \vdash \mathtt{l} \xrightarrow{\mathcal{A}}_0 L_1}\,\textbf{var}^{\mathcal{A}}}{E \vdash (\mathtt{isnull\ l}) \xrightarrow{\mathcal{A}}_1 \{\mathit{false}\}}\,\textbf{isnull-false}^{\mathcal{A}} \quad \dfrac{\dfrac{}{E \vdash \mathtt{x} \xrightarrow{\mathcal{A}}_0 \top^{\mathcal{A}}}\,\textbf{any}^{\mathcal{A}} \quad \dfrac{\dfrac{}{E \vdash \mathtt{l} \xrightarrow{\mathcal{A}}_0 L_1}\,\textbf{var}^{\mathcal{A}}}{E \vdash (\mathtt{tl\ l}) \xrightarrow{\mathcal{A}}_1 L_0}\,\textbf{tl}^{\mathcal{A}}}{E \vdash (\mathtt{cons\ x\ (tl\ l)}) \xrightarrow{\mathcal{A}}_2 L_1}\,\textbf{cons}^{\mathcal{A}}}{E \vdash (\mathtt{if\ (isnull\ l)\ then\ nil\ else\ (cons\ x\ (tl\ l))}) \xrightarrow{\mathcal{A}}_4 L_1}\,\textbf{if}^{\mathcal{A}}$$

Figure 7: Example operation of **dge**'s static component. E maps identifier l to all lists of length ≥ 1; *i.e.*, $E \equiv \{\mathrm{l} \mapsto L_1\}$.

The compiler deduces the cutoff value using abstract evaluation in the following manner. Suppose that **dge**'s dynamic component (§3.2) precisely keeps the lengths of all lists of length $< n$, and that all lists with lengths $\geq n$ are approximated as such. The compiler abstractly evaluates $(\mathtt{f}\ L_i)$ for $0 \leq i < n$. When $(\mathtt{f}\ L_i) \xrightarrow{\mathcal{A}}_x V$, it notes the least i such that the cost x of this application is always greater than the overhead threshold T. This least i, if it exists, represents a length cutoff for `l` at which the creation of a parallel thread for `(f l)` is always beneficial. The value of this least i is the cutoff value in the conditional guarding the application.

In general, the compiler can use the abstract evaluation semantics to determine a cost threshold for *any* expression e, not just for the application of functions to lists. To do so, it must first identify all lists in e; it then abstractly evaluates e for all list-length combinations and notes the lengths at which parallel evaluation of e is viable. This list-length information is then used to construct a predicate to select sequential or parallel evaluation for e.

Section 4 provides a concrete example of the abstract evaluation a compiler must perform to use dynamic granularity estimation.

3.2 Dynamic Component

At run-time, **dge**'s dynamic component maintains an approximation to the length of a list l along with l's physical representation. We assume an implementation that represents lists with *cons cells* in a heap. A fixed field of b bits encodes length information. This gives lists of length $< 2^b - 1$ an exact length (in the length field) at run time. Longer lists of length $\geq 2^b - 1$ have approximate lengths denoted by ∞. When a new list is formed with the list constructor, as in $l \equiv (\text{cons } x\ l')$, the length field on l is set to $\overline{l'} + 1$ if $\overline{l'}$ is not ∞. Otherwise, it is set to ∞.

An implementation of **dge**'s dynamic component can store the b bits of length information either:

1. in a cons cell, or

2. in the pointers to a cons cell

Storing the approximation within the cell requires an additional memory access when forming a new cell since the length field pointed to by the new cell's tail pointer must be fetched. If the cons-cell representation does not contain b unused bits, additional storage must also be allocated in the cell under the first scheme. The second approach requires the pointer representation to contain b unused bits, but avoids an additional memory fetch since construction of a new cons cell always requires the pointer to the list that becomes the new cell's tail field. The first approach is significantly simpler to implement because it only requires modification to the portion of the compiler that generates the code for cons-cell creation (§7). The second approach requires modifications to the implementation's run-time system (*e.g.*, the garbage collector), the generation of special pointer dereferencing code, and (potentially) a revision of the memory layout.

The final concern in the design of the dynamic component is, how many bits, b, to allocate for the length field. A value for b is best selected by consulting the empirical results of applying **dge**'s static analysis (§3.1) to actual programs because, for a typical application $(f\ l)$, where $|(f\ l)|$ depends on the length of l, it is likely that a threshold value for $\bar{l}$ exists at which parallel evaluation of $(f\ l)$ is fruitful. The number of bits b should be large enough to delineate this threshold for most cases.

4 Examples

Here we illustrate the operation of dynamic granularity estimation's static component (abstract evaluation) and show how the compiler can use the information thus obtained, along with run-time list lengths, to dynamically schedule concurrent expressions only when beneficial.

Figure 7 depicts the static deductions that **dge** performs for the expression:

```
e ≡ if (isnull l) then nil
    else (cons x (tl l))
```

The compiler, upon encountering e in a program, can use **dge** to determine e's cost given the length of the list bound to identifier `l`. The figure abstractly evaluates e in the environment $\{\mathrm{l} \mapsto L_1\}$ (*i.e.*, in an environment where `l` is bound to the set of lists of length ≥ 1). Abstract evaluation of e in this environment indicates that e's evaluation produces a list in L_1 and requires at least four e-units (*i.e.*, $\{\mathrm{l} \mapsto L_1\} \vdash e \xrightarrow{\mathcal{A}}_4 L_1$). Abstract evaluation of e in the environment $\{\mathrm{l} \mapsto L_0\}$ produces a list in L_0 and requires two e-units (using the $\textbf{if}^{\mathcal{A}}$, $\textbf{isnull}^{\mathcal{A}}$, and $\textbf{nil}^{\mathcal{A}}$ rules).

As an example of how a compiler combines information from **dge**'s static and dynamic components, consider the function `f`:

```
fun f l = if (isnull l) then nil
          else f (tl l)
```

Abstract evaluation at compile time determines that $(\mathtt{f}\ L_0)$ requires three e-units, $(\mathtt{f}\ L_1)$ requires seven e-units, and $(\mathtt{f}\ L_2)$ requires eleven e-units. In general, abstract (and standard) evaluation of $(\mathtt{f}\ L_n)$ requires $3+4n$ e-units. However, a compiler need only abstractly evaluate (f L_i) for $0 \leq i < 2^b - 1$, where b is the number of bits of list-length information maintained by dge's dynamic component (§3.2), since this encompasses the size information available at run time. The compiler then selects the least i such that |(f L_i)| $> T$ where T is the implementation-specific e-unit threshold (§2). Assuming the concrete values $b = 2$ and $T = 10$ in this example, a compiler using dge can statically deduce that a concurrent thread for (f l) is beneficial when l's length equals or exceeds two.

As a final example, dynamic granularity estimation statically determines that the time complexity of qs (Figure 2) depends on its list parameter. In particular, it detects that split always traverses the entire tail of this parameter. Therefore, the qs function's recursive applications—as well as external applications of qs in other parts of the program—warrant concurrent threads when qs's list parameter is sufficiently[4] large.

5 Extensions

This section describes possible extensions to dynamic granularity estimation that admit general dynamic data structures and mutable data.

5.1 Other Data Structures

In addition to lists, dge can handle general recursive structures (*e.g.*, trees) by defining the size of such a structure to be the sum of the sizes of its substructures. Physical representation of a structure's node then contains the sum of the sizes of the structures pointed to by the node. A node for a binary tree, for example, would carry the sum of the sizes of its left and right subtrees. A static analysis, similar to the analysis presented here for lists, can determine the data sizes for which an expression e's concurrent evaluation is beneficial. However, upon deconstruction of a dynamic node of size n, the analysis must now consider all possible combinations for the substructure's sizes. For example, deconstructing a binary tree of size n with subtrees *left* and *right* requires abstract evaluation with all (n) size assignments such that $|\mathit{left}| + |\mathit{right}| = n - 1$. Enumerating and abstractly evaluating these combinations increases the static analysis' complexity. It is, however, plausible that static examination of all small structures is practical and suffices to delineate a viable size threshold for making thread-creation decisions.

Run-time examination of the size of an *array* can be used to dynamically determine the granularities of expressions in array-based languages (*e.g.*, Fortran and C). An array descriptor (see, for example, [9]) can be used to dynamically convey an array's size and bounds.

Static analysis can then determine, for a program expression e manipulating array a, the sizes of a for which concurrent evaluation of e is beneficial.

5.2 Mutable Dynamic Data

In languages with imperative assignment to *mutable* dynamic data (*e.g.*, ML), it is potentially expensive to dynamically maintain conservative size approximations for these data. This is because a mutable datum's size may decrease and, as with immutable data, mutable data are often shared. To maintain conservative approximations, it may therefore be necessary to propagate—upon assignment into a dynamic structure—a new size to many structures. Identification of structures that share a datum d is, however, difficult because d has no information about the pointers to it. A possible approach to extending dge to mutable data is to *not* propagate reductions in a mutable datum's size. Instead, its size estimate can be reconstituted periodically. Such size reconstitution can occur in the language implementation's garbage collector.[5] Since this approach permits approximations that may overestimate a datum's size, it may—in some cases—select expressions for concurrent evaluation that do not contain enough computation to compensate for scheduling overheads. However, if a large percentage of the dynamic scheduling decisions are correct, dynamic granularity estimation in the presence of modifications to dynamic structures may be viable.

6 Implementation

The dynamic component of dynamic granularity estimation has been implemented in the Standard ML of New Jersey 0.73 optimizing compiler [2]. The MP queue-based multiprocessing platform [22, 4] provides thread creation, synchronization, and management primitives. The *sml2c* code generator [28] outputs C code for execution on a 20-processor shared-memory Sequent Symmetry.

The compiler and run-time system were modified to incorporate one machine word (32 bits) of length information into the standard (three-word) representation of every cons cell (*cf.* §3.2). The compiler's front end was modified to distinguish cons cells from all other types of dynamic objects. This modification identifies cons cells as such for the compiler's back end. The code generator was modified to produce code that computes list lengths upon cons-cell formation. Since a list's length is represented by a full machine word, code for approximating list lengths is unnecessary and is not generated. We introduced high-level functions to provide access to a list's length information. This allows integer lengths to be manipulated as ML values and to be compared against the overhead-threshold values (determined empirically, §2). Low-level primitives, *i.e.* abstract machine instructions, would provide even better performance.

The static component for dge has not been implemented. Abstract evaluation was performed manually.

7 Results

Figure 8 gives the results of dynamic granularity estimation applied to a quicksort (qs, Figure 2) sorting a list of 10000 random integers. The recursive applications of qs for sorting sublists were performed in parallel on 8 processors.[6] The graph plots list-length cutoffs versus

[4] The length of qs's parameter, at which parallel evaluation of an application of qs is beneficial, depends on the machine-dependent threshold T.

[5] A copying garbage collector (*e.g.*, [3, 8]) traverses a data structure in its entirety—it is a simple matter for such a collector to recompute structure sizes.

[6] The graph's standard parallel execution time is a speedup of 3.8 (on 8 processors) over standard sequential execution.

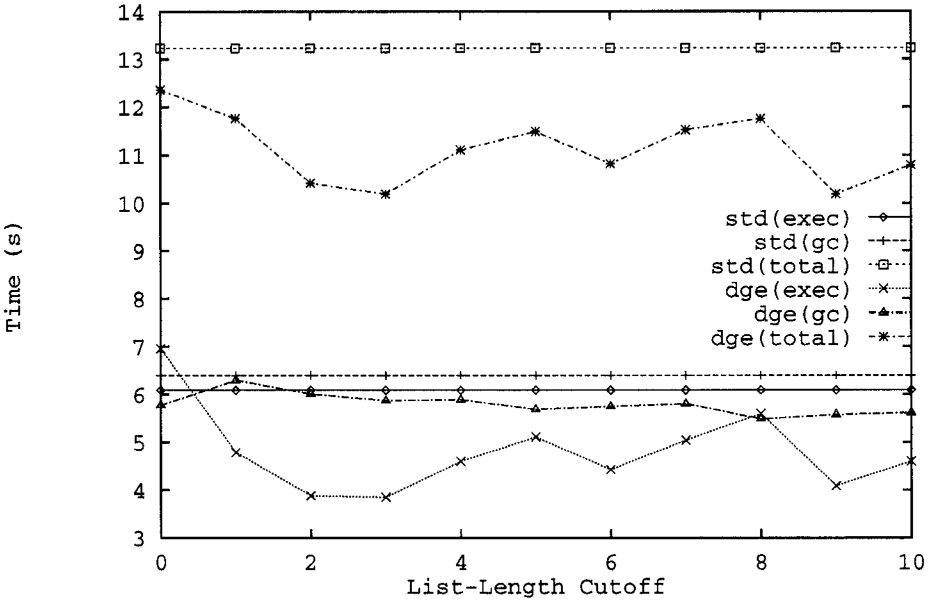

Figure 8: Effect of varying the list-length cutoff threshold in parallel evaluation with 8 processors of quicksort (Figure 2).

execution time. Here, we examine the effect of varying qs's list-length cutoff value on the program's execution time. Parameters of a specific language implementation and machine architecture would enable **dge**'s static component to automatically select a concrete cutoff.

Execution, garbage collection, and total times are given for qs with and without **dge**. The graph's top two curves are the total time required with dynamic granularity estimation (dge) and with standard parallel evaluation (std) respectively. The x-axis is the cutoff values at which threads are retained for sequential evaluation. For the **dge** times, a length cutoff i indicates that the arguments to append in qs evaluate in parallel only when the lengths of the sublists bound to l and g both equal or exceed i. The (std) times are for an ML implementation without the modifications and associated overhead for maintaining list-lengths at run time. The graph's lower curves break the total time into execution (exec) and garbage collection (gc) times. Time spent in the operating system are included in the total times.

Dynamic granularity estimation improves qs's performance at all cutoff values i, $0 \leq i \leq 10$. If thread creation is throttled when sublists are of length < 3, **dge** reduces the total time to execute the program by $\approx$ 23%. Figure 8 also reveals that garbage collection times slightly decrease as the cutoff length increases—fewer threads require fewer memory resources.

Two peculiarities in the timings of Figure 8 require further explanation. First, the non-monotonicity of the execution times arises because of a secondary effect: As the machine fills with threads, it becomes advantageous not to create new threads—even if these threads contain large amounts of computation relative to scheduling costs—since the machine is fully utilized. The input data to qs and the length cutoff (indirectly) influence the machine's load and cause this behavior. The second peculiarity is that the performance of **dge** at a cutoff of zero is better than that of the standard implementation. This is so even though both versions create the same threads and the run-time system for **dge** incurs overhead; it allocates more data and performs more computation in maintaining list lengths than standard parallel evaluation. This occurs because the larger cons cells (four machine words versus three) of the **dge** run time improve processor data-cache performance.[7]

8 Related Work

Most similar to our work is that of Debray, Lin, and Hermenegildo [6] in the context of parallel logic languages. They solve recurrence equations at compile time to obtain upper bounds on execution times. For recursive functions dependent on input sizes, their technique traverses function inputs at run time to compute sizes—parallelization dynamically hinges on these sizes. Our **dge** technique uses lower-bound cost estimates; we lose parallelism in return for parallelism guaranteed to be beneficial whereas the technique of Debray *et al.* may sometimes create small inexpensive threads (with relatively large scheduling overheads) in return for more parallelism. A system that computes *both* upper and lower bounds may provide even better information for dynamic expression scheduling.

Other related work addresses granularity estimation performed entirely at compile time. Aside from sim-

[7] This was verified by experiment. Setting cons-cell sizes to four machine words, improves the performance of some programs. Note that this phenomenon is, however, highly machine and implementation dependent.

ple heuristics [12], work on static granularity estimation falls into one of two categories: load-balancing strategies that continually monitor the number of active threads in the machine to determine when it saturates, and systems that statically derive an algorithm's time complexity, if possible.

In Halstead's Multilisp [13, 14], the program ceases to create new parallel threads when the machine saturates with threads. When this occurs, processors evaluate the available threads to completion. Idle processors steal threads from busy processors in this load-based inlining scheme. Load-based inlining, in the presence of Multilisp's futures, poses deadlock problems, but these can be avoided by Mohr *et al.*'s lazy task creation technique [21, 20]. Lazy task creation efficiently extracts computation from inlined threads when no runnable threads exist. Although lazy task creation increases the granularity of programs by coalescing threads, unlike **dge**, it does not prevent the production of fine-grain threads that are detrimental to the program's quick evaluation. WorkCrews [30] is a thread management package that performs lazy task creation, but requires programmer knowledge of the mechanism. Qlisp [10] provides primitives for performing load-based thread creation as well as automatic load-based inlining [23].

Dynamic granularity estimation is a *load-insensitive* technique that only creates parallel threads that are guaranteed to meet or exceed some granularity criterion. Therefore, **dge** is orthogonal to—and complements—existing load-based inlining and task creation methods.

Harrison's parallel Lisp system, PARCEL [15], employs a non-standard list representation that dynamically maintains information about a list's length. PARCEL uses length information to implement lists contiguously in memory, but not for making parallelization or load-balancing decisions.

Static time-complexity analysis has been studied extensively; static algorithm and program analyzers have been built. Since the general problem of deducing a program's complexity is undecidable, these systems cannot always deduce a program's complexity. In many cases, however, the analyzers do correctly deduce the complexity of a program. METRIC [32] transforms Lisp programs into a set of mutually recursive equations and then seeks their solution to yield the program's complexity. Le Métayer's ACE complexity evaluator [19] matches list-based functional programs against a predefined library of function definitions to map programs to their time complexities. Sands extended this approach to higher-order lazy languages [27].

Dornic, Jouvelot, and Gifford [7] describe a practical *time system* that statically infers a function's complexity from its local definition; *i.e.*, their analysis does not require interprocedural information. Reistad and Gifford [25] recently extended this system to admit static programmer annotation of upper bounds on data structure sizes. Statically, their time system propagates such upper bounds from a datum's point of creation to its subsequent uses. Static time systems are, however, overly imprecise since they determine the costs of recursive functions using only a programmer-supplied upper bound of data structure sizes, or they err conservatively and always assume that application of a recursive function is expensive. In contrast to dynamic granularity estimation, static time-complexity analyses cannot accurately predict an expression's cost when dynamic data sizes are not known at compile time.

Dynamic granularity estimation's static analysis is a form of abstract interpretation [5, 1, 18]. It differs from conventional abstract interpretation in two respects: it assumes the availability of dynamic information, and it does not abstract to finite domains—instead, the threshold that governs thread creation is used to terminate **dge**'s analysis. Wadler addresses the difficulties of static time analysis in (lazy) functional languages [31].

We have previously used run-time information to dynamically discover parallelism in imperative higher-order programs that build and modify dynamic data structures [17, 16].

9 Conclusion

Dynamic granularity estimation (**dge**) is a hybrid static-dynamic technique that assists the automatic parallelization of functional programs—it examines the run-time sizes of data structures and only creates parallel threads that always contain enough computation to offset their scheduling overheads. Hybrid (compile/run time) techniques like **dge** are necessary for effective parallelization since static analyses performed entirely at compile time are inherently conservative. An implementation of **dge** for lists suggests that run-time techniques are a powerful means for selecting threads suitable for parallel evaluation. We view the application of **dge** to languages with general dynamic data structures and arrays as a promising line for further investigation.

Acknowledgement

This work was supported in part by the National Science Foundation under grant CCR-9101035 and by the Wisconsin Alumni Research Foundation. L. Huelsbergen was supported by an ARPA fellowship in parallel processing; thanks to John Williams and IBM Almaden for hosting the internship associated with this fellowship.

References

[1] S. Abramsky and C. L. Hankin, editors. *Abstract Interpretation of Declarative Languages*. Ellis Horwood Ltd., Chichester, West Sussex, England, 1987.

[2] A. W. Appel and D. B. MacQueen. A Standard ML compiler. *Functional Programming Languages and Computer Architecture*, 274:301–324, 1987.

[3] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[4] E. C. Cooper and J. G. Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, School of Computer Science, Carnegie Mellon University, December 1990.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of

programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252. Association for Computing Machinery, 1977.

[6] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task granularity analysis in logic programs. In *Conference on Programming Language Design and Implementation*, pages 174–188, June 1990.

[7] V. Dornic, P. Jouvelot, and D. K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems*, 1(1):33–45, March 1992.

[8] R. R. Fenichel and J. C. Yochelson. A Lisp garbage-collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[9] C. N. Fischer. *Crafting a Compiler.* Benjamin-Cummings, 1988.

[10] R. P. Gabriel and J. McCarthy. Queue-based multiprocessing Lisp. In *Lisp and Functional Programming*, pages 25–44. Association for Computing Machinery, August 1984.

[11] R. Goldman and R. P. Gabriel. Qlisp: Experience and new directions. In *Proceedings of ACM/SIGPLAN PPEALS 1988 (Parallel Programming: Experience with Applications, Languages and Systems)*, pages 111-123, July 1988.

[12] S. L. Gray. Using futures to exploit parallelism in Lisp. Master's thesis, MIT, February 1986.

[13] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[14] R. H. Halstead, Jr. An assessment of Multilisp: Lessons from experience. *International Journal of Parallel Programming*, 15(6):459–501, 1986.

[15] W. L. Harrison and D. A. Padua. PARCEL: Project for the automatic restructuring and concurrent evaluation of lisp. In *International Conference on Supercomputing*, pages 527–538, July 1988.

[16] L. Huelsbergen. *Dynamic Language Parallelization.* PhD thesis, University of Wisconsin–Madison, August 1993.

[17] L. Huelsbergen and J. R. Larus. Dynamic program parallelization. In *Lisp and Functional Programming*, pages 311–323. Association for Computing Machinery, June 1992.

[18] L. S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, 1991.

[19] D. Le Métayer. ACE: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.

[20] E. Mohr. *Dynamic Partitioning of Parallel Lisp Programs.* PhD thesis, Yale University, August 1991.

[21] E. Mohr, D. Kranz, and R. H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Lisp and Functional Programming*, pages 185–197. Association for Computing Machinery, June 1990.

[22] J. G. Morrisett and A. Tolmach. Procs and locks: A portable multiprocessing platform for Standard ML of New Jersey. In *Principles and Practice of Parallel Programming*, pages 198–207. Association for Computing Machinery, May 1993.

[23] J. D. Pehoushek and J. S. Weening. Low-cost process creation and dynamic partioning in Qlisp. In *US/Japan Workshop on Parallel Lisp*, pages 183–199. Lecture Notes in Computer Science, June 1989.

[24] G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[25] B. Reistad and D. Gifford. Static dependent costs for estimating execution time. In *Lisp and Functional Programming.* Association for Computing Machinery, June 1994.

[26] J. C. Reynolds. GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM*, 13(5):308–319, 1970.

[27] D. Sands. Complexity analysis for a lazy higher-order language. In *ESOP*, pages 361–376. Lecture Notes in Computer Science, May 1990.

[28] D. Tarditi, A. Acharya, and P. Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 1990.

[29] M. Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, Department of Computer Science, May 1988.

[30] M. T. Vandevoorde and E. S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, 1988.

[31] P. L. Wadler. Strictness analysis aids time analysis. In *Symposium on Principles of Programming Languages*, pages 119–132. Association for Computing Machinery, January 1988.

[32] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, September 1975.