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

Lorenz Huesbergen
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, 26].

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. 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 $e \equiv \langle f, l \rangle$ 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 (1 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
Figure 1: The impact of overhead. Time starts at $t_0$. The concurrent evaluation of $e_1$ and $e_2$, with overhead ($O = O_{fork} + O_{join}$) 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.

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 $l$ and $g$) 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 (append|| evaluates its arguments in parallel). The compiler also deduces the cutoff value.
the cost of applying $e$ 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, $dg e$ approximates the lengths of all lists; the length information of the lists bound to the identifiers 1 and g in the $dq$ function is available for making the final parallelization decision.

Dynamic techniques, like $dg e$, 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 $dq$ example. When a statically-unknown list reaches $dq$, the sublist partition that $dq$'s auxiliary split function creates is also unknown. Therefore, the costs of the recursive applications of $dq$ that sort the sublists cannot be known at compile time. In the absence of precise static information about $dq$'s list parameter, it is not possible to statically decide when concurrent evaluation of $dq$'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 $dq$ 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 $dg e$ 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 $dg e$'s static ([3.1] and dynamic ([3.2]) components, illustrate $dg e$'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 $\lambda$-calculus, a functional\footnote{Restriction to a functional language allows efficient implementation of $dg e$'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.}, call-by-value, higher-order language [24, 26]. The ground terms of $\lambda$ are variables and constants:

$$
\begin{align*}
  x & \in \text{VAR} \\
  b & \in \text{CONST} = \{\text{nil}, \text{true}, \text{false}\}
\end{align*}
$$

The terms of $\lambda$ are expressions ($e \in \text{EXP}$) and values ($v \in \text{VAL} \subseteq \text{EXP}$):

$$
\begin{align*}
  \lambda & \in \text{VAR} \\
  e \in \text{EXP} & = \{x, e + e, e - e, e * e, e / e, \text{if then else}, \text{cons} e e, \text{hd} e, \text{tl} e, \text{isnull} e\}
\end{align*}
$$

We assume that $\lambda$ 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 $\lambda$ therefore contains cons, hd, tl, and isnull directly. Section 5 describes possible extensions of $dg e$ to general recursive data types 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.\footnote{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.} 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 $\lambda$, we assume that the evaluation of variables, constants, and $\lambda$-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 $\mathcal{L}$. When $i$ is a natural number, $\mathcal{L}^i$ 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 $A \setminus B$ is their difference. The empty set is denoted by $\emptyset$, and $\text{Fin}(A)$ denotes the set of finite subsets of $A$. If $f$ is a map, then the domain and range of $f$ are $\text{Dom}(f)$ and $\text{Rng}(f)$. 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 $A^{\text{fin}} \rightarrow B$ where any $f \in A^{\text{fin}} \rightarrow B$ can be written as $\{a_1 \rightarrow b_1, \ldots, a_n \rightarrow b_n\}$. The empty map is written $\emptyset$. If $f$ and $g$ are maps, then $f + g$ is the map with $f$ modified by $g$ and has the domain $\text{Dom}(f) \cup \text{Dom}(g)$ and the values:

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

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

$$
\frac{P_1 \cdots P_n}{C}
$$
where $n > 0$. Successful inference of the premises, $P_k$, 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 $e \equiv (f \, l)$, where $l$ is the list length. The compiler inserts a check if $l$'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 the dynamic component of the language and then the abstract semantics. 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 is used to identify functions whose list parameters control their complexity; an abstract semantics that interprets a list $l$ as its length $l$ 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 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., avoids scheduling overheads).

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 is used to identify functions whose list parameters control their complexity; an abstract semantics that interprets a list $l$ as its length $l$ 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 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., avoids scheduling overheads).

3.1.1 Standard Semantics $S$

The dynamic objects of the standard semantics $S$ 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 $\lambda$: The constant $\text{nil}$ is the empty list and a cons pair $(v, l)$ 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 $E \vdash e \rightarrow_i v$ (where $E \in \text{ENV}, e \in \text{EXP}, v \in \text{DVAL}$, and $i \in \mathbb{Z}$) indicates that the evaluation of expression $e$ to value $v$ with respect to environment $E$ requires $i$ e-units. For example, the $\text{app}$ 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 $a + b + c$ 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 $\lambda e \cdot e$'s value terms (e.g., variables and $\lambda$-abstractions) requires zero e-units under this relation; a specific implementation would, however, use an e-unit measure.

The rule states that if the evaluation of $e_1$ to $v_1$ requires $a$ e-units, the evaluation of $e_2$ to $v_2$, the evaluation of $v_1$ to $v_2$ requires $c$ e-units, then the evaluation of the application $(e_1 \, e_2)$ requires $a + b + c$ 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 $\lambda e \cdot e$'s value terms (e.g., variables and $\lambda$-abstractions) requires zero e-units under this relation; a specific implementation would, however, use an e-unit measure.

3.1.2 Abstract Semantics $A$

A non-standard (abstract) semantics $A$ 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 $\sigma(V)$:

$$\sigma\{\{v_1, \ldots, v_n\}\} = \bigcup_{i=1}^{n} \sigma\{\{v_i\}\}$$

$$\sigma(\{\text{true}\}) = \{\text{true}\}$$

$$\sigma(\{\text{false}\}) = \{\text{false}\}$$

$$\sigma(L_k) = \{l \mid l \text{ is a list of length } \geq k\}$$

$$\sigma(\{E^A\}) = \{f \mid \forall z \in \text{Dom}(E^A), f(z) \in E^A(z)\}$$

$$\sigma(\{[v, e, E^A]\}) = \{[v, e, E] \mid E \in \sigma(E^A)\}$$

$$\sigma((\top^A)) = \{v \mid v \in \text{DVAL}\}$$

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

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

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

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

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

Figure 6 gives the operational rules for the abstract semantics using the $\rightarrow_i^A$ evaluation relation. Beginning with an unevaluated term, the abstract rules are run backwards in a goal-directed fashion towards the
\[
\begin{align*}
  b & \in \text{BOOL} = \{\text{true}, \text{false}\} \\
  (v, l) & \in \text{CONS} = \text{DVAL} \times \text{LIST} \\
  l & \in \text{LIST} = \{\text{nil}\} + \text{CONS} \\
  [x, e, E] & \in \text{CLOS} = \text{VAR} \times \text{EXP} \times \text{ENV} \\
  v & \in \text{DVAL} = \text{BOOL} + \text{LIST} + \text{CLOS} \\
  E & \in \text{ENV} = \text{VAR} \overset{\text{fin}}{\to} \text{DVAL}.
\end{align*}
\]

**Figure 3:** Dynamic objects of the standard semantics \( \mathcal{S} \).

\[
\begin{align*}
  x & \mapsto v \in E \\
  E \vdash \text{var} x \to_0 v & \quad \text{(var)}
\end{align*}
\]

\[
\begin{align*}
  E \vdash (\lambda x. e) \to [x, e, E] & \quad \text{(abs)} \\
  E \vdash e_1 \to [x, e, E'] & \\
  E \vdash e_2 \to_{\text{v}} v \\
  E' \setminus \{x \mapsto v\} \vdash e \to_{\text{v}} v' \\
  E \vdash (e_1; e_2) \to \text{app} \Gamma_\varepsilon[\text{app} \Gamma_\varepsilon] & \quad \text{(app)} \\
  E \vdash e_1 \to_{\text{true}} E \vdash e_2 \to_{\text{v}} v & \quad \text{(if-true)} \\
  E \vdash (\text{if} e_1 \text{ then } e_2 \text{ else } e_3) \to \text{if-true} \Gamma_\varepsilon[\text{if-true} \Gamma_\varepsilon] & \quad \text{(if-true)} \\
  E \vdash e_1 \to_{\text{false}} E \vdash e_3 \to_{\text{v}} v & \quad \text{(if-true)} \\
  E \vdash (\text{if} e_1 \text{ then } e_2 \text{ else } e_3) \to \text{if-true} \Gamma_\varepsilon[\text{if-true} \Gamma_\varepsilon] & \quad \text{(if-true)} \\
  E \vdash \text{nil} \to_{\text{nil}} \text{nil} & \quad \text{(nil)} \\
  E \vdash e_1 \to_{\text{v}} v & \\
  E \vdash (\text{cons} e_1; e_2) \to_{\text{cons} \Gamma_\varepsilon[\text{cons} \Gamma_\varepsilon]} & \quad \text{(cons)} \\
  E \vdash e \to_{\text{v}} v & \\
  E \vdash (\text{hd} e) \to_{\text{hd} \Gamma_\varepsilon[\text{hd} \Gamma_\varepsilon]} & \quad \text{(hd)} \\
  E \vdash e \to_{\text{v}} v & \\
  E \vdash (\text{tl} e) \to_{\text{tl} \Gamma_\varepsilon[\text{tl} \Gamma_\varepsilon]} & \quad \text{(tl)} \\
  E \vdash e \to_{\text{v}} \text{nil} & \quad \text{(isnull-true)} \\
  E \vdash (\text{isnull} e) \to_{\text{isnull} \Gamma_\varepsilon[\text{isnull} \Gamma_\varepsilon]} & \quad \text{(isnull-true)} \\
  E \vdash e \to_{\text{v}} \text{true} & \quad \text{(isnull-true)} \\
  E \vdash e \to_{\text{v}} \text{false} & \quad \text{(isnull-true)}
\end{align*}
\]

**Figure 4:** Standard semantics \( \mathcal{S} \) with time annotations.
\begin{align*}
b &\in \text{BOOL}^A = \{\text{true}, \text{false}\} \\
L_k &\in \text{LIST}^A = \{L_0, L_1, \ldots\} \text{ where } L_k \text{ denotes all lists of length } \geq k \\
x, e, E^A &\in \text{CLOS}^A = \text{VAR} \times \text{EXP} \times \text{ENV}^A \\
v &\in \text{DVAL}^A = \text{BOOL}^A + \text{LIST}^A + \text{CLOS}^A \\
V &\in \text{DVALSET}^A = \text{FIN}(\text{DVAL}^A) + T^A \\
E^A &\in \text{ENV}^A = \text{VAR} \stackrel{\text{fin}}{\rightarrow} \text{DVALSET}^A
\end{align*}

Figure 5: Dynamic objects of the abstract semantics $A$.

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

Foremost, note that the any$^A$ rule can always be applied. Rule any$^A$ 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$^A$ when the premises of no other rule hold. The rule any$^A$ is also applied if the depth of the proof exceeds the parallelization cutoff value $T$ ([§2]). This is further explained in §3.12 below.

The var$^A$ rule retrieves the mapping of a variable from an environment at zero cost. The abs$^A$ 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$^A$ 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$^A$ is the union of the value sets produced by the applications of the closures $F$. The app-$T^A$ rule handles the case where $F$ is not known.

The conditional rules (if-true$^A$, if-false$^A$, if$^A$) 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$^A$ 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$^A$ 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-$T^A$ rule handles the case where all information about the list being consed onto has been lost.

Selecting the head (hd$^A$ rule) of any object returns any value ($T^A$) since a list’s contents (its elements) are not maintained in the abstract semantics. Selecting the tail (tl$^A$ 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-$T^A$ rule handles application of tl to an unknown list.

Testing for the empty list with isnull produces the set \{false\} when isnull’s argument is a list of at least length $\geq 1$ (isnull-false$^A$ rule). Otherwise, this test conservatively returns \{true, false\} under abstract evaluation (isnull$^A$ 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 $A$ 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$^A$ 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, (if 1), that applies function $f$ to a list 1. 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 1 (available at run time) and compares it to a compiler-deduced cutoff value (described below). When 1’s length is at least equal to this cutoff, the conditional selects parallel evaluation for (if 1).
Figure 6: Abstract semantics $\mathcal{A}$ with time annotations.
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 less than \(n\), and that all lists with lengths \(\geq n\) are approximated as such. The compiler abstractly evaluates \(f \, L_i\) for \(0 \leq i < n\). When \(f \, L_i\) \(\rightarrow^{\alpha} V\), 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 \(f\) at which the creation of a parallel thread for \(f\) 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 \texttt{cons} cells in a heap. A fixed field of \(b\) bits encodes length information. This gives lists of length \(< \, 2^b\, - 1\) an approximate length (in the length field) at run-time. Longer lists of length \(\geq 2^b\, - 1\) have approximate lengths denoted by \(\infty\). When a new list is formed with the list constructor, as in \(l \equiv \text{\texttt{cons} } x \, l'\), the length field on \(l\) is set to \(l' + 1\) if \(l'\) is not \(\infty\). Otherwise, it is set to \(\infty\).

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

1. in a \texttt{cons} cell, or
2. in the pointers to a \texttt{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 \(f\) 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:

\[
e \equiv \text{if (isnull 1) then nil else (cons x (tl 1))}
\]

The compiler, upon encountering \(e\) in a program, can use dge to determine \(e\)'s cost given the length of the list bound to \texttt{identifier} \(l\). The figure abstractly evaluates \(e\) in the environment \(\{l \mapsto \texttt{L}_1\}\) (i.e., in an environment where \(l\) is bound to the set of lists of length \(\geq 1\)). Abstract evaluation of \(e\) in this environment indicates that \(e\)'s evaluation produces a list in \(\texttt{L}_1\) and requires at least four \(\texttt{e-units}\) (i.e., \(\{l \mapsto \texttt{L}_1\} + e \rightarrow^4 \texttt{L}_1\) ). Abstract evaluation of \(e\) in the environment \(\{l \mapsto \texttt{L}_0\}\) produces a list in \(\texttt{L}_0\) and requires two \(\texttt{e-units}\) (using the \(\texttt{if}\), \(\texttt{isnull}\), and \(\texttt{nil}\) rules).

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

\[
\text{fun } f \, 1 = \text{if (isnull 1) then nil else } f \, \text{(tl 1)}
\]

Abstract evaluation at compile time determines that \((f\, \texttt{L}_0)\) requires three \(\texttt{e-units}\), \((f\, \texttt{L}_1)\) requires seven \(\texttt{e-units}\), and \((f\, \texttt{L}_2)\) requires eleven \(\texttt{e-units}\). In general, abstract (and standard) evaluation of \((f\, \texttt{L}_n)\) requires \(3 + 4n\) \texttt{e-units}. However, a compiler need only abstractly
evaluate \((f \cdot L_i)\) for \(0 < i < 2^d - 1\), where \(b\) is the number of bits of list-length information maintained by \(\text{dge}\)'s dynamic component (§3.2), since this encompasses the size information available at run time. The compiler then selects the least \(i\) such that \( \mid (f \cdot L_i) \mid > T \) 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 \(\text{dge}\) can statically deduce that a concurrent thread for \((f 1)\) is beneficial when \(1\)'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, \(\text{dge}\) 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 \(\mid left \mid + \mid right \mid = n - 1\). 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 \(\alpha\), the sizes of \(\alpha\) 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 \(d\)'s, however, difficult because \(d\) has no information about the pointers to it. A possible approach to extending \(\text{dge}\) 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 \(\text{smile}\) code generator [28] outputs C code for execution on a 20-processor shared-memory 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 \(\text{dge}\) 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.
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 1 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 ≤ i ≤ 10. If thread creation is throttled when sublists are of length < 3, dge reduces the total time to execute the program by ≈ 20%. 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.

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-

---

**Figure 8:** Effect of varying the list-length cutoff threshold in parallel evaluation with 8 processors of quicksort (Figure 2).
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 procedural information. Réétdal and Gifford [25] recently extended this system to admit static programmer annotation of upper bounds on data structures 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 [8, 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 high-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-910335 and by the Wisconsin Alumni Research Foundation. L. Huesbergen 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


