Workshop on
Declarative Aspects of Multicore Programming

DAMP 2008

Sponsored by ACM SIGPLAN and Intel Corporation

January 9, 2008
San Francisco, CA, USA
(co-located with POPL 2008)

Foreword

Parallelism is going mainstream. Many chip manufactures are turning to multicore processor designs rather than scalar-oriented frequency increases as a way to get performance in their desktop, enterprise, and mobile processors. This endeavor is not likely to succeed long term if mainstream applications cannot be parallelized to take advantage of tens and eventually hundreds of hardware threads. Multicore architectures will differ in significant ways from their multisocket predecessors. For example, the communication to compute bandwidth ratio is likely to be higher, which will positively impact performance. More generally, multicore architectures introduce several new dimensions of variability in both performance guarantees and architectural contracts, such as the memory model, that may not stabilize for several generations of product.

Programs written in functional or (constraint-)logic programming languages, or even in other languages with a controlled use of side effects, can greatly simplify parallel programming. Such declarative programming allows for a deterministic semantics even when the underlying implementation might be highly non-deterministic. In addition to simplifying programming this can simplify debugging and analyzing correctness.

These are the informal proceedings of DAMP 2008, an Intel-sponsored, one-day workshop, co-located with POPL 2008, and held on January 9, 2008, in San Francisco, CA. DAMP seeks to explore ideas in programming language design that will greatly simplify programming for multicore architectures, and more generally for tightly coupled parallel architectures. The emphasis is on functional and (constraint-)logic programming, but any programming language ideas that aim to raise the level of abstraction are welcome. DAMP seeks to gather together researchers in declarative approaches to parallel programming and to foster cross fertilization across different approaches. Previous DAMPs were held, also co-located with POPL, in Nice, France, in 2007 and in Charleston, SC, USA, in 2006.

Manuel Hermenegildo
DAMP'08 Program Chair

Leaf Petersen and Neal Glew
Intel Corporation, Santa Clara, CA, USA
DAMP'08 General Chairs

Program Committee:

Koen De Bosschere (U. of Gent, Belgium)
Manuel Carro (Tech. U. of Madrid, Spain)
Manuel Chakravarty (U. of N.S. Wales, Australia)
Clemens Grelck (U. of Luebeck, Germany)
Dan Grossman (U. of Washington, USA)
Suresh Jagannathan (Purdue U., USA)
Pedro Lopez-Garcia (Tech. U. of Madrid, Spain)
Lee Naish (Melbourne University, Australia)
Leaf Petersen (Intel Corporation, USA)
Enrico Pontelli (New Mexico State U., USA)
John Reppy (U. of Chicago, USA)
Vitor Santos-Costa (U. of Porto, Portugal).

Program Chair:

Manuel Hermenegildo
Tech. U. Madrid / IMDEA-Software
University of New Mexico.

General Chairs:

Leaf Petersen and Neal Glew
Intel Corporation, Santa Clara, CA, USA.

URL:

`http://www.cliplab.org/Conferences/DAMP08`

Past DAMPs:

`http://glew.org/damp2006`
`http://www.cs.cmu.edu/ damp`

Table of Contents and Program

Intel 64 Architecture Memory Ordering

Bratin Saha

Programming Systems Laboratory
Intel Corporation

Abstract. The advent of multi-core processors has brought parallel programming to the mainstream. Memory ordering plays a fundamental role in writing efficient and correct parallel programs. This talk will discuss the recently released Intel 64 architecture memory ordering. We will discuss the motivation, and the guarantees provided by this ordering. We will also discuss how it relates to language level memory models.

Partial Vectorisation of Haskell Programs

Manuel M. T. Chakravarty[1], Roman Leshchinskiy[1], Simon Peyton Jones[2], and Gabriele Keller[1]

[1] Programming Languages and Systems, School of Computer Science and Engineering, University of New South Wales, {chak,rl,keller}@cse.unsw.edu.au
[2] Microsoft Research Ltd, Cambridge, England, simonpj@microsoft.com

Abstract. Vectorisation for functional programs, also called the flattening transformation, relies on drastically reordering computations and restructuring the representation of data types. As a result, it only applies to the purely functional core of a fully-fledged functional language, such as Haskell or ML. A concrete implementation needs to apply vectorisation selectively and integrate vectorised with unvectorised code. This is challenging, as vectorisation alters the data representation, which must be suitably converted between vectorised and unvectorised code. In this paper, we present an approach to partial vectorisation that selectively vectorises sub-expressions and data types, and also, enables linking vectorised with unvectorised modules.

Keywords: Vectorisation; flattening; program transformation; Haskell

1 Introduction

The idea of implementing *nested data parallelism* [1] in functional programs by a vectorising program transformation is at least as old as Blelloch & Sabot's seminal work [2, 3] on the *flattening transformation.* We have since generalised the basic idea to cover the central features of modern functional languages, such as algebraic data types, parametric polymorphism, and higher-order functions [4–6]. However, apart from a prototype that compiled a subset of Paralation Lisp, the only complete implementation of vectorisation by flattening was, to the best of our knowledge, the experimental NESL system [7]. Due to its experimental nature, NESL was a rather limited functional language; for example, it did not admit user-defined algebraic data types and higher-order functions, had only rudimentary I/O, and did not have a module system or support separate compilation. In fact, NESL was implemented as a whole program compiler that vectorised the entire program.

In our implementation of vectorisation in the Glasgow Haskell Compiler (GHC) as part of the *Data Parallel Haskell project* [8], we cannot follow NESL's approach. We expect that a real application will consist of a computationally-intensive core that must be vectorised, embedded in a larger program that parses its command line, reads configuration files, drives a GUI, outputs Postscript, and so on. None of this surrounding code can or should be vectorised.

Consequently, we need a form of *selective* vectorisation that vectorises as much as possible, but leaves sub-expressions that depend on impure features, or unvectorised external code as is. Moreover, we must integrate vectorised with unvectorised code, which is challenging because vectorisation alters the representation of data structures and functional values; hence, values need to be suitably converted when passed between vectorised and unvectorised code.

This paper makes the following technical contributions:

- We give the first presentation of a *selective vectorisation transformation* that only vectorises sub-expressions that do not rely on impure or otherwise non-vectorisable features and in particular on external, unvectorised data structures and code (Section 3).
- We describe the integration of vectorised and unvectorised modules and the additional information that vectorisation requires to be maintained across separately compiled modules (Section 4).

Before we dive into these technical details, Section 2 summarises the main ideas of vectorisation and introduces our approach by example. We cover related work in Section 5.

2 Vectorisation in a nutshell

The various aspects of vectorising purely functional programs including algebraic data types, parametric polymorphism, and higher-order functions were described in detail in previous work [4–6, 8]. In the following, we summarise the core ideas with a concrete example. Afterwards, we motivate and informally illustrate the main ideas of the present paper by extending our example to include I/O.

2.1 Data Parallel Haskell

Data Parallel Haskell (DPH) introduces a type of *parallel arrays*, denoted `[:e:]` for arrays of type `e`, together with a large number of parallel collective operations. As far as possible, these operations have the same names as Haskell's standard list functions, but with a `P` suffix added—i.e., `mapP`, `filterP`, `unzipP`, and so forth. The language also includes parallel array comprehensions which are similar to list comprehensions but operate on parallel arrays. Details of the language extension and examples are in [9].

The crucial difference between Haskell lists and parallel arrays is that the latter have a parallel evaluation semantics. More precisely, demand for any element of a parallel array results in the evaluation of all elements—in particular, on a parallel machine, we expect the evaluation of these elements to happen in parallel. Figure 1 gives an excerpt from a two-dimensional Barnes-Hut n-body simulator, an example that we chose because it is both computationally intensive and very hard to express using flat data parallelism. The parallelism happens inside `oneStep`, where all particles are processed in a single parallel step and where the main workhorses `buildTree` and `accelerate` (which, in turn, contain

```
type Vector   = (Float, Float)
type Area     = (Vector, Vector)

data MassPnt  = MassPnt { mass :: Float, location :: Vector }
data Particle = Particle { center :: MassPnt, velocity :: Vector }

data Tree = Node MassPnt [:Tree:]    — Rose tree for spatial decomposition

— Perform spatial decomposition and build the quadtree
buildTree :: [:MassPnt:] -> Tree

— Change velocity of each particle in ps according to force affected by masses in tree
accelerate :: Tree -> [:Particle:] -> [:Particle:]

— Move a mass center according to the given velocity
movePnt :: MassPnt -> Vector -> MassPnt

— Move a particle according to its velocity
moveParticle :: Particle -> Particle
moveParticle p = let v = velocity p
                 in Particle (movePnt (center p) v) v

— Compute one step of the n-body simulation
oneStep :: Float -> [:Particle:] -> [:Particle:]
oneStep ps
  = [: moveParticle p | p <- accelerate tree ps :]
  where
    mps   = [:mp | Particle mp v <- ps:]
    tree  = buildTree mps
```

Fig. 1. Excerpt of 2-D Barnes-Hut n-body code

parallel computations) are invoked. The function `buildTree` constructs a quadtree and `accelerate` uses it to compute the acceleration of a set of particles in $O(n \log n)$ work complexity. All this needs to be vectorised for parallel execution. More precisely, we need to ensure that we call the fully vectorised variants of the functions $\texttt{buildTree}_\text{V}$, $\texttt{accelerate}_\text{V}$, and $\texttt{moveParticle}_\text{V}$, to achieve $O(\log^2 n)$ parallel step complexity; details are in [10, 4, 9].

2.2 Full vectorisation and why it may fail

Consider a top-level function definition $f\texttt{::}t\ \texttt{=}\ e$, where t is the (monomorphic) type of f. The *full vectorisation* transformation generates a new variant of f, thus:

$f_\text{V}\ \texttt{::}\ \mathcal{T}[\![t]\!]\ \texttt{=}\ \mathcal{V}[\![e]\!]$ — If e is vectorisable

```
oneStepIO :: [:Particle:] -> IO [:Particle:]
oneStepIO ps
  = do { print qs; return qs }          — Side effecting I/O computation
  where
    mps  = [:mp | Particle mp v <- ps:]  — purely functional code...
    tree = buildTree mps                 — ...that must be...
    qs   = [: moveParticle p             — ... vectorised and...
           | p <- accelerate tree ps:]   — ... run in parallel
```

Fig. 2. Parallel code mixed with I/O

Here, f_{V} is the *fully vectorised* variant of f, whose right-hand side is generated by the *full vectorisation transform* $\mathcal{V}[\![\cdot]\!]$. Full vectorisation returns an expression of a different type to the input, so the type of f_{V} is obtained by vectorising the type t, thus $\mathcal{T}[\![t]\!]$. In general, if $e :: t$ then $\mathcal{V}[\![e]\!] :: \mathcal{T}[\![t]\!]$.

Full vectorisation is the flattening transformation of Blelloch and Sabot, subsequently elaborated by ourselves to handle polymorphism, user-defined algebraic data types, and higher-order functions [4–6]. It is, however, not the subject of this paper, so we will keep details of full vectorisation to a minimum.

In real programs, however, full vectorisation of the entire program may be neither possible, nor even desirable. Haskell[1] supports a significant number of impure features, including monadic I/O and mutable variables, exceptions, thread-based concurrency, and calls to external C code. Code using these impure features resists vectorisation due to such code's dependence on a particular evaluation order.

As an example, consider the code in Figure 2 which extends the original `oneStep` with a call to the I/O function

```
print :: Show a => a -> IO ()
```

which prints values to the standard output. Its purpose here is to output the state of simulated particles after each time step, for example, to drive an animation. We cannot vectorise the entire body of `oneStepIO` *because we do not have a vectorised version of* `print`. In all likelihood, the module `System.IO`, which exports `print`, will not have been compiled with vectorisation in the first place, since vectorising it would be pointless. But even if we tried to vectorise `System.IO`, we would still not get $\texttt{print}_{\mathrm{V}}$ because of this function's dependence on sequential C procedures. In short, the full vectorisation transform $\mathcal{V}[\![e]\!]$ may *fail*. It may fail because it encounters some impure feature in e that prevents vectorisation; or because e mentions some imported function f that was compiled without vectorisation, or for which vectorisation failed. In the latter case, no binding for f_{V} would have been created.

[1] Here we mean the extension of Haskell 98 implemented by GHC, which has many additional features that will be in the next standard. Nevertheless, already Haskell 98 supports a range of I/O operations.

2.3 Selective vectorisation

We cannot vectorise the whole of `oneStep`, but we still want to selectively vectorise *as much of it as possible*, so that the code computing the new particles is evaluated in parallel (the `where` clause in Figure 2). In general, for each top-level binding f`::`t `=` e we apply the *selective vectorisation transform* $\mathcal{S}[\![\cdot]\!]$ to e, thus:

$$f \texttt{ :: } t \texttt{ = } \mathcal{S}[\![e]\!]$$

In contrast to full vectorisation, selective vectorisation keeps the result type the same—if necessary by introducing suitable conversions. This is exactly what happens with `oneStepIO`, for example.

In fact, even if we *are* able to fully vectorise f, we must still retain a binding of name and type f`::`t. After all, f might be exported and used by a module not compiled with vectorisation or used in a context that we cannot vectorise. To keep matters simple and predictable, we therefore generate the binding f`::`t `=` $\mathcal{S}[\![e]\!]$ regardless of whether or not full vectorisation succeeds.

Conversions. The selective vectorisation transform should vectorise the pure, performance-critical part of `oneStepIO` and get $\texttt{qs}_\text{V}$ `::` $\mathcal{T}[\![\texttt{[:Particle:]}]\!]$. This means that although we cannot vectorise `print qs`, we still want to use $\texttt{qs}_\text{V}$ as the argument to `print`. But the types do not match! So we must convert from $\mathcal{T}[\![\texttt{[:Particle:]}]\!]$ to `[:Particle:]` using the (overloaded) function `fromV`. Thus, selective vectorisation of `oneStepIO` should turn `print qs` into `print (fromV` $\texttt{qs}_\text{V}$`)`.

The functions `fromV` and `toV` marshal arguments and results "across the border" between un-vectorised and vectorised code. The selective vectorisation transform generates suitable `fromV` and `toV` functions, based on the types to be marshaled. However, this is neither possible nor desirable for all types—for complicated types it is simply too expensive. Hence, selective vectorisation decides which sub-expressions to vectorise, using both

- the presence or absence of vectorised versions $\texttt{f}_\text{V}$ of the free variables `f` of the expression, and
- the presence or absence of conversion functions `fromV` and `toV` at the required types (i.e., the types of the free variables and result).

We formalise this idea in Section 3.2.

Optimality. In general, there is more than one way to selectively vectorise a given expression. This raise the question of which of multiple translations to choose, and especially, whether one translation is "better" than the others. Unfortunately, these questions are not easy to answer as we have two potentially opposing requirements. On one hand, we want to vectorise as much code as possible—after all, only vectorised code will make good use of parallelism. On the other hand, the use of the conversion functions `fromV` and `toV` can be expensive if large data structures are converted, especially if that happens repeatedly

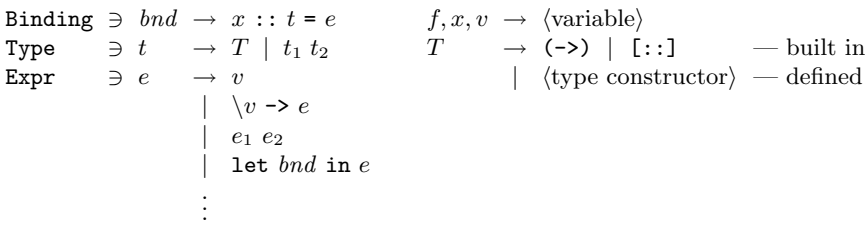


Fig. 3. Fragment of GHC's Core intermediate language

in a recursive function. We leave a detailed analysis of this trade off and the development of a cost model or a heuristic approach to decide on which sub-expression to vectorise for future work. The transformation formalised in the next section simply attempts to vectorise as many sub-expression as possible. However, some guidance by the programmer is possible as the transformation does not assume that conversions are available for all types; i.e., by not having conversions for types inhabited by values that may be costly to convert (e.g., complex tree structures), programmers can indirectly guide selective vectorisation.

3 Selective vectorisation precisely

Our description of selective vectorisation has been entirely informal thus far. In the rest of the paper we give a more precise description. The presentation is based on our earlier work [11, 6], where we introduced vectorisation as a *total* transformation on a source language that included only vectorisable constructs, types, and primitive functions. In what follows we show how to extend this work to a source language that does not have this convenient property. We do this by specifying a *partial* transformation that may fail for some expressions, and by precisely characterising which parts of an expression are vectorised, and which are not. Our implementation uses a particular source language — namely, GHC's Core language [12] – but our method will work for *any* language.

In the following, we use double square brackets $[\![\cdot]\!]$ not only to denote source code fragments that are transformed by one of our transformation functions (i.e., to denote source code "arguments"), but also for the source code "results" of these transformation functions. In other words, we use $[\![\cdot]\!]$ much like quasi-quotes in meta-programming systems, such as Template Haskell.

We will consider only *monomorphic* programs. Our system is quite capable of handling polymorphism (and must do so for Haskell), but polymorphism adds complications that distract from the main point of this paper, which is partial vectorisation.

Figure 3 displays the fragment of Core that is relevant for the present paper. The left-hand side column gives the names for the syntactic categories that we use in the following to give type signatures to translation schemes.

3.1 Vectorising types

In general, if $e :: t$ then $\mathcal{V}[\![e]\!] :: \mathcal{T}[\![t]\!]$. In such situations it is usually illuminating to look at the type transform first. There is one case for each form of type in Figure 3[2]:

$\mathcal{T}[\![\cdot]\!]$ :: Type -> Type
$\mathcal{T}[\![(\texttt{->})]\!]$ = (:->)
$\mathcal{T}[\![\texttt{[::]}]\!]$ = PA
$\mathcal{T}[\![\texttt{T}]\!]$ | $\langle \texttt{T}_\texttt{V}$ exists$\rangle$ = $\texttt{T}_\texttt{V}$
$\mathcal{T}[\![t_1\ \ t_2]\!]$ = $\mathcal{T}[\![t_1]\!]\ \mathcal{T}[\![t_2]\!]$

The type transform simply replaces functions (->) with vectorised functions (:->), and arrays ([::]) with vectorised arrays (PA), and other data types T with a vectorised version of that type, $\texttt{T}_\texttt{V}$.

In general, like the term transform, the type transform is *partial*: given a type constructor T, $\mathcal{T}[\![\texttt{T}]\!]$ fails if $\texttt{T}_\texttt{V}$ does not exist. There are two cases to consider: either T is a *primitve* type, or it is an *algebraic data type*, which we consider next in turn.

Primitive types. For some primitive types, such as Int, the vectorised version is the same as the ordinary version; that is, $\texttt{Int}_\texttt{V} = \texttt{Int}$. But for other primitive types, there might *be* no vectorised version; for example $\mathcal{T}[\![\texttt{IO}]\!]$ fails, because there is no vectorised version $\texttt{IO}_\texttt{V}$ of Haskell's IO monad.

User-defined algebraic data types. Suppose T is a user-defined algebraic data type T. In order to vectorise code involving T we need its vectorised version $\texttt{T}_\texttt{V}$. We can generate $\texttt{T}_\texttt{V}$ from T by by vectorising its component types in the obvious way. Thus, for example,

```
data T  = C t1 t2 | D
```

generates the new data type declaration:

$\texttt{data T}_\texttt{V}\ \texttt{=}\ \texttt{C}_\texttt{V}\ \mathcal{T}[\![\texttt{t1}]\!]\ \mathcal{T}[\![\texttt{t2}]\!]\ \texttt{|}\ \texttt{D}_\texttt{V}$

If any of the argument types cannot be vectorised, then neither can T. In the special (but very common) case where $\mathcal{T}[\![\texttt{t1}]\!] = \texttt{t1}$ and $\mathcal{T}[\![\texttt{t2}]\!] = \texttt{t2}$, we can avoid creating a fresh data type, instead simply setting $\texttt{T}_\texttt{V} = \texttt{T}$, just as we do for Int. So, returning to Figure 1, we have $\texttt{MassPnt}_\texttt{V}$ = MassPnt and $\texttt{Particle}_\texttt{V}$ = Particle. In contrast, for Tree we get

$\texttt{data Tree}_\texttt{V}\ \texttt{=}\ \texttt{Node}_\texttt{V}\ \texttt{MassPnt (PA Tree}_\texttt{V}\texttt{)}$

[2] We use Haskell's guard notation here. The guard "| $\langle \texttt{T}_\texttt{V}$ exists$\rangle$" means "this equation applies only if $\texttt{T}_\texttt{V}$ exists".

Functions. The vectorised version of function arrow `(->)` is the type of vectorised functions `(:->)`, but how is `(:->)` defined? Consider the following (contrived) example:

```
app :: (Int -> Int) -> (Int, [:Int:])
app f = (f 1, [:f x | x <- [:1, 2, 3:]:])
```

Here we apply `f` outside and inside an array comprehension; in the former case we must run `f` sequentially, but in the latter it should be evaluated in parallel. To support parallel application of `f`, vectorisation generates a data-parallel, or *lifted* version of `f`, denoted by $\texttt{f}^{\uparrow}$, such that if `f :: t -> u`, then $\texttt{f}^{\uparrow}$ `:: PA` $\mathcal{T}[\![\texttt{t}]\!]$ `->` `PA` $\mathcal{T}[\![\texttt{u}]\!]$ (for full details of lifting, see [11]). In the fully-vectorised version of `app`, we therefore need `f`'s regular as well as its lifted variant, so we must pass *both* versions of `f` to $\texttt{app}_\text{V}$. To a first approximation, therefore, the type `(:->)` is defined thus:

```
data a :-> b = MkFun (a -> b) (PA a -> PA b)
```

That is, vectorisation replaces a function of type `t -> u` by a *pair* of functions, of type (`(`$\mathcal{T}[\![\texttt{t}]\!]$ `->` $\mathcal{T}[\![\texttt{u}]\!]$`, PA` $\mathcal{T}[\![\texttt{t}]\!]$ `->` `PA` $\mathcal{T}[\![\texttt{u}]\!]$`)`. This definition is not quite right, because of nested functions and partial applications [6], but the details are not important for this paper. All that we need is the existence of the vectorised function constructor `(:->)`, and its apply operator

```
($:) :: (a :-> b) -> a -> b
```

Vectorised arrays. Under selective vectorisation, the non-vectorised (and hence sequential) part of the program may still manipulate "parallel" arrays. For example, we might read a file to create a parallel array of type `[:Int:]`, that is then passed to a vectorised computation. Conversely, in the function `oneStepIO` in Figure 2, we consume a parallel array produced by a vectorised computation in sequential I/O code.

While the type of parallel arrays `[:a:]` in the source language is parametric, the representation of arrays and array operations after vectorisation depends on the element type. For example, an array of pairs is represented as a pair of arrays. The reasons for this requirement, and a sketch of how we realise this in our implementation using *type families*, are provided in previous work [4, 8].

Concretely, `PA` is a type-indexed data type family representing vectorised arrays. In GHC's type-family notation [13] we write

```
data family PA (a::*)
```

Then we give a `data instance` declaration for each type that we want to store in a vectorised array. For example:

```
data instance PA (a,b) = PAPair (PA a) (PA b)
```

Hence, for each user-defined algebraic data type, we must generate a `data instance` declaration that describes how a vectorised array of such values is represented. For example, the `Tree` type in Figure 1 generates the following declaration:

```
data instance PA Tree = Node_PA (PA MassPnt) (Segd, PA Tree)
```

`Segd` is a *segment descriptor* encoding the structure of nested arrays; c.f., for example [8] for more details of our use of type-indexed data types.

3.2 Vectorising expressions

Now we are ready to consider the selective vectorisation of expressions. Our approach relies on three mutually recursive transformation schemes defined in Figure 4:

```
𝒱⟦·⟧  :: Expr t -> Env -> Maybe (Expr 𝒯⟦t⟧)
𝒮⟦·⟧  :: Expr t -> Env -> Expr t
𝒮𝒱⟦·⟧ :: Expr t -> Env -> (Maybe (Expr 𝒯⟦t⟧), Expr t)
```

Full vectorisation $\mathcal{V}[\![\cdot]\!]$, and selective vectorisation $\mathcal{S}[\![\cdot]\!]$, have already been introduced, although here we give them types that (a) express partiality by returning a `Maybe`, and (b) express the type transformation by parameterising `Expr`.

The definitions of $\mathcal{V}[\![\cdot]\!]$ and $\mathcal{S}[\![\cdot]\!]$ do not directly depend on each other. Instead, the recursive knot is tied by $\mathcal{SV}[\![\cdot]\!]$ which uses both transformations to transform sub-expressions. $\mathcal{SV}[\![\cdot]\!]$ acts as a mediator between selective and partial vectorisation. Its main task is to intertwine vectorised and unvectorised code by introducing appropriate conversions.

The transformations are parametrised with an environment which maps variables to their vectorised versions if available. This information is required by partial vectorisation to transform variables, as apparent in the corresponding rule taken from Figure 4:

```
𝒱⟦x⟧ env
  | (x ↦ x_V) ∈ env = Just ⟦x_V⟧
  | otherwise       = Nothing
```

In the following, we look at the transformations in more detail and explain what happens at the interfaces between vectorised and unvectorised code.

3.3 Embedding unvectorised sub-expressions

To see how the transformations defined in Figure 4 allow for mixing vectorised and unvectorised code, let us consider an example that demonstrates how vectorised code may depend on unvectorised code. Assume a variable `M.constTable :: [:Int:]` defined in a module `M` that was not compiled with vectorisation; i.e., $\texttt{M.constTable}_\text{V}$ does not exist. In a naive implementation, we might abandon the vectorisation of an expression such as `sumP M.constTable` altogether and evaluate it sequentially. However, this is clearly suboptimal; instead, we would like to convert `M.constTable` to a vectorised representation (this is easily possible for arrays of primitive types) and pass it to the vectorised, i.e., parallel implementation of `sumP`. Ultimately, we would like to have

```
𝒮⟦·⟧ :: Expr t -> Env -> Expr t
𝒮⟦x⟧ env     = ⟦x⟧
𝒮⟦e₁ e₂⟧ env = ⟦e₁S e₂S⟧
  where
    (_, e₁S) = 𝒮𝒱⟦e₁⟧ env
    (_, e₂S) = 𝒮𝒱⟦e₂⟧ env

...⟨similar for other cases of 𝒮⟦·⟧⟩...

𝒮⟦let x :: t = e₁ in e₂⟧ env
  = ⟦let bs in e₂S⟧
  where
    (bs, env') = 𝒮𝒱_B⟦x :: t = e₁⟧ env
    e₂S = 𝒮⟦e₂⟧ env'

𝒱⟦·⟧ :: Expr t -> Env -> Maybe (Expr 𝒯⟦t⟧)
𝒱⟦x⟧ env
  | (x ↦ xV) ∈ env = Just ⟦xV⟧
  | otherwise      = Nothing
𝒱⟦e₁ e₂⟧ env
  | (Just e₁V, _) <- 𝒮𝒱⟦e₁⟧ env
  , (Just e₂V, _) <- 𝒮𝒱⟦e₂⟧ env = Just ⟦e₁V $: e₂V⟧
  | otherwise                  = Nothing
𝒱⟦let x :: t = e₁ in e₂⟧ env
  | (Just e₂V, _) <- 𝒮𝒱⟦e₂⟧ env' = Just ⟦let bs in e₂⟧
  | otherwise                    = Nothing
  where
    (bs, env') = 𝒮𝒱_B⟦x :: t = e₁⟧ env

...⟨similar for other cases of 𝒱⟦·⟧⟩...

𝒮𝒱⟦·⟧ :: Expr t -> Env -> (Maybe (Expr 𝒯⟦t⟧), Expr t)
𝒮𝒱⟦e⟧ env = case 𝒱⟦e⟧ env of
              Just eV
                | ⟨fromV eV exists⟩ -> (Just eV, ⟦fromV eV⟧)
                | otherwise         -> (Just eV, eS)
              Nothing
                | ⟨toV eS exists⟩   -> (Just ⟦toV eS⟧, eS)
                | otherwise         -> (Nothing, eS)
  where
    eS = 𝒮⟦e⟧ env

𝒮𝒱_B⟦·⟧ :: Binding -> Env -> ([Binding], Env)
𝒮𝒱_B⟦x :: t = e⟧ env
  | ⟨t is vectorisable⟩
  , (Just eV, eS) <- 𝒮𝒱⟦e⟧ env' = ([x::t = eS, xV::tV = eV], env')
  | otherwise                   = ([x::t = 𝒮⟦e⟧ env], env)
  where
    env'  = env ∪ {x ↦ xV}
```

Fig. 4. Selective vectorisation

$$\mathcal{V}[\![\texttt{sumP M.constTable}]\!]\ \texttt{env} = \texttt{Just}\ [\![\texttt{sumP}_\text{V}\ \texttt{\$:}\ (\texttt{toV M.constTable})]\!]$$

In other words, we would like vectorisation to succeed for the entire expression even though it fails for one of the sub-expressions. That is why $\mathcal{V}[\![\cdot]\!]$ uses $\mathcal{SV}[\![\cdot]\!]$ to vectorise sub-expressions as it is the latter that can introduce the necessary conversions. For example, the rule for vectorising application given in Figure 4 passes the two sub-expressions on to $\mathcal{SV}[\![\cdot]\!]$ which tries to transform them such that they can be used in a vectorised context. In our example, vectorisation immediately succeeds for `sumP` which has a vectorised version:

$$\mathcal{SV}[\![\texttt{sumP}]\!]\ \texttt{env} = (\texttt{Just}\ [\![\texttt{sumP}_\text{V}]\!], [\![\texttt{sumP}_\text{S}]\!])$$

but fails for `M.constTable` which has no vectorised variant. Fortunately, $\mathcal{SV}[\![\cdot]\!]$ is able to rectify this by introducing a conversion:

$$\begin{aligned}&\mathcal{SV}[\![\texttt{M.constTable}]\!]\ \texttt{env}\\&\quad = (\texttt{Just}\ [\![\texttt{toV M.constTable}_\text{S}]\!]), [\![\texttt{M.constTable}_\text{S}]\!])\end{aligned}$$

This enables the application rule of $\mathcal{V}[\![\cdot]\!]$ to succeed, producing the desired result. Note that the definition of $\mathcal{V}[\![\cdot]\!]$ does not contain any interfacing logic—interfacing is delegated entirely to $\mathcal{SV}[\![\cdot]\!]$. This is also the reason why we only included three example rules in the definition of $\mathcal{V}[\![\cdot]\!]$—the complete definition can be obtained by using $\mathcal{SV}[\![\cdot]\!]$ in place of direct recursion in the definition of vectorisation given in [11] and by accounting for partiality in the exact same manner as we demonstrated for application.

3.4 Vectorising as much as possible

Using unvectorised in vectorised code is only half of the story, however. Arguably much more important is the ability to pass results of parallel computations to inherently unvectorisable tasks such as I/O. In fact, we have already seen an example where this is absolutely essential: the statement `print qs` in Figure 2 outputs the result of a computation which we expect to be executed in parallel and which, therefore, must be vectorised. Again, it is the task of $\mathcal{SV}[\![\cdot]\!]$ to introduce the necessary conversion before passing the computed value to the unvectorisable `print`.

The mechanism employed here is quite similar to the one discussed in the previous section. Since we cannot vectorise `print`, we have:

$$\mathcal{SV}[\![\texttt{print}]\!]\ \texttt{env} = (\texttt{Nothing}, [\![\texttt{print}_\text{S}]\!])$$

The situation is quite different for `qs`, however. Not only can it be fully vectorised to $\texttt{qs}_\text{V}$, the latter can also be converted to an unvectorised representation. In such a case, $\mathcal{SV}[\![\cdot]\!]$ will throw away the result of selectively vectorising the sub-expression and simply suitably convert the fully vectorised version. In our example, this amounts to:

$$\mathcal{SV}[\![\texttt{qs}]\!]\ \texttt{env} = (\texttt{Just}\ [\![\texttt{qs}_\text{V}]\!], [\![\texttt{fromV qs}_\text{V}]\!])$$

For the overall expression, the transformation rule for selectively vectorising applications then generates

$$\mathcal{S}[\![\texttt{print qs}]\!]\ \texttt{env} = [\![\texttt{print}_\text{S}\ (\texttt{fromV qs}_\text{V})]\!]$$

This is optimal in the sense that we evaluate as much as possible in parallel (namely all of `qs`) before passing the result to the inherently sequential code. The definitions of $\mathcal{S}[\![\cdot]\!]$ and $\mathcal{SV}[\![\cdot]\!]$ ensure that the transformation always prefers fully vectorised expression to using selectively vectorised ones, thereby preserving the maximum degree of parallelism while still allowing for partiality of vectorisation.

3.5 Vectorising `let`

During selective vectorisation, `let` bindings are handled in just the same way as that described in Sections 2.2 and 2.3. Given a single[3] binding $x\texttt{::}t\ \texttt{=}\ e$, we always produce the selectively vectorised binding $x\texttt{::}t\ \texttt{=}\ \mathcal{S}[\![e]\!]$. Additionally, if e can be fully vectorised, we generate the fully vectorised binding $x_\text{V}\texttt{::}\mathcal{T}[\![t]\!]\ \texttt{=}\ \mathcal{V}[\![e]\!]$. In this case, we also extend the environment to account for the newly introduced vectorised version of x such that it is visible during the vectorisation of the body. This is described by the `let` cases of $\mathcal{V}[\![\cdot]\!]$ and $\mathcal{S}[\![\cdot]\!]$, which invoke the transformation scheme $\mathcal{SV}_B[\![\cdot]\!]$ to deal with the binding. The same transformation is used to handle top-level bindings.

As an example, the body of `moveParticle` from Figure 1 is vectorised as follows:

$$\begin{array}{l}\mathcal{V}[\![\texttt{let v = velocity p in Particle (movePnt (center p) v) v}]\!]\ \texttt{env}\\ \quad\texttt{= Just}\ [\![\texttt{let}\ \texttt{v}_\text{V}\ \texttt{=}\ \texttt{velocity}_\text{V}\ \texttt{\$:}\ \texttt{p}_\text{V}\\ \qquad\qquad\texttt{v}\ \ \texttt{= fromV (velocity}_\text{V}\ \texttt{\$:}\ \texttt{p}_\text{V}\texttt{)}\\ \qquad\texttt{in Particle}_\text{V}\ \texttt{\$:}\\ \qquad\qquad\texttt{(movePnt}_\text{V}\ \texttt{\$: (center}_\text{V}\ \texttt{\$:}\ \texttt{p}_\text{V}\texttt{)}\ \texttt{\$:}\ \texttt{v}_\text{V}\texttt{)}\ \texttt{\$:}\ \texttt{v}_\text{V}]\!]\end{array}$$

Obviously, there is ample room for optimisation here. For example, when $\mathcal{SV}_B[\![\cdot]\!]$ generates two bindings in a local `let`, only one of them may be used in the `let` body, but simple dead-code elimination will excise the unused binding.

What about recursion?[4] Whether the right-hand side of a recursive binding such as `r :: t = f r` can be vectorised depends, among other things, on whether a fully vectorised version of `r` is available. Of course, we cannot know if $\texttt{r}_\text{V}$ is available until we have tried to vectorise the right-hand side. Obviously, this is a classical case of circular dependency which our transformation must resolve.

Our solution to this problem is somewhat simple-minded, but effective. First, we attempt to vectorise the right-hand side assuming that $\texttt{r}_\text{V}$ exists. If it can be

[3] Both in the definition of the transformation and in the subsequent discussion, we restrict ourselves to a single, possibly recursive binding of the form `let` x `::` t `=` e_1 `in` e_2. Our method easily generalises to multiple mutually-recursive bindings.

[4] NB: Haskell does not distinguish between value and function bindings. Any `let` binding can be recusive.

fully vectorised, we generate two bindings as described above using the two expressions produced by $\mathcal{SV}[\![\cdot]\!]$. If partial vectorisation fails, however, we are faced with an additional complication. We cannot use the selectively vectorised expression that was produced by $\mathcal{SV}[\![\cdot]\!]$ under the assumption that $\texttt{x}_\text{V}$ exists; after all, it may contain references to the latter. The following example demonstrates this. Suppose we have the following functions:

```
f   :: Int -> T
g   :: T -> Int
gV  :: TV :-> Int
```

Here, `g` has a vectorised version but `f` does not. Moreover, we assume that the type `T` is vectorisable but does not support conversion to and from $\texttt{T}_\text{V}$. Then, for the binding `x :: T = f (g x)` partial vectorisation would fail but selective vectorisation, if performed under the assumption that $\texttt{x}_\text{V}$ exists, would yield `x :: T = f (toV (`$\texttt{g}_\text{V}$ `$: `$\texttt{x}_\text{V}$`))`. This definition is unusable, however, as we have no binding for $\texttt{x}_\text{V}$ after all. We have to selectively vectorise the binding once more, this time omitting $\texttt{x}_\text{V}$ from the environment. In our example, this would leave the original binding unchanged which is, indeed, the only possible solution.

4 Selective vectorisation of modules

Between all the modules forming a program, some modules will be compiled with vectorisation and some will not. In the following, we discuss how these two types of modules fit together and what extra information vectorisation requires to be communicated between separately compiled modules.

4.1 Modules compiled without vectorisation

The scheme for vectorising expressions, and in particular bindings, presented before is designed such that modules compiled without vectorisation

1. do not have to be aware of vectorisation at all and
2. are just a special case of those compiled with vectorisation.

Concerning Point (1), vectorised modules still contain all the original type definitions of the source and, although bindings of the form v `::` t `=` e are transformed into v `::` t `=` e_S, their interface remains the same. Concerning Point (2), even if a module is compiled with vectorisation, it is conceivable that none of its functions or type declarations is actually vectorisable under partial vectorisation. In this case, the interface of the module remains as if it hadn't been compiled with vectorisation at all. (Nevertheless, some expression bodies may have been selectively vectorised.)

4.2 Interface files

GHC uses *interface files* to communicate exported type declarations and function signatures between separately compiled modules. Whenever a module `A` is compiled, an interface file `A.hi` is generated from its type information. Whenever a module `B` imports `A`, the compiler reads `A.hi` to obtain type information for all imported entities.

If optimisation is enabled (and vectorisation is a form of optimisation), GHC emits further information into interface files. This additional information includes information computed by code analysis (such as strictness information) as well as the right-hand sides of function definitions that are considered for cross-module inlining.

When a module is compiled with vectorisation, GHC includes the following additional information in the interface file:

- For each value or function binding `x`, GHC indicates whether $\texttt{x}_\text{V}$ exists.
- For each type constructor `T`, GHC indicates whether $\texttt{T}_\text{V}$ exists, and if so, whether `T` itself serves as $\texttt{T}_\text{V}$. If $\texttt{T}_\text{V}$ exists, we may also have the type-specific conversion functions `toV` and `fromV`.
- For each type constructor `T`, we have a type family instance defining the representation of the non-parametric array representation described in Section 3.1. This always exists as we use parametric, boxed arrays as a fallback for types where we cannot derive an optimised representation.

5 Related work

There is a long list of work concerning vectorisation, some of which we have mentioned throughout this paper. We have discussed much of this previous work in [8] and will refrain from repeating this discussion. To the best of our knowledge, none of this previous work has mentioned partial vectorisation. In particular, NESL [3, 7] was implemented as a whole program compiler performing wholesale vectorisation.

The Proteus system [14] promised a combination of data and control parallelism, but Proteus had a particular focus on manual refinement of algorithms and where data parallel components were automatically vectorised, this again was a complete whole-program transformation. Moreover, the system was never fully implemented.

Manticore [15] supports a range of forms of parallelism including nested data parallelism. Manticore employs some of the same techniques as we do, but it does not seem to use vectorisation in the same form. The Manticore implementation is, at the time of writing, work in progress.

6 Conclusion

We argued that vectorisation for a fully fledged functional language needs to be partial; i.e., only part of a program is vectorised. We presented and in part formalised a partial vectorisation transformation that vectorises sub-expressions

selectively and uses conversions where necessary to integrate vectorised and unvectorised code. At the time of writing, we are working on completing a first version of vectorisation for GHC, which includes our partial vectorisation strategy. All code is publicly accessible from the GHC HEAD repository at `http://darcs.haskell.org/ghc/`.

References

1. Blelloch, G.E.: Programming parallel algorithms. Communications of the ACM **39**(3) (1996) 85–97
2. Blelloch, G.E., Sabot, G.W.: Compiling collection-oriented languages onto massively parallel computers. Journal of Parallel and Distributed Computing **8** (1990) 119–134
3. Blelloch, G.E.: Vector Models for Data-Parallel Computing. The MIT Press (1990)
4. Keller, G., Chakravarty, M.M.T.: Flattening trees. In Pritchard, D., Reeve, J., eds.: Euro-Par'98, Parallel Processing. Number 1470 in Lecture Notes in Computer Science, Berlin, Springer-Verlag (1998) 709–719
5. Chakravarty, M.M.T., Keller, G.: More types for nested data parallel programming. In Wadler, P., ed.: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00), ACM Press (2000) 94–105
6. Leshchinskiy, R., Chakravarty, M.M.T., Keller, G.: Higher order flattening. In: Third International Workshop on Practical Aspects of High-level Parallel Programming (PAPP 2006). Number 3992 in LNCS, Springer-Verlag (2006)
7. Blelloch, G.E., Chatterjee, S., Hardwick, J.C., Sipelstein, J., Zagha, M.: Implementation of a portable nested data-parallel language. Journal of Parallel and Distributed Computing **21**(1) (April 1994) 4–14
8. Chakravarty, M.M.T., Leshchinskiy, R., Peyton Jones, S., Keller, G., Marlow, S.: Data Parallel Haskell: a status report. In: DAMP 2007: Workshop on Declarative Aspects of Multicore Programming, ACM Press (2007)
9. Chakravarty, M.M.T., Keller, G., Lechtchinsky, R., Pfannenstiel, W.: Nepal—nested data parallelism in Haskell. In Sakellariou, R., Keane, J., Gurd, J.R., Freeman, L., eds.: Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference. Number 2150 in Lecture Notes in Computer Science, Berlin, Germany, Springer-Verlag (2001) 524–534
10. Barnes, J., Hut, P.: A hierarchical $O(n \log n)$ force calculation algorithm. Nature **324** (December 1986)
11. Leshchinskiy, R.: Higher-Order Nested Data Parallelism: Semantics and Implementation. PhD thesis, Technische Universität Berlin (2005)
12. Peyton Jones, S., Santos, A.: A transformation-based optimiser for Haskell. Science of Computer Programming **32**(1–3) (1998) 3–47
13. GHC Team: Type families. `http://haskell.org/haskellwiki/GHC/Type_families` (2007)
14. Mills, P., Nyland, L., Prins, J., Reif, J.: Software issues in high-performance computing and a framework for the development of hpc applications. In: Computer Science Agendas for High Perfromance Computing, ACM Press (1994)
15. Fluet, M., Ford, N., Rainey, M., Reppy, J., Shaw, A., Xiao, Y.: Status report: The manticore project. In: 2007 ACM SIGPLAN Workshop on ML, ACM Press (2007)

Efficient Heap Management for Declarative Data Parallel Programming on Multicores

Clemens Grelck[1,2] and Sven-Bodo Scholz[1]

[1] University of Hertfordshire
Department of Computer Science
Hatfield, United Kingdom
c.grelck,sscholz@herts.ac.uk
[2] University of Lübeck
Institute of Software Technology and Programming Languages
Lübeck, Germany
grelck@isp.uni-luebeck.de

Abstract. Declarative data parallel programming for shared memory multiprocessor systems implies paradigm-specific demands on the organisation of memory management. As a key feature of declarative programming implicit memory management is indispensable. Yet, the memory objects to be managed are very different from those that are predominant in general-purpose functional or object-oriented languages. Rather than complex structures of relatively small items interconnected by references, we are faced with large chunks of memory, usually arrays, which often account for 100s of MB each. Such sizes make relocation of data or failure to update arrays in-place prohibitively expensive.
To address these challenges of the data parallel setting, the functional array language SaC employs continuous garbage collection via reference counting combined with several aggressive optimisation techniques. However, we have observed that overall memory performance does not only rely on efficient reference counting techniques, but to a similar extent on the underlying memory allocation strategy. As in the general memory management setting we can identify specific demands of the declarative data parallel setting on heap organisation.
In this paper, we propose a heap manager design tailor-made to the needs of concurrent executions of declarative data parallel programs whose memory management is based on reference counting. We present runtime measurements that quantify the impact of the proposed design and relate it to the performance of several different general purpose heap managers that are available in the public domain.

1 Introduction

The ubiquity of multicore processors has made parallel processing a mainstream necessity rather than a niche business [1]. Declarative languages may benefit from this paradigm shift as their problem-oriented nature and the absence of side-effects facilitate (semi-)implicit parallelisation or at least help in explicit

parallelisation on a high level of abstraction. One approach is to exploit data parallelism on arrays as pursued by functional languages such as SISAL [2], NESL [3] or SAC [4, 5] and recently also adopted by HASKELL [6].

Declarative processing of large arrays of data has a specific challenge known as the aggregate update problem [7]. The (otherwise very desirable) absence of side-effects prevents incremental in-place updates of array element values as they are abundant in imperative array processing. A naive solution requires to copy the entire array which quickly becomes prohibitive with increasing array size. Efficient declarative array processing requires a mechanism that determines when it is safe to update an array in place and when not. The decision to reuse memory associated with an argument array to store a result array also depends on the characteristics of the operation itself. With the prevailing tracing garbage collectors [8] this is generally not feasible. The only way to achieve in-place updates of arrays in main-stream functional languages seems to be making arrays stateful, either by language semantics as in ML [9] or through a proper functional integration of states via monads in Haskell [10] or uniqueness typing in Clean [11]. However, these approaches also enforce a very non-declarative style of programming as far as arrays are concerned [12].

To mitigate the aggregate update problem without compromising a declarative style of programming, SISAL, NESL and SAC use reference counting [8] as a basis for memory management. At runtime each array is associated with a reference counter that keeps track of the number of active references to an array. Reference counting allows us to release unused memory as early as possible and to update arrays destructively in suitable operations provided that the reference counter indicates no further pending references. Strict evaluation generally tends to reduce the number of pending references; it seems to be necessary to make this memory management technique effective.

Reference counting does have its well known downsides, e.g. memory overhead, de-allocation cascades or the difficulty to identify reference cycles. However, in the particular setting of array processing they are less severe: individual chunks of memory are relatively large, data is not deeply structured, and cyclic references typically precluded by language semantics. Static analysis can be used effectively to reduce the overhead inflicted by reference counter updates, to identify opportunities for immediate memory and even data reuse and to make reuse and de-allocation decisions already at compile time. Surveys of such techniques can be found in [13, 14].

Unlike most forms of tracing garbage collection, reference counting is just half the story. It leads to a sequence of allocation and de-allocation requests that still need to be mapped to the linear address space of a process by some underlying mechanism. This is often considered a non-issue because low-level *memory allocators* have been available for decades for explicit heap management in machine-oriented languages (see [8] for a survey). Yet, many (SAC) applications spend a considerable proportion of their execution time in the memory allocator. As soon as runtime performance is a criterion for the suitability of a declarative language for a certain application domain, every (milli-)second counts, and im-

provements in the interplay between the reference counting mechanism and the underlying heap manager can have a significant impact on overall performance.

We propose a heap manager that is tailor-made for the needs of multithreaded declarative array processing and reference counting. Our design aims at outperforming existing allocators by exploiting three distinct advantages: Firstly, we adapt allocation/de-allocation strategies to the specific characteristics of array processing and reference counting. For example, we expect a large variety in the size of requested memory chunks from very small to very large, but only a relatively small number of different chunk sizes. Furthermore, reference counting (unlike manual memory management) guarantees that any allocated chunk of memory is released eventually. Consequently, overhead may arbitrarily be split between allocation and de-allocation operations.

Secondly, we use a rich (internal) interface between reference counting mechanism and allocator, that allows us to exchange extra information and let our allocator benefit from static code analysis. In contrast, the standard interfaces between applications and allocators are very lean (e.g. `malloc` and `free` in C or `new` and `delete` in C++) and restrict the flow of information from the application to the allocator.

Thirdly, we tightly integrate our allocator with the multithreaded runtime system [15]. As soon as threads run truly simultaneously on a multiprocessor system or multicore processor, access to heap-internal data structures requires synchronisation, which adversely affects performance and may even serialise program execution through the back door. While some general-purpose memory allocators do take multithreading into account [16–18], they need to deal with a wide range of multithreaded program organisations reasonably well. In contrast, the multithreaded runtime organisation of compiler-parallelised code typically is rather restricted. For example, the automatic parallelisation feature of the SaC compiler [15] results in a runtime program organisation where the number of threads is limited by the number of hardware execution units, certain threads are a-priori known to execute exclusively and memory allocated by one thread is known to be de-allocated by the same thread.

The contributions of this paper are

- to quantify the impact of heap management on compiler-parallelised declarative array processing code,
- to identify specific aspects of heap management that allow a tailor-made heap manager to exploit distinctive advantages over off-the-shelf implementations,
- to propose the design of a tailor-made heap manager in the context of SaC
- and to evaluate the benefit of using private heap management.

The remainder of the paper is organised as follows. In Section 2 we illustrate the problem using a microbenchmark. In Section 3 we outline the design of the SaC private heap manager and demonstrate its impact on overall runtime performance in Section 4. Finally, Section 5 outlines some related work before we draw conclusions in Section 6.

2 Problem illustration

We illustrate the impact of memory allocators on overall runtimes by means of the small SaC example program shown in Fig. 1. The program emulates a memory allocation and de-allocation pattern typical for many data parallel applications: a data parallel operation is repetitively applied to a vector `A` of length `[10000000/X]`, as generated by the library function `mkarray`. The data parallel operation within the `for`-loop is defined in terms of a `with`-construct, a SaC array comprehension. For each element of `A`, it recomputes its value by first allocating a vector of length `X` and subsequently summing these elements up.[3] This creates a very common memory demand pattern: within a data parallel section each element computation requires the creation of a temporary array and, hence, some memory allocation and de-allocation. Furthermore, all allocations are of the same size which, again, is typical for many scientific applications such as those investigated in [20, 21, 19].

```
int main()
{
  A = mkarray( [10000000/X], 0);
  for (i=0; i<50; i+=1) {
    A = with {
          (. <= [idx] <= .) : A[idx] + sum( mkarray( [X], idx));
        }: modarray( A);
  }
  return( sum( A));
}
```

Fig. 1. Example SaC program

It should be noted here that we carefully restricted compiler optimisations in order to ensure that for this simple example allocations and de-allocations of intermediate vectors of length `X` effectively happen at runtime. Normally, the SaC compiler would fold the reduction operation (`sum`) and the build operation (`mkarray`) [22]. Even if that failed, memory management optimisations would pre-allocate memory for the temporary vector outside of the `with`-construct [14].

In order to quantify the impact of the memory allocator, we measure the runtimes of the program from Fig. 1 with varying values for `X`: 1000, 250, 100 and 25. The definition of the vector lengths of `A` and that of the intermediate vectors guarantee that, irrespective of the value of `X`, we always perform 500 million additions. This allows us to observe the impact of the allocator as the decrease of `X` corresponds to an increase in the number of memory allocations (and de-allocations). We observe the effect of 500.000, 2 million, 5 million and 20 million memory allocations and de-allocations, respectively. Fig. 2 shows program execution times of our example program on a 12-processor SUN Ultra Enterprise 4000 multiprocessor using the standard Solaris memory allocator.

[3] For any details about the `with`-construct as well as about the purely functional semantics of this rather imperative looking code see [4, 5].

Additional experiments using the Gnu allocator coming with LINUX essentially led to the same observations.

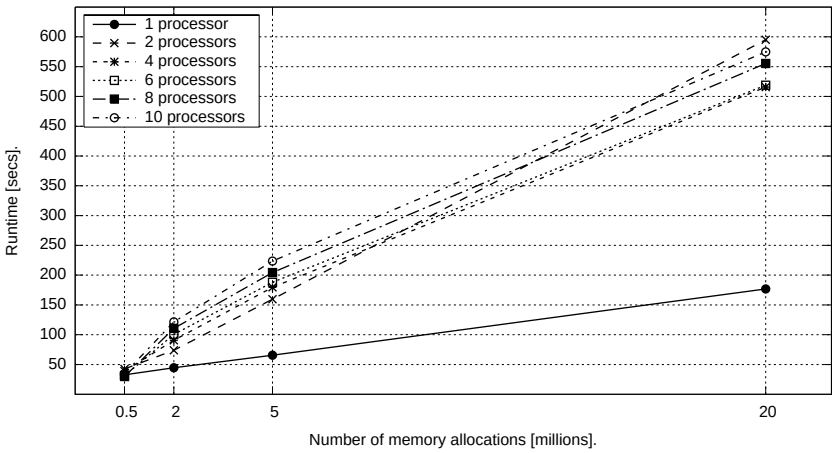


Fig. 2. Execution times of SAC program shown in Fig. 1

Focussing on single processor performance first, we can see that despite a fixed amount of overall computations, runtimes grow linearly with the number of memory allocations. Although this is not surprising in principle, the growth rate in fact is. From the measurements we can deduce that the pure computing time is roughly 30 seconds. Taking into account that even in the most allocation intensive case we have about 25 additions per allocation plus a certain loop overhead, the measured memory management overhead (roughly 140 seconds!) is extremely high. Obviously, the general-purpose allocators cannot benefit from the simple case of alternating allocations and de-allocations of always the same amount of memory. Our observations allow the conclusion that for these application scenarios the memory allocator is the key to overall runtime performance, whereas improvements in code generation and the compiler in general may easily be ineffective.

Looking at multiprocessor performance we observe that increasing parallelism yields substantial slowdowns, although our microbenchmark is almost embarrassingly parallel and the total workload is substantial. The reason for this behaviour lies in an inherently sequential design of the memory allocator, which seemingly has been adapted for multithreaded execution more or less naively by locking operations. This solution proves to be unsuitable for memory management intensive data parallel applications like our microbenchmark.

The bottom line of these observations is twofold: Firstly, a genuinely multithreaded memory allocator design that reduces locking to a minimum is indispensable. Secondly, data parallel applications may spend considerable proportions of their overall runtime in memory management operations, making the memory allocator a prime target for optimisation.

3 Design of a SAC-specific heap manager

The memory organisation used by the SAC *private heap manager* (or SACPHM for short) is characterised by a hierarchy of multiple nested heaps, as illustrated in Fig. 3. At the top of the hierarchy is a single *global heap*, which controls the entire address space of the process. It may actually grow or shrink during program execution, as additional memory is requested from the operating system or unused memory is released to it.

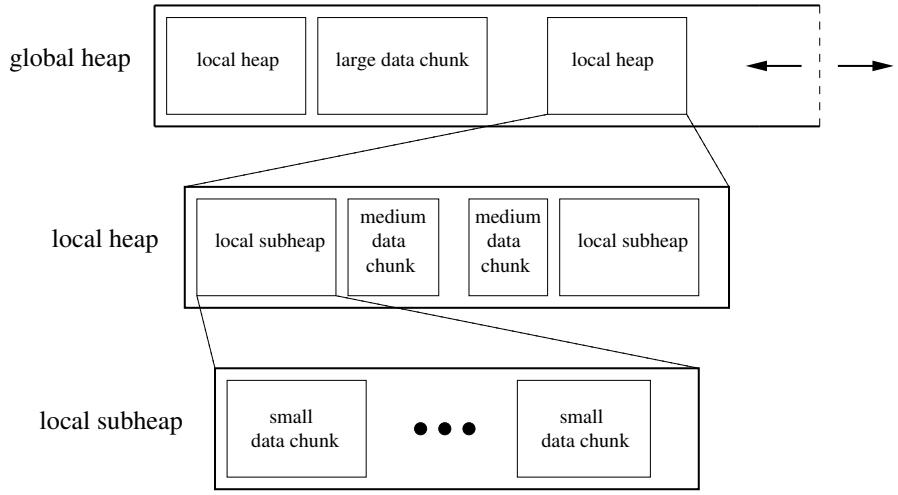


Fig. 3. Memory organisation using multiple nested heaps.

However, only very large chunks of memory are directly allocated from the global heap. Memory requests below some threshold size are satisfied from one of possibly several *local heaps*. A local heap is a contiguous piece of memory with a fixed size, which in turn is allocated from the global heap. Grouping together memory chunks of similar sizes tends to have a positive impact on memory fragmentation. Once the capacity of a local heap is exhausted, an additional local heap is allocated from the global heap.

In multithreaded execution each thread is associated with its individual local heap(s). This organisation addresses both scalability and false sharing [23] issues: Each thread may allocate and de-allocate arrays of up to a certain size without interfering with other threads. Small amounts of memory are guaranteed to be allocated from different parts of the address space if requested by different threads. Furthermore, housekeeping data structures for maintaining local heaps are kept separate by different threads. This allows us to keep them in processor-specific cache memories without invalidation by a cache coherence mechanism.

Three properties of the SAC multithreaded runtime system [15] are essential to make this design feasible. Firstly, the number of threads is limited by

the number of parallel processing units available (i.e. rather small), and thread creation/termination are limited to program startup/termination. Hence, it becomes feasible to a-priori associate each thread with some (non-negligible) local heap memory. Secondly, the data parallel approach encourages (though does not enforce) applications where the memory demands of the individual threads are rather similar. As a consequence, we may pre-allocate some heap memory for each thread at program startup when initialising the heap. Thirdly, the runtime system guarantees that any memory allocated by one thread is eventually released by the same thread. This restriction keeps thread-private local heaps in a coherent state throughout program execution. A general-purpose memory allocator cannot make such assumptions. Instead, it should work reasonably well both for large numbers of threads and heterogeneous allocation behaviour, including threads that mostly allocate memory while others mostly de-allocate memory following a producer/consumer pattern.

In our allocator design only accesses to the global heap may require synchronisation. The word *may* here is motivated by another restriction of the multithreaded runtime system: Program execution is organised as a sequence of alternating single-threaded and multithreaded supersteps [15]. Any memory management request to the global heap issued in a single-threaded superstep proceeds without synchronisation. Our experience is that very large arrays, allocated from the global heap, are predominantly allocated (and de-allocated) during single-threaded execution for subsequent multithreaded initialisation of elements. In practice, locking is reduced to the very rare case when the initially pre-allocated local heap of some thread is exhausted and needs to be extended during program execution.

The hierarchical memory organisation is repeated once again on the level of local heaps. Only medium-sized memory requests are directly satisfied by one of the local heaps. Allocations of memory chunks below a certain size are again grouped together in *local subheaps*, which in turn are allocated from local heaps. The distinction between heaps and subheaps is mainly motivated by different housekeeping mechanisms. In local heaps as well as in the global heap we allocate differently sized chunks from a contiguous address space, whereas subheaps use a fixed-size chunk allocation scheme. The latter allows us to quickly identify a suitable available memory chunk. Likewise, marking chunks as available or allocated inflicts very little time overhead. For larger chunks of memory, however, the resulting internal fragmentation is not tolerable. Therefore, we use a more expensive variable chunk size scheme above a certain threshold size. This scheme keeps track of chunk sizes and, in particular, splits larger parts of contiguous memory into pieces to accommodate allocation requests and coalesces adjacent free chunks of memory to conversely form larger chunks.

Both fixed and variable chunk size heap organisation schemes are well studied [8]. Nonetheless, we can customise our concrete implementation to specific aspects of data parallel array processing. In this context we often observe that applications only use a very restricted number of differently sized arrays (although the range of different array sizes may be very large). Therefore, we assume lo-

cality of time in similar way as cache memories do: If we de-allocate a memory chunk of some size, we consider it likely that we need to allocate a memory chunk of the same size very soon thereafter. Consequently, we employ a deferred coalescing scheme that only reconstructs larger chunks of memory if a concrete allocation request cannot be satisfied from the immediately available resources. Deferred coalescing moves overhead from de-allocations to allocations, which is not so desirable for general-purpose allocators because the number of allocations typically exceeds the number of de-allocations. However, with the allocator being a backend for the reference counting mechanism it is guaranteed that any allocated chunk of memory is released eventually. Hence, it doesn't matter whether we concentrate effort in allocation or in de-allocation operations.

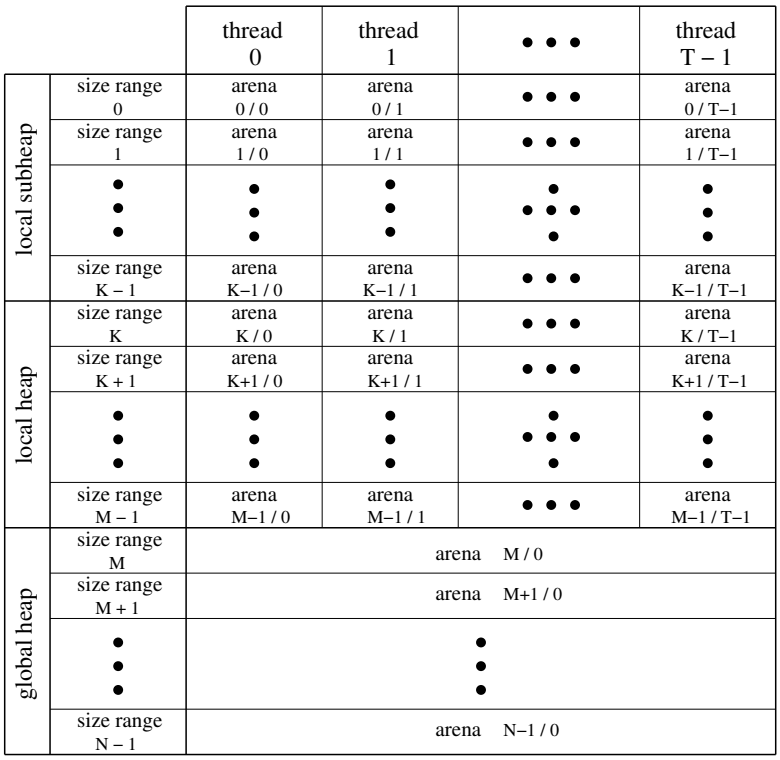


Fig. 4. Matrix of allocation arenas

The hierarchy of nested heaps requires a coarse-grained classification of memory requests into small, medium and large chunks. In order to accelerate searching for appropriate memory chunks, we effectively use a finer-grained classification and introduce an entire matrix of *allocation arenas*, as illustrated in Fig. 4. Allocation arenas represent the basic organisational entities for heap manage-

ment. Each request for allocation or de-allocation of memory first identifies the appropriate allocation arena, which provides

- a list of available, appropriately sized chunks of memory,
- an allocation strategy,
- a de-allocation strategy and
- a backup strategy to obtain more memory.

Arena identification is a binary decision problem whose depth is logarithmic in the number of allocation arenas. However, in practice the exact amount of memory needed to represent some array is often known statically to the program and, hence, also the appropriate allocation arena can be determined at compile time. It is the restricted interface of (e.g. `malloc` and `free`) that prevents general-purpose memory allocators from taking advantage of static chunk size knowledge. In contrast, our integrated solution employs a much richer interface between reference counting mechanism and backend heap manager that already selects the allocation arena at compile time whenever possible.

Likewise, the tight integration of reference counting and heap management permits specific optimisations. For example, at runtime any SaC array is represented by a (typically large) data vector and a (always small) descriptor that accommodates the reference counter and dynamic shape information. This design enables the seemless flow of arrays between program parts written in SaC and program parts written in other languages using the SaC foreign language interface. The separation of SaC-specific structural and administrative information from actual data, unfortunately, also requires two allocations and two de-allocations per array. In most cases allocation and de-allocation of data vector and descriptor are made in conjunction, but this is not guaranteed and upon de-allocation it is undecidable whether a data vector has been allocated within the realm of SaC or outside. However, our private heap manager makes exactly this decidable. This optimisation alone accounts for about 10% execution time improvement through a range of applications.

4 Evaluation

We have repeated the initial experiment described in Section 2 with the SaC private heap manager and three off-the-shelf multithreaded memory allocators:

- MTmalloc [16] is a replacement for the standard memory allocator, which is provided by SUN itself starting with Solaris-7.
- PTmalloc [17] adapts the serial allocator DLmalloc [24] for use with multithreaded applications. It also employs multiple heaps to reduce contention, but there is no static mapping of heaps to threads. Upon each memory request, threads search for an unlocked heap, lock the heap, and then apply the serial allocation/de-allocation techniques adopted from DLmalloc.
- Hoard [18] seems to be the most recent development in multithreaded memory managers. It maps a possibly large number of threads to a generally much smaller number of separate heaps by means of hashing techniques.

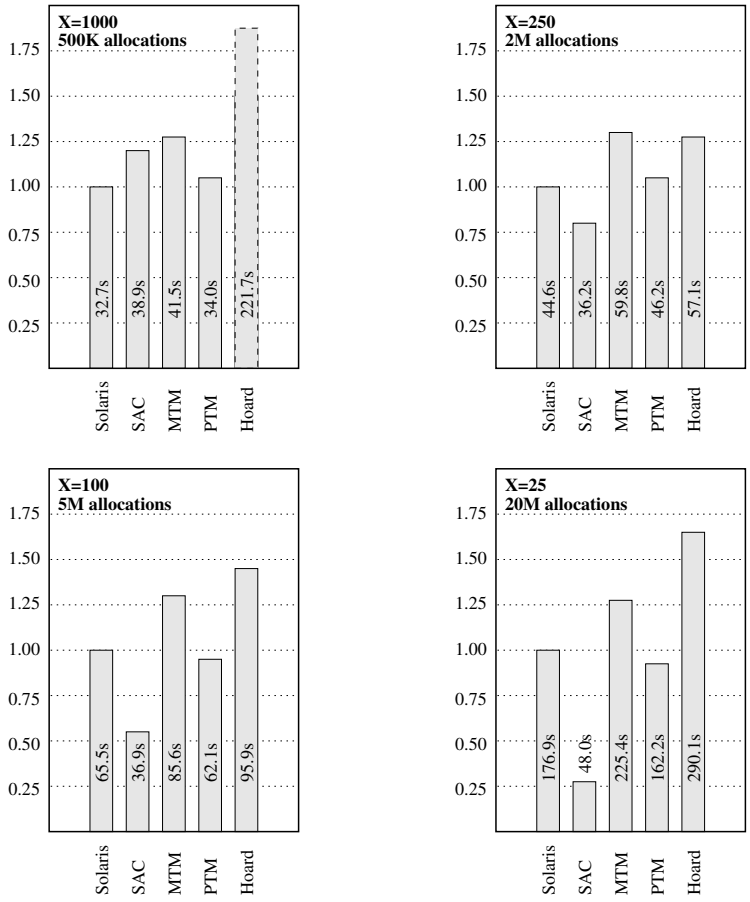


Fig. 5. Single processor performance of multithreaded allocators in comparison with the serial Solaris allocator as base line using the SAC microbenchmark of Fig. 1

Fig. 5 shows the single processor performance achieved by SacPhm and that of the three other multithreaded memory allocators in comparison with the serial Solaris allocator used in Section 2. Regardless of the concrete problem size, MTmalloc incurs a runtime overhead of about 25% compared with the serial allocator. For Hoard the respective overhead grows with increasing memory management frequency from about 25% to more than 60%. Surprisingly, performance is much worse for problem size X=1000, where single processor execution time exceeds that of any other allocator by almost an order of magnitude. Having a closer look at the implementation of Hoard reveals that memory requests exceeding a certain threshold size are directly mapped to virtual memory by using the `mmap` and `munmap` system routines. Obviously, their frequent application incurs prohibitive overhead.

In contrast, PTmalloc performs similar to the serial allocators, slightly outperforming them with increased memory management frequency. For SacPhm

it can be observed that starting out with a performance loss of about 20% relative to standard allocators for problem size X=1000, this slowdown turns into a significant speedup with increasing frequency of memory allocations and deallocations. With overall execution time being clearly dominated by dynamic memory management overhead for X=25, the SAC-specific memory allocator makes the overall program run 3.7 times faster than with the SOLARIS allocator.

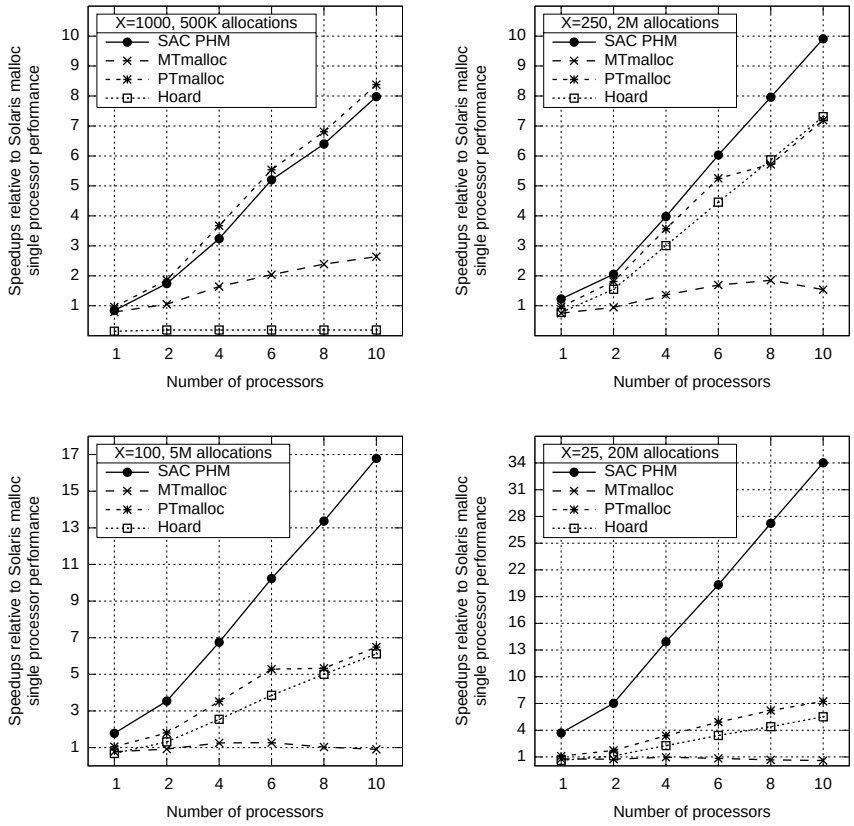


Fig. 6. Multi processor performance of multithreaded allocators in comparison with the serial SOLARIS allocator as base line using the SAC microbenchmark of Fig. 1

Fig. 6 shows the multiprocessor performance of the multithreaded allocators relative to the base line set by the serial SOLARIS allocator. This "true" parallel performance takes the different sequential performance levels into account, hence the different starting points of the curves for a single processor. First of all, we observe that MTMALLOC scales rather poorly for all problem sizes investigated. This observation is rather surprising for an allocator that is particularly designed for exactly this scenario. However, it coincides with much more thorough investigations made by the developers of HOARD [18]. In contrast, PTMALLOC scales

fairly well; it is not clear why hardly any speedup can be observed when switching from 6 to 8 processors for some problem sizes, but additional measurements have confirmed these figures. High scalability can be observed for HOARD for all problem sizes that are not mapped directly to the virtual memory manager (i.e. X=1000). SACPHM turns out to scale as well as HOARD, but provides this scalability on top of a substantially higher single processor performance.

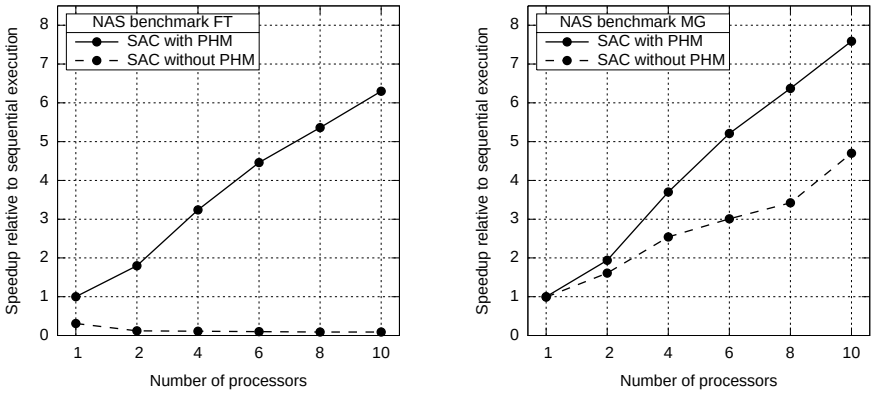


Fig. 7. Performance impact of SACPHM on NAS benchmarks FT (left) and MG (right)

In addition to our microbenchmark we have also investigated the impact of heap management on two non-trivial benchmarks from the NAS benchmark suite [25]: SAC implementations of 3-dimensional multigrid relaxation (benchmark MG [20]) and 3-dimensional fast-Fourier transforms (benchmark FT [21]). Fig. 7 quantifies the effect of SACPHM on runtime performance (size class A).

The two benchmarks show very different allocation/de-allocation patterns. FT uses fairly large arrays of complex numbers (256x256x128) that are exclusively allocated and de-allocated in single-threaded mode. We use the Danielson-Lanczos algorithm for 1d-FFTs on vectors of length 128 and 256, respectively, and explicitly create 2 (4) vectors of length 64, 4 (8) vectors of length 32, etc. All these allocations and the corresponding de-allocations happen during multi-threaded execution. Unsurprisingly, Fig. 7 shows SACPHM to be indispensable. However, our declarative implementation of FT is so much dominated by memory management overhead that the good single-processor performance of SACPHM on small chunks of data that are repeatedly allocated and de-allocated turns out to be crucial as well.

The benchmark MG implements a multigrid method that starts with arrays of size 256x256x256 and performs alternating convolution and mapping steps. In each mapping step the array size shrinks by a factor of two in each dimension until the minimum size of 4x4x4 is reached; further mapping steps let the array size grow again. Like in the FT benchmark, we are faced with a substantial number of allocations (and de-allocations) over a wide range of chunk sizes. However, in contrast to FT none of them occur during data-parallel operations.

As a consequence, the serial allocator performs reasonably well. Nevertheless, it takes SacPhm to achieve good overall speedups through a reduction of absolute overhead inflicted by dynamic memory management.

5 Related work

In the previous section have already acknowledged and evaluated several general-purpose memory allocators that are specifically designed for multithreaded program execution, namely SUN's MTmalloc [16], PTmalloc [17] and Hoard [18].

Only few declarative languages besides SaC explicitly focus on arrays. We mention Sisal [2] and Nesl [3], which both use reference counting. While the developers of Sisal spent considerable effort into efficient reference counting [13], they left the underlying heap management issues to the C system library [26].

In Nesl the VCode interpreter takes responsibility for memory management [27, 28]. Its design differs from our solution in various aspects. Firstly, by making the reference counter a part of the heap administration data structures Nesl fully integrates reference counting with its own heap management. In contrast, we explicitly allocate reference counters (as part of a more general array descriptor) on the heap. This approach allows us to employ (third party) heap managers that are fully unaware of our reference counting scheme for experimental comparisons like the one in Section 4 as well as for backup reasons. Secondly, the Nesl solution organises the heap differently. While Nesl does use multiple free lists for different chunk sizes for the same purposes as we do, it nevertheless allocates all chunk sizes from the same contiguous address space. In contrast, our allocation arenas (inspired from general-purpose multithreaded allocator designs) actually keep differently sized chunks in different areas of the address space. This design has a positive impact on fragmentation and solves the false sharing problem. Thirdly, we haven't found any information concerning multithreaded heap management in the context of Nesl, and the solution described in [27] does not support concurrently executing threads.

6 Conclusion

We have outlined an important aspect of the memory management subsystem of the functional array language SaC: the integration between the reference counting mechanism that decides when to allocate and de-allocate heap memory and the underlying heap manager that maps concurrent allocation and de-allocation requests of multiple threads to the linear address space of a process. Empirical data shows the significance of an integrated approach to achieve good runtime performance in declarative array processing on multiprocessor and multicore systems.

As soon as runtime performance is an issue (and in parallel processing it usually is), declarative programming languages often find themselves in a defensive position. In machine-oriented programming languages one typically blames the

application programmer (rather than the C compiler, for instance) for unsatisfactory performance. Frequently, additional effort and expert knowledge manage to improve performance, albeit often at the expense of readability and portability. Declarative programming languages raise the level of abstraction in programming from a machine-oriented to a problem-oriented view. Yet, they need to meet the programmer's performance expectations. The more performance matters, the more difficult this is to achieve.

Automatic memory management plays a crucial role here because it is a key feature of declarative languages and it must directly compete with manual dynamic memory management or even static memory layouts used by machine-oriented approaches. From the user's perspective it is indistinguishable whether unsatisfactory performance is caused by inefficiencies in compilation/parallelisation schemes or by false assumptions of an off-the-shelf memory allocator. The bottom line is that it takes a fully integrated approach to be successful: code generation needs to be well integrated with reference counting to directly reuse memory as often as possible, and reference counting needs to be well integrated with an underlying heap manager to reduce the overhead inflicted by remaining allocations and de-allocations. Furthermore, the heap manager needs to be integrated with the multithreaded runtime system to avoid costly synchronisation when concurrent threads access the implicitly shared heap.

References

1. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal **30** (2005)
2. Cann, D.: Retire Fortran? A Debate Rekindled. CACM **35** (1992)
3. Blelloch, G.E.: Programming Parallel Algorithms. CACM **39** (1996)
4. Scholz, S.B.: Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. J. Functional Programming **13** (2003) 1005–1059
5. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multithreaded execution. Intern. Journal of Parallel Programming **34** (2006) 383–427
6. Chakravarty, M.M.T., Leshchinskiy, R., Peyton Jones, S., Keller, G., Marlow, S.: Data parallel haskell: a status report. In: Workshop on Declarative Aspects of Multicore Programming (DAMP'07), Nice, France, ACM Press (2007)
7. Hudak, P., Bloss, A.: The Aggregate Update Problem in Functional Programming Systems. In: 12th ACM Symposium on Principles of Programming Languages (POPL'85), New Orleans, USA, ACM Press (1985) 300–313
8. Wilson, P.R., Johnstone, M.S., Neely, M., Boles, D.: Dynamic Storage Allocation: A Survey and Critical Review. In: International Workshop on Memory Management (IWMM'95), Kinross, UK. LNCS 986, Springer-Verlag (1995) 1–116
9. Milner, R., Tofte, M., Harper, R.: The Definition of Standard ML. MIT Press, Cambridge, USA (1990)
10. Peyton Jones, S., Launchbury, J.: State in Haskell. Lisp and Symbolic Computation **8** (1995) 293–341
11. Smetsers, S., Barendsen, E., van Eekelen, M., Plasmeijer, M.: Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. University of Nijmegen, The Netherlands (1993)

12. Serrarens, P.: Implementing the Conjugate Gradient Algorithm in a Functional Language. In: 8th International Workshop on Implementation of Functional Languages (IFL'96), Bonn, Germany. LNCS 1268, Springer-Verlag, (1997) 125–140
13. Cann, D., Evripidou, P.: Advanced Array Optimizations for High Performance Functional Languages. IEEE Trans. on Parallel and Distributed Systems **6** (1995)
14. Grelck, C., Trojahner, K.: Implicit Memory Management for SAC. In: 16th International Workshop on Implementation and Application of Functional Languages (IFL'04), Lübeck, Germany (2004) 335–348
15. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. J. Functional Programming **15** (2005) 353–401
16. Sun Microsystems Inc.: A Comparison of Memory Allocators in Multiprocessors. Solaris Developer Connection, Sun Microsystems Inc., Mountain View, USA (2000)
17. Gloger, W.: Dynamic Memory Allocator Implementations in Linux System Libraries. 4th International Linux Kongress (LK'97), Würzburg, Germany (1997)
18. Berger, E., McKinley, K., Blumofe, R., Wilson, P.: Hoard: A Scalable Memory Allocator for Multithreaded Applications. In: 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), Cambridge, USA. ACM Press (2000) 117–128
19. Shafarenko, A., et.al.: Implementing a numerical solution of the KPI equation using Single Assignment C: lessons and experiences. In: Implementation and Application of Functional Languages, 17th International Workshop (IFL'05), Dublin, Ireland. LNCS 4015, Springer-Verlag (2006)
20. Grelck, C.: Implementing the NAS Benchmark MG in SAC. In: 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, USA, IEEE Computer Society Press (2002)
21. Grelck, C., Scholz, S.B.: Towards an Efficient Functional Implementation of the NAS Benchmark FT. In: 7th International Conference on Parallel Computing Technologies (PaCT'03), Nizhni Novgorod, Russia. LNCS 2763, Springer-Verlag (2003) 230–235
22. Scholz, S.B.: With-loop-folding in SAC — Condensing Consecutive Array Operations. In: Implementation of Functional Languages, 9th International Workshop (IFL'97), St. Andrews, UK. LNCS 1467, Springer-Verlag (1998) 72–92
23. Torellas, J., Lam, M., Hennessy, J.: False Sharing and Spatial Locality in Multiprocessor Caches. IEEE Transactions on Computers **43** (1994) 651–663
24. Lea, D.: A Memory Allocator. Unix/Mail **6/96** (1996)
25. van der Wijngart, R.: NAS Parallel Benchmarks Version 2.4. Technical Report NAS-02-007, NASA Ames Research Center, Moffet Field, USA (2002)
26. Cann, D.C.: Compilation Techniques for High Performance Applicative Computation. Technical Report CS-89-108, Lawrence Livermore National Lab, Livermore, USA (1989)
27. Blelloch, G., Chatterjee, S., Hardwick, J., Sipelstein, J., Zagha, M.: Implementation of a Portable Nested Data-Parallel Language. Technical Report CMU-CS-93-112, Carnegie Mellon University, Pittsburgh, USA (1993)
28. Blelloch, G., Chatterjee, S., Hardwick, J., Sipelstein, J., Zagha, M.: Implementation of a Portable Nested Data-Parallel Language. Journal of Parallel and Distributed Computing **21** (1994) 4–14

Implementing Joins using Extensible Pattern Matching

Philipp Haller[1], Tom Van Cutsem[2⋆]

[1] EPFL, 1015 Lausanne, Switzerland
firstname.lastname@epfl.ch
+41 21 693 6483, +41 21 693 6660
[2] Programming Technology Lab, Vrije Universiteit Brussel, Belgium

Abstract. Join patterns are an attractive declarative way to synchronize both threads and asynchronous distributed computations. We explore joins in the context of extensible pattern matching that recently appeared in languages such as F# and Scala. Our implementation supports Ada-style rendezvous, and constraints. Furthermore, we integrated joins into an existing actor-based concurrency framework. It enables join patterns to be used in the context of more advanced synchronization modes, such as future-type message sending and token-passing continuations.

Keywords: Concurrent Programming, Join Patterns, Chords, Actors

1 Introduction

Recently, the pattern matching facilities of languages such as Scala and F# have been generalized to allow representation independence for objects used in pattern matching [5,17]. Extensible patterns open up new possibilities for implementing abstractions in libraries which were previously only accessible as language features. More specifically, we claim that extensible pattern matching eases the construction of declarative approaches to synchronization in libraries rather than languages. To support this claim, we show how a concrete declarative synchronization construct, join patterns, can be implemented in Scala, a language with extensible pattern matching. Join patterns [8,9] offer a declarative way of synchronizing both threads and asynchronous distributed computations that is simple and powerful at the same time. They form part of functional languages such as JoCaml [7] and Funnel [12]. Join patterns have also been implemented as extensions to existing languages [2,19].

Recently, Russo [15] and Singh [16] have shown that advanced programming language features, such as generics or software transactional memory, make it feasible to provide join patterns as libraries rather than language extensions. As we will argue in section 2, an implementation using extensible pattern matching improves upon these previous approaches by providing a better integration between library and language. More concretely, we make the following contributions:

- Our implementation technique overcomes several limitations of previous library-based designs and language extensions. In all library-based implementations that

⋆ supported by a Ph.D. fellowship of the Research Foundation Flanders (FWO).

we know of, pattern variables are represented implicitly as parameters of join continuations. Mixing up parameters of the same type inside the join body may lead to obscure errors that are hard to detect. Our design avoids these errors by using the underlying pattern matcher to bind variables that are explicit in join patterns. The programmer may use a rich pattern syntax to express constraints using nested patterns and guards. However, efficiently supporting general guards in join patterns is currently an open problem, and we do not attempt to solve it.

– We present a complete implementation of our design as a Scala library[3] that supports Ada-style rendezvous and constraints. Moreover, we integrate our library into an existing event-based concurrency framework. This enables expressive join patterns to be used in the context of more advanced synchronization modes, such as future-type message sending and token-passing continuations. Our integration is notable in the sense that the new library provides a conservative syntax extension. That is, existing programs continue to run without change when compiled or linked against the extended framework.

The rest of this paper is structured as follows. In the following section we briefly highlight join patterns as a declarative synchronization abstraction, how they have been integrated in other languages before, and how combining them with pattern matching can improve this integration. Section 3.1 shows how to synchronize threads using join patterns written using our library. Section 3.2 shows how to use join patterns with actors. In section 4 we discuss a concrete Scala Joins implementation for threads and actors. Section 5 discusses related work, and section 6 concludes.

2 Motivation

Background: Join Patterns A join pattern consists of a body guarded by a linear set of events. The body is executed only when *all* of the events in the set have been signaled to an object. Threads may signal synchronous or asynchronous events to objects. By signaling a synchronous event to an object, threads may implicitly suspend. The simplest illustrative example of a join pattern is that of an unbounded FIFO buffer. In Cω, it is expressed as follows [2]:

```
public class Buffer {
  public async Put(int x);
  public int Get() & Put(int x) { return x; }
}
```

A detailed explanation of join patterns is outside the scope of this paper. For the purposes of this paper, it suffices to understand the operational effect of a join pattern. Threads may put values into a buffer `b` by invoking `b.Put(v)`. They may also read values from the buffer by invoking `b.Get()`. The join pattern `Get() & Put(int x)` (called a *chord* in Cω) specifies that a call to `Get` may only proceed if a `Put` event has previously been signaled. Hence, if there are no pending `Put` events, a thread invoking `Get` is automatically suspended until such an event is signaled.

[3] Available at `http://lamp.epfl.ch/~phaller/joins/`.

The advantage of join patterns is that they allow a *declarative* specification of the synchronization between different threads. Often, the join patterns correspond closely to a finite state machine that specifies the valid states of the object [2]. Section 3.1 provides a more illustrative example of the declarativeness of join patterns.

Existing library-based designs In Cω, join patterns are supported as a language extension through a dedicated compiler. This ensures that join patterns are well integrated in the language. With the introduction of generics in C#, Russo has made join patterns available as a regular library for C# 2.0 called Joins [15]. In that library, the `Buffer` example can be expressed as follows:

```
public class Buffer {
  // Declare (a)synchronous channels
  public readonly Asynchronous.Channel<int> Put;
  public readonly Synchronous<int>.Channel Get;
  public Buffer() {
    Join join = Join.Create();
    join.Initialize(out Put); join.Initialize(out Get); // initialize channels
    join.When(Get).And(Put).Do(delegate(int x) { return x; });
  }
}
```

In C# Joins, join patterns consist of linear combinations of channels and a delegate (a function object) which encapsulates the body. Join patterns are triggered by invoking channels, which are special delegates.

Even though the synchronization between `Get` and `Put` is still readily apparent in the above example, the Joins library design has some drawbacks. First and foremost, the way in which arguments are passed between the channels and the body is very implicit: the delegate is implicitly invoked with the value passed via the `Put` channel. Contrast this with the Cω example in which the variable `x` is explicitly tied to the `Put` message. Furthermore, because joins are defined by means of an ad hoc combination mechanism, it is impossible to declaratively specify additional pattern matches or even guards. For example, it is not possible to add a join pattern triggering only on calls to `Put` where `x > 100`. Instead, one would have to add a test to the body of the join which partially defeats the declarative nature of the synchronization. In section 3, we show how these drawbacks can be eliminated by integrating join patterns with a host language's standard support for pattern matching.

Joins for Actors While join patterns have been successfully used to synchronize threads, to the best of our knowledge, they have not yet been applied in the context of an actor-based concurrency model. In Scala, actor-based concurrency is supported by means of a library [10]. Because we provide join patterns as a library extension as well, we have created the opportunity to combine join patterns with the event-driven concurrency model offered by actors. We give a detailed explanation of this combination in section 3.2. However, in order to understand this integration, we first briefly highlight how to write concurrent programs using Scala's actor library.

Scala's actor library is largely inspired by Erlang's model of concurrent processes communicating by message-passing [1]. New actors are defined as classes inheriting the `Actor` class. The actor's life cycle is described by its `act` method. The following code shows how to implement the unbounded buffer as an actor:

```
class Buffer extends Actor {
  def act() { loop(Nil) }
  def loop(buf: List[Int]) {
    receive {
      case Put(x) => loop(buf ::: List(x)) // append x to buf
      case Get() if !buf.isEmpty => reply(buf.head); loop(buf.tail) }
  }
}
```

The `receive` method allows an actor to selectively wait for certain messages to arrive in its mailbox. The actor processes at most one message at a time. Messages that are sent concurrently to the actor are queued in its mailbox. Interacting with a buffer actor occurs as follows:

```
val buffer = new Buffer; buffer.start()
buffer ! Put(42) // asynchronous send, returns nothing
println(buffer !? Get()) // synchronous send, waits for reply
```

Synchronous message sends make the sending process wait for the actor to reply to the message (by means of `reply(value)`). Scala actors also offer more advanced synchronization patterns such as futures [11,21]. `actor !! msg` denotes an asynchronous send that immediately returns a future object. In Scala, a future is a nullary function that, when applied, returns the future's computed result value. If the future is applied before the value is computed, the caller is blocked.

In the above example, the required synchronization between `Put` and `Get` is achieved by means of a *guard*. The guard in the `Get` case disallows the processing of any `Get` message while the `buf` queue is empty. In the implementation, all cases are sequentially checked against the incoming message. If no case matches, or all of the guards for matching cases evaluate to false, the actor keeps the message stored in its mailbox and awaits other messages.

Even though the above example remains simple enough to implement, the synchronization between `Put` and `Get` remains very implicit. The actual *intention* of the programmer, i.e. the fact that an item can only be produced when the actor received both a `Get` *and* a `Put` message, remains implicit in the code. Hence, even actors can benefit from the added declarative synchronization of join patterns, as we will illustrate in section 3.2.

3 A Scala Joins Library

We now discuss a Scala library (henceforth called Scala Joins) providing join patterns implemented via extensible pattern matching. First, we explain how Scala Joins enables the declarative synchronization of threads, postponing joins for actors until section 3.2.

3.1 Joining Threads

Scala Joins draws on Scala's extensible pattern matching facility [5]. This has several advantages: first of all, the programmer may use Scala's rich pattern syntax to express constraints using nested patterns and guards. Moreover, reusing the existing variable binding mechanism avoids typical problems of other library-based approaches where the order in which arguments are passed to the function implementing the join body is merely conventional, as explained in section 2. Similar to C# Joins's channels, joins in Scala Joins are composed of synchronous and asynchronous *events*. Events are strongly typed and can be invoked using standard method invocation syntax. The FIFO buffer example is written in Scala Joins as follows:

```
class Buffer extends Joins {
  val Put = new AsyncEvent[Int]
  val Get = new SyncEvent[Int]
  join { case Get() & Put(x) => Get reply x }
}
```

To enable join patterns, a class inherits from the `Joins` class. Events are declared as regular fields. They are distinguished based on their (a)synchrony and the number of arguments they take. For example, `Put` is an asynchronous event that takes a single argument of type `Int`. Since it is asynchronous, no return type is specified (it immediately returns `unit` when invoked). In the case of a synchronous event such as `Get`, the first type parameter specifies the return type. Therefore, `Get` is a synchronous event that takes no arguments and returns values of type `Int`.

Joins are declared using the `join { ... }` construct. This construct enables pattern matching via a list of `case` declarations that each consist of a left-hand side and a right-hand side, separated by `=>`. The left-hand side defines a join pattern through the juxtaposition of a linear combination of asynchronous and synchronous events. As is common in the joins literature, we use `&` as the juxtaposition operator. Arguments of events are usually specified as variable patterns. For example, the variable pattern `x` in the `Put` event can bind to any value (of type `Int`). This means that on the right-hand side, `x` is bound to the argument of the `Put` event when the join pattern matches. Standard pattern matching can be used to constrain the match even further (an example of this is given below).

The right-hand side of a join pattern defines the join body (an ordinary block of code) that is executed when the join pattern matches. Like JoCaml, but unlike Cω and C# Joins, Scala Joins allows any number of synchronous events to appear in a join pattern. Because of this, it is impossible to use the return value of the body to implicitly reply to the single synchronous event in the join pattern. Instead, the body of a join pattern explicitly replies to all of the synchronous events that are part of the join pattern on the left-hand side. Synchronous events are replied to by invoking their `reply` method. This wakes up the thread that originally signalled that event.

To demonstrate how join patterns can be combined with ordinary pattern matching, consider the traditional problem of synchronizing multiple concurrent readers with one or more writers who need exclusive access to a resource. A multiple reader/one writer lock can be implemented in our library as follows:[4]

[4] This implementation is based on that of Cω [2] and Russo's Joins library for C# [15].

```
class ReaderWriterLock extends Joins {
  private val Sharing = new AsyncEvent[Int]
  val Exclusive, ReleaseExclusive = new NullarySyncEvent
  val Shared, ReleaseShared = new NullarySyncEvent
  join {
    case Exclusive() & Sharing(0) => Exclusive reply
    case ReleaseExclusive() => { Sharing(0); ReleaseExclusive reply }
    case Shared() & Sharing(n) => { Sharing(n+1); Shared reply }
    case ReleaseShared() & Sharing(1) => { Sharing(0); ReleaseShared reply }
    case ReleaseShared() & Sharing(n) => { Sharing(n-1); ReleaseShared reply }
  }
  Sharing(0) }
```

In the above example, events are used to encode the state of the reader-writer lock. The last statement ensures that the lock starts off in an idle state (no thread is sharing the lock). A writer can signal a synchronous `Exclusive` event to acquire the lock. Concurrent readers are represented by means of a `Sharing(n)` event which encodes the number of currently active readers.

In the join pattern `Exclusive() & Sharing(0)`, regular pattern matching is used to constrain the pattern only to `Sharing` events whose argument equals `0`, thus ensuring that this pattern only triggers when no other thread is sharing the lock. Similarly, the join pattern `ReleaseShared() & Sharing(1)` only triggers when the *last* reader releases the lock. If join patterns would not be integrated with pattern matching, code like this would require additional tests in the body of more general join patterns.

3.2 Joining Actors

We now describe an integration of our joins library with Scala's actor library [10]. The following example shows how to re-implement the unbounded buffer example using Joins:

```
val Put = new Join1[Int]
val Get = new Join
class Buffer extends JoinActor {
  def act() {
    receive { case Get() & Put(x) => Get reply x }
  } }
```

It differs from the thread-based bounded buffer using joins in the following ways:

- The `Buffer` class inherits the `JoinActor` class to declare itself to be an actor capable of processing join patterns.
- Rather than defining `Put` and `Get` as synchronous or asynchronous *events*, they are all defined as *join messages* which may support both kinds of synchrony (this is explained in more detail below).
- The `Buffer` actor defines `act` and awaits incoming messages by means of `receive`. Note that it is still possible for the actor to serve regular messages within the `receive` block. Logically, regular messages can be regarded as unary join patterns. However,

they don't have to be declared as joinable messages; in fact, our joins extension is fully source and binary compatible with the existing actor library.

We illustrate below how the buffer actor can be used as a coordinator between a consumer and a producer actor. The producer sends an asynchronous `Put` message while the consumer awaits the reply to a `Get` message by invoking it synchronously (using `!?`).[5]

```
val buffer = new Buffer; buffer.start()
val prod = actor { buffer ! Put(42) }
val cons = actor { (buffer !? Get()) match { case x:Int => /* process x */ } }
```

By applying joins to actors, the synchronization dependencies between `Get` and `Put` can be specified declaratively by the buffer actor. The actor will receive `Get` and `Put` messages by queuing them in its mailbox. Only when all of the messages specified in the join pattern have been received is the body executed by the actor. Before processing the body, the actor atomically removes all of the participating messages from its mailbox. Replies may be sent to any or all of the messages participating in the join pattern. This is similar to the way replies are sent to events in the thread-based joins library described previously.

Contrary to the way events are defined in the thread-based joins library, an actor does not explicitly define a join message to be synchronous or asynchronous. We say that join messages are "synchronization-agnostic" because they can be used in different synchronization modes between the sender and receiver actors. However, when they are used in a particular join pattern, the sender and receiver actors have to agree upon a valid synchronization mode. In the previous example, the `Put` join message was sent asynchronously, while the `Get` join message was sent synchronously. In the body of a join pattern, the receiver actor replied to `Get`, but not to `Put`.

The advantage of making join messages synchronization agnostic is that they can be used in arbitrary synchronization modes, including more advanced synchronization modes such as ABCL's future-type message sending [21] or Salsa's token-passing continuations [18]. Every join message instance has an associated *reply destination*, which is an output channel on which processes may listen for possible replies to the message. How the reply to a message is processed is determined by the way the message was sent. For example, if the message was sent purely asynchronously, the reply is discarded; if it was sent synchronously, the reply awakes the sender. If it was sent using a future-type message send, the reply resolves the future.

4 Integrating Joins and Extensible Pattern Matching

Our implementation technique for joins is unique in the way events interact with an extensible pattern matching mechanism. We explain the technique using a concrete implementation in Scala. However, we expect that implementations based on, e.g., the active patterns of F# [17] would not be much different. In the following we first talk about pattern matching in Scala. After that we dive into the implementation of events which crucially depends on properties of Scala's extensible pattern matching. Finally, we highlight how joins have been integrated into Scala's actor framework.

[5] Note that the `Get` message has return type `Any`. The type of the argument values is recovered by pattern matching on the result, as shown in the example.

Partial Functions In the previous section we used the `join { ... }` construct to declare a set of join patterns. It has the following form:

```
join {
  case pat_1 => body_1
  ...
  case pat_n => body_n
}
```

The patterns pat_i consist of a linear combination of events evt_1 `&` ... `&` evt_m. Threads synchronize over a join pattern by invoking one or several of the events listed in a pattern pat_i. When all events occurring in pat_i have been invoked, the join pattern matches, and its corresponding join $body_i$ is executed.

In Scala, the pattern matching expression inside braces is treated as a first-class value that is passed as an argument to the `join` function. The argument's type is an instance of `PartialFunction`, which is a subclass of `Function1`, the class of unary functions. The two classes are defined as follows.

```
abstract class Function1[A, B] {
  def apply(x: A): B }
abstract class PartialFunction[A, B] extends Function1[A, B] {
  def isDefinedAt(x: A): Boolean }
```

Functions are objects which have an `apply` method. Partial functions are objects which have in addition a method `isDefinedAt` which tests whether a function is defined for a given argument. Both classes are parameterized; the first type parameter `A` indicates the function's argument type and the second type parameter `B` indicates its result type.

A pattern matching expression `{ case` p_1 `=>` e_1`; ...; case` p_n `=>` e_n `}` is then a partial function whose methods are defined as follows.

- The `isDefinedAt` method returns `true` if one of the patterns p_i matches the argument, `false` otherwise.
- The `apply` method returns the value e_i for the first pattern p_i that matches its argument. If none of the patterns match, a `MatchError` exception is thrown.

Join patterns as partial functions. Whenever a thread invokes an event `e`, each join pattern in which `e` occurs has to be checked for a potential match. Therefore, events have to be associated with the set of join patterns in which they participate. As shown before, this set of join patterns is represented as a partial function. Invoking `join(pats)` associates each event occurring in the set of join patterns with `pats`.

When a thread invokes an event, the `isDefinedAt` method of `pats` is used to check whether any of the associated join patterns match. If yes, the corresponding join body is executed by invoking the `apply` method of `pats`. A question remains: what argument is passed to `isDefinedAt` and `apply`, respectively? To answer this question, consider the simple buffer example from the previous section. It declares the following join pattern:

```
join { case Get() & Put(x) => Get reply x }
```

Assume that no events have been invoked before, and a thread t invokes the `Get` event to remove an element from the buffer. Clearly, the join pattern does not match, which

causes t to block since `Get` is a synchronous event (more on synchronous events later). Assume that after thread t has gone to sleep, another thread s adds an element to the buffer by invoking the `Put` event. Now, we want the join pattern to match since both events have been invoked. However, the result of the matching does not only depend on the event that was last invoked but also on the fact that *other events* have been invoked previously. Therefore, it is *not* sufficient to simply pass a `Put` message to the `isDefinedAt` method of the partial function the represents the join patterns. Instead, when the `Put` event is invoked, the `Get` event has to somehow "pretend" to also match, even though it has nothing to do with the current event. While previous invocations can simply be buffered inside the events, it is non-trivial to make the pattern matcher actually consult this information during the matching, and "customize" the matching results based on this information. To achieve this customization we use extensible pattern matching.

Extensible Pattern Matching Emir et al. [5] recently introduced *extractors* for Scala that provide representation independence for objects used in patterns. Extractors play a role similar to *views* in functional programming languages [20,13] in that they allow conversions from one data type to another to be applied implicitly during pattern matching. As a simple example, consider the following object that can be used to match even numbers:

```
object Twice {
  def apply(x: Int) = x*2
  def unapply(z: Int) = if (z%2 == 0) Some(z/2) else None }
```

Objects with `apply` methods are uniformly treated as functions in Scala. When the function invocation syntax `Twice(x)` is used, Scala implicitly calls `Twice.apply(x)`. The `unapply` method in `Twice` reverses the construction in a pattern match. It tests its integer argument `z`. If `z` is even, it returns `Some(z/2)`. If it is odd, it returns `None`. The `Twice` object can be used in a pattern match as follows:

```
val x = Twice(21)
x match {
  case Twice(y) => println(x+" is two times "+y)
  case _ => println("x is odd") }
```

To see where the `unapply` method comes into play, consider the match against `Twice(y)`. First, the value to be matched (`x` in the above example) is passed as argument to the `unapply` method of `Twice`. This results in an optional value which is matched subsequently.[6] The preceding example is expanded as follows:

```
val x = Twice.apply(21)
Twice.unapply(x) match {
  case Some(y) => println(x+" is two times "+y)
  case None => println("x is odd") }
```

[6] The optional value is of parameterized type `Option[T]` that has the two subclasses `Some[T](x: T)` and `None`.

Extractor patterns with more than one argument correspond to `unapply` methods returning an optional tuple. Nullary extractor patterns correspond to `unapply` methods returning a Boolean.

In the following we show how extractors can be used to implement the matching semantics of join patterns. In essence, we define appropriate `unapply` methods for events which get implicitly called during the matching.

Matching Join Patterns As shown previously, a set of join patterns is represented as a partial function. Its `isDefinedAt` method is used to find out whether one of the join patterns matches. In the following we are going to explain the code that the Scala compiler produces for the body of this method. Let us revisit the join pattern that we have seen in the previous section:

```
Get() & Put(x)
```

In our library, the `&` operator is an extractor (see previous section) that defines an `unapply` method; therefore, the Scala compiler produces the following matching code:

```
&.unapply(m) match {
  case Some((u, v)) =>
    u match {
      case Get() => v match {
        case Put(x) => true
        case _ => false }
      case _ => false }
  case None => false }
```

We defer a discussion of the argument m that is passed to the `&` operator. For now, it is important to understand the general scheme of the matching process. Basically, calling the `unapply` method of the `&` operator produces a pair of intermediate results wrapped in `Some`. Standard pattern matching decomposes this pair into the variables `u` and `v`. These variables, in turn, are matched against the events `Get` and `Put`. Only if both of them match, the overall pattern matches.

Since the `&` operator is left-associative, matching more than two events proceeds by first calling the unapply methods of all the `&` operators from right to left, and then matching the intermediate results with the corresponding events from left to right.

Since events are objects that have an `unapply` method, we can expand the code further:

```
&.unapply(m) match {
  case Some((u, v)) =>
    Get.unapply(u) match {
      case true => Put.unapply(v) match {
        case Some(x) => true
        case None => false }
      case false => false }
  case None => false }
```

As we can see, the intermediate results produced by the `unapply` method of the `&` operator are passed as arguments to the `unapply` methods of the corresponding events. Since the `Get` event is parameterless, its `unapply` method returns a Boolean, telling whether it matches or not. The `Put` event, on the other hand, takes a parameter; when the pattern matches, this parameter gets bound to a concrete value that is produced by the `unapply` method.

The `unapply` method of a parameterless event such as `Get` essentially checks whether it has been invoked previously. The `unapply` method of an event that takes parameters such as `Put` returns the argument of a previous invocation (wrapped in `Some`), or signals failure if there is no previous invocation. In both cases, previous invocations have to be buffered inside the event.

Firing join patterns. As mentioned before, executing the right-hand side of a pattern that is part of a partial function amounts to invoking the `apply` method of that partial function. Basically, this repeats the matching process, thereby binding any pattern variables to concrete values in the pattern body. When firing a join pattern, the events' `unapply` methods have to dequeue the corresponding invocations from their buffers. In contrast, invoking `isDefinedAt` does not have any effect on the state of the invocation buffers. To signal to the events in which context their `unapply` methods are invoked, we therefore need some way to propagate out-of-band information through the matching. For this, we use the argument m that is passed to the `isDefinedAt` and `apply` methods of the partial function. The `&` operator propagates this information verbatim to its two children (its `unapply` method receives m as argument and produces a pair with two copies of m wrapped in `Some`). Eventually, this information is passed to the events' `unapply` methods.

4.1 Implementation Details

Events are represented as classes that contain queues to buffer invocations. The `Event` class is the super class of all synchronous and asynchronous events:[7]

```
abstract class Event[R, Arg](owner: Joins) {
  val tag = owner.freshTag
  val argQ = new Queue[Arg]
  def apply(arg: Arg): R = synchronized { argQ += arg; invoke() }
  def invoke(): R
  def unapply(isDryRun: Boolean): Option[Arg] =
    if (isDryRun && !argQ.isEmpty)
      Some(argQ.front)
    else if (!isDryRun && owner.matches(tag))
      Some(argQ.dequeue())
    else None
}
```

[7] In our actual implementation the fact whether an event is parameterless is factored out for efficiency. Due to lack of space, we show a simplified class hierarchy.

The Event class takes two type arguments R and Arg that indicate the result type and parameter type of event invocations, respectively. Events have a unique owner which is an instance of the Joins class. This class provides the join method that we used in the buffer example to declare a set of join patterns. An event can appear in several join patterns declared by its owner. The tag field holds an identifier which is unique with respect to a given owner instance. Whenever the event is invoked via its apply method, we append the provided argument to the argQ. The abstract invoke method is used to run synchronization-specific code; synchronous and asynchronous events differ mainly in their implementation of the invoke method (we show a concrete implementation for synchronous events below). In the unapply method we test whether matching occurs during a dry run. If it does not, we ask the owner whether the event belongs to a matching join pattern (owner.matches(tag)) in which case an event invocation is dequeued.

Synchronous events are implemented as follows:

```
abstract class SyncEvent[R, Arg] extends Event[R, Arg] {
  val waitQ = new Queue[SyncVar[R]]
  def invoke(): R = { val res = new SyncVar[R]
    waitQ += res; owner.matchAndRun(); res.get }
  def reply(res: R) = waitQ.dequeue().set(res)
}
```

Synchronous events contain a logical queue of waiting threads, waitQ, which is implemented using the implicit wait set of synchronous variables.[8] The invoke method is run whenever the event is invoked. It creates a new SyncVar and appends it to the waitQ. Then, the owner's matchAndRun method is invoked to check whether the event invocation triggers a complete join pattern. After that, the current thread waits for the SyncVar to become initialized by accessing it. If the owner detects (during owner.matchAndRun()) that a join pattern triggers, it will apply the join, thereby re-executing the pattern match (binding variables etc.) and running the join body. Inside the body, synchronous events are replied to by invoking their reply method. Replying means dequeuing a SyncVar and setting its value to the supplied argument. If none of the join patterns matches, the thread that invoked the synchronous event is blocked (upon calling res.get) until another thread triggers a join pattern that contains the same synchronous event.

Thread-safety. Our implementation avoids races when multiple threads try to match a join pattern at the same time; checking whether a join pattern matches (and, if so, running its body) is an atomic operation. Notably, the isDefinedAt/apply methods of the join set are only called from within the synchronized matchAndRun method of the Joins class. The unapply methods of events, in turn, are only called from within the matching code inside the partial function, and are thus guarded by the same lock. The internal state of individual events is updated consistently: the apply method is atomic, and the reply method is called only from within join bodies which are guarded by the owner's lock. We don't assume any concurrency properties of the argQ and waitQ queues.

[8] A SyncVar is an atomically updatable reference cell; it blocks threads trying to access an uninitialized cell.

4.2 Implementation of Actor-based Joins

Actor-based joins integrate with Scala's pattern matching in essentially the same way as the thread-based joins, making both implementations very similar. We highlight how joins are integrated into the actor library, and how reply destinations are supported.

In the Scala actors library, `receive` is a method that takes a `PartialFunction` as a sole argument, similar to the `join` method defined previously. To make `receive` aware of join patterns, the abstract `JoinActor` class overrides these methods by wrapping the partial function into a specialized partial function that understands join messages. `JoinActor` also overrides `send` to set the reply destination of a join message. Message sends such as `a!msg` are interpreted as calls to `a`'s `send` method.

```
abstract class JoinActor extends Actor {
  override def receive[R](f: PartialFunction[Any, R]): R =
    super.receive(new JoinPatterns(f))
  override def send(msg: Any, replyTo: OutputChannel[Any]) {
    setReplyDest(msg, replyTo)
    super.send(msg, replyTo) }
  def setReplyDest(msg: Any, replyTo: OutputChannel[Any]) {...} }
```

`JoinPatterns` is a special partial function that detects whether its argument message is a join message. If it is, then the argument message is transformed to include out-of-band information that will be passed to the pattern matcher, as is the case for events in the thread-based joins library. The boolean argument passed to the `asJoinMessage` method indicates to the pattern matcher whether or not join message arguments should be dequeued upon successful pattern matching. If the `msg` argument is not a join message, `asJoinMessage` passes the original message to the pattern matcher unchanged, enabling regular actor messages to be processed as normal.

```
class JoinPatterns[R](f: PartialFunction[Any, R])
  extends PartialFunction[Any, R] {
  def asJoinMessage(msg: Any, isDryRun: Boolean): Any =
    ...
  override def isDefinedAt(msg: Any) =
    f.isDefinedAt(asJoinMessage(msg, true))
  override def apply(msg: Any) =
    f(asJoinMessage(msg, false))
}
```

Recall from the implementation of synchronous events that thread-based joins used constructs such as `SyncVars` to synchronize the sender of an event with the receiver. Actor-based joins do not use such constructs. In order to synchronize sender and receiver, every join message has a reply destination (which is an `OutputChannel`, set when the message is sent in the actor's `send` method) on which a sender may listen for replies. The `reply` method of a `JoinMessage` simply forwards its argument value to this encapsulated reply destination. This wakes up an actor that performed a synchronous send (`a!?msg`) or that was waiting on a future (`a!!msg`).

5 Discussion and Related Work

Benton et al. [2] note that supporting general guards in join patterns is difficult to implement efficiently as it requires testing all possible combinations of queued messages to find a match. Side effects pose another problem. Benton et al. suggest a restricted language for guards to overcome these issues. However, to the best of our knowledge, there is currently no joins framework that supports a sufficiently restrictive yet expressive guard language to implement efficient guarded joins. Our current implementation does not handle general guards, although they are permitted in Scala's pattern syntax [6]. We find that guards often help at the interface to a component that uses private messages to represent values satisfying a guard, as in the following example:

```
join { case Put(x) if (x > 0) => this.PositivePut(x)
       case PositivePut(x) & Get() => Get reply x }
```

Cω [2] is a language extension of C# supporting *chords*, linear combinations of methods. In contrast to Scala Joins, Cω allows at most one synchronous method in a chord. The thread invoking this method is the thread that eventually executes the chord's body. The benefits of Cω as a language extension over Scala Joins are that chords can be enforced to be well-formed and that their matching code can be optimized ahead of time. In Scala Joins, the joins are only analyzed at pattern-matching time. The benefit of Scala Joins as a library extension is that it provides more flexibility, such as multiple synchronous events. Russo's `Joins` library [15] exploits the expressiveness of C# 2.0's generics to implement Cω's synchronization constructs. Piggy-backing on an existing variable binding mechanism allows us to avoid problems with `Joins`' delegates where the order in which arguments are passed is merely conventional. Scala's implicit arguments also help to alleviate some of the initialization boilerplate. CCR [3] is a C# library for asynchronous concurrency that supports join patterns without synchronous components. Join bodies are scheduled for execution in a thread pool. Our library integrates with JVM threads using synchronous variables, and supports event-based programming through its integration with Scala Actors. Singh [16] shows how a small set of higher-order combinators based on Haskell's software transactional memory (STM) can encode expressive join patterns. Salsa [18] is a language extension of Java supporting actors. In Salsa, actors may synchronize upon the arrival of multiple messages by means of a *join continuation*. However, join continuations only allow an actor to synchronize on gathering replies to previously sent messages. Using joins, Scala actors may synchronize on any incoming message. CML [14] allows threads to synchronize on first-class composable events; because all events have a single commit point, certain protocols may not be specified in a modular way (for example when an event occurs in several join patterns). By combining CML's events with all-or-nothing transactions, transactional events [4] overcome this restriction but may have a higher overhead than join patterns.

6 Conclusion

We presented a novel implementation of join patterns based on extensible pattern matching constructs of languages such as Scala and F#. The embedding into general pattern

matching provides expressive features such as nested patterns and guards for free. The resulting programs are often as concise as if written in more specialized language extensions. We implemented our approach as a Scala library that supports Ada-style rendezvous and constraints and furthermore integrated it with the Scala Actors event-based concurrency framework without changing the syntax and semantics of existing programs.

References

1. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.
2. Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst*, 26(5):769–804, 2004.
3. Georgio Chrysanthakopoulos and Satnam Singh. An asynchronous messaging library for C#. In *Proc. SCOOL Workshop, OOPSLA*, 2005.
4. Kevin Donnelly and Matthew Fluet. Transactional events. In *Proc. ICFP*, pages 124–135. ACM, 2006.
5. Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In Erik Ernst, editor, *Proc. ECOOP*, volume 4609 of *LNCS*, pages 273–298. Springer, 2007.
6. Martin Odersky et al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.
7. Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. JoCaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, volume 2638 of *LNCS*, pages 129–158. Springer, 2002.
8. Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. POPL*, pages 372–385. ACM, January 1996.
9. Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A Calculus of Mobile Agents. In *CONCUR*, pages 406–421. Springer-Verlag, August 1996.
10. Philipp Haller and Martin Odersky. Actors that unify threads and events. In *Proc. COORDINATION*, LNCS. Springer, June 2007.
11. Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
12. Martin Odersky. Functional Nets. In *European Symposium on Programming 2000*, Lecture Notes in Computer Science. Springer Verlag, 2000.
13. C. Okasaki. Views for Standard ML, 1998.
14. J. H. Reppy. CML: A higher-order concurrent language. In *Proc. PLDI*, pages 294–305, Toronto, Ontario, Canada, June 1991. ACM Press, New York.
15. Claudio V. Russo. The Joins concurrency library. In *PADL*, pages 260–274, 2007.
16. Satnam Singh. Higher-order combinators for join patterns using STM. In *Proc. TRANSACT Workshop, OOPSLA*, 2006.
17. Don Syme, Gregory Neverov, and James Margetson. Extensible Pattern Matching via a Lightweight Language Extension. In *Proc. ICFP*, 2007.
18. Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.
19. G.S. von Itzstein and David Kearney. Join Java: An alternative concurrency semantic for Java. Technical Report ACRC-01-001, University of South Australia, 2001.
20. Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *POPL*, pages 307–313, 1987.
21. Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proc. OOPSLA*, pages 258–268, 1986.

Executing Action Languages for Planning Problems on Multi-core Platforms: Some Preliminary Results

To Thanh Son, Phan Huy Tu, Enrico Pontelli, Tran Cao Son

New Mexico State University
Department of Computer Science
{sto,tphan,epontell,tson}@cs.nmsu.edu

Abstract. The goal of this paper is to demonstrate that parallel programming techniques can boost AI planning systems in various aspects. It shows that an appropriate parallelization of a sequential planning system often brings gain in performance and/or scalability. We start by describing general schemes for parallelizing the construction of a plan. We then discuss the applications of these techniques to two domain-independent heuristic search based planners—a competitive conformant planner (CPA) and a state-of-the-art classical planner (FF). We present experimental results which show that performance improvements and scalability are obtained in both cases. Finally, we discuss the issues that should be taken into consideration when designing a parallel planning system and relate our work to the existing literature.

1 Introduction and Motivation

Planning is the problem of finding a sequence of actions that changes the state of the world from an initial state to a state that satisfies a given set of requirements. Planning is computationally hard. Even the problem of searching for a polynomially bounded plan for propositional domains is NP-complete [3], and it becomes computationally even harder (Σ_P^2-complete) when the initial state is incomplete [3]. These complexity results suggest that domain-independent planners are likely to fail to build a plan within acceptable time bounds for certain problem instances.

In spite of these limitations, over the years, we have witnessed a continuous interest by researchers in building domain-independent planners. The challenge has led to the development of more creative ways to attack this problem, such as the development of new data structure (e.g., the planning graph [4]), the development of several domain-independent heuristics (e.g., [5, 15]), and the development of new planning techniques (e.g., SAT-based or model checking based planners [18, 9], Answer Set planning [20]).

Conformant planning is the problem of finding a sequence of actions that changes the state of the world from *every* possible initial state (or equivalently, a set of initial states) to a state that satisfies a given set of requirements. Like classical planning—which deals with planning problems in presence of a *complete* initial state—conformant planning can be viewed as a *search* problem. Several approaches to conformant planning have been developed. Graphplan is extended in [23] to deal with incompleteness of the initial state. Satisfiability and model-based planning have been applied to conformant planning in [9, 8]. Conformant answer set planning is discussed in [13].

Planning as *heuristic search* has played an important role in planning research. Indeed, heuristic search planners are among the best domain-independent planners[1] for *classical domains* (e.g., FF, HSP2, AltAlt, etc. [1]). For domains with *incomplete* information, heuristic search planners or *conformant planners* are also among the fastest developed (e.g., [6, 7]). The main difference between classical planning and conformant planning lies in the size of the search space. The former searches for solutions in the *state-space*—which can be exponential in the number of propositions in the problem—while the latter performs the search in the *belief-state* space—which is double exponential in the number of propositions.

It is apparent that one of the keys to the success of domain-independent search based planners is the design of good heuristics and a better and compact representation of the search space (e.g., [7]). It is known, however, that there is a trade-off between the cost of computing a heuristic function and its performance. In this context, it is interesting to observe that a domain-independent conformant planner, called CPA [24], can be competitive with many state-of-the-art conformant planners, even though it uses a rather simple heuristic to guide its search. Instead of employing a sophisticated heuristic, CPA uses *approximation* (more on CPA in the next section). The performance of CPA demonstrates that, for various classes of planning problems, the performance of a heuristic function can be compensated by *changes in the reasoning mechanisms*.

In this paper, we continue along the same line of thought, proposing another mechanism, orthogonal to the development of new heuristics, to enhance performance of heuristic search-based planners. More precisely, we investigate the use of *parallel machines* to extract concurrency from the reasoning process employed in planning. Our approach is motivated by: *(a)* the demand for solving larger planning instances; *(b)* the architectural trends of providing users with affordable multi-core platforms; *(c)* the availability of affordable components to build Beowulf clusters; and *(d)* the observation that, with few exceptions (e.g., [26]), existing planning systems are tailored to *sequential computing platforms*.

These issues lead to the following research questions: *(1) Can the computing power of parallel platforms be used to improve planning efficiency and to solve larger problems, which cannot be solved by current planners?* and *(2) Which parallel computing techniques can be applied to planning?* In this paper, we answer these questions.

Our study offers the following outcomes: *(1)* Parallel machines can be effectively used in planning, to solve larger problems and speed up planning time; *(2)* Parallel search techniques [14] cannot be applied blindly; *(3)* Parallel machines can effectively compensate the informativeness of the heuristic function; *(4)* State-of-the-art planners can be adapted to take full advantage of the added computational power provided by multi-core and distributed platforms.

Let us underline that an extensive literature exists dealing with the development of parallel solutions to search problems in various domains (e.g., constraint solving, logic programming, SAT, combinatorial optimization [14, 22, 28, 19]). Although these works provide a basic backbone to our effort, they also clearly highlight that a generic solution

[1] SAT-based planners are becoming more competitive (`zeus.ing.unibs.it/ipc-5`), due to efficient SAT-solvers, but we believe heuristic-based planners will remain important for many years.

to parallel search does not exist, and solutions developed in other domains need to be properly modified to be effective in the context of planning. Thus, it is important to know what will be the pros and cons of parallelization of a planning system and what are the issues that need to be investigated. We hope this paper will provide directions in answering such questions.

2 Preliminaries

2.1 Action Representation and Planning as Search

We employ the action language $\mathcal{AL}$ in [2] for action representation, and we represent a planning problem instance as $\mathcal{P} = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{D}$ (the *planning domain*) encodes the actions with their effects and preconditions, and $\mathcal{I}$ and $\mathcal{G}$ describe the *initial state* and the *goal*. $\mathcal{I}$ can be represented by a set of states $\Sigma_{\mathcal{I}}$, and it is said to be *complete* if $|\Sigma_{\mathcal{I}}| = 1$. An action expressed in the *Planning Domain Definition Language (PDDL)*

```
(:action action_name
 :parameters (list of ?xi - typei)
 :precondition (condition)
 :effect (and (when cond1 => eff1) ...))
```

can be viewed as a set of ground laws in $\mathcal{AL}$ as follows. For each valid vector $\boldsymbol{x}$ of the parameters of `action_name`, we have the set consisting of an executability condition

executable `action_name`$(\boldsymbol{x})$ **if** `condition`

and a set of action effect rules

`action_name`$(\boldsymbol{x})$ **causes** `effi` **if** `condi`

for $i = 1, \ldots, k$. The main difference between $\mathcal{AL}$ and PDDL is that $\mathcal{AL}$ allows for the representation of arbitrary axioms. The usefulness of axioms in planning has been discussed in [25].

The semantics of a planning domain can be defined by a *state-transition system* $(\mathcal{S}, \mathcal{A}, \Phi)$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, and Φ is a mapping from a pair of the form *(action, state)* to a set of states. For an action a and a state s, $\Phi(a, s)$ denotes the set of possible states resulting from the execution of action a in the state s. When a is not executable in s (or its preconditions are not satisfied in s), $\Phi(a, s) = \emptyset$. An action is *deterministic* if $|\Phi(a, s)| \leq 1$.

For a set of states S, also called a *belief state* (or b-state for short), and an action a, we say that a is executable in S if a is executable in every $s \in S$, and we write $\Phi(a, S) = \bigcup_{s \in S} \Phi(a, s)$; otherwise, $\Phi(a, S) = \emptyset$ (a is not executable in S). Φ is also extended to $\widehat{\Phi}$, which maps action sequences and b-states to b-states. $\widehat{\Phi}(\alpha, S)$ is defined as $\widehat{\Phi}([], S) = S$; and $\widehat{\Phi}([a_0, \ldots, a_n], S) = \widehat{\Phi}([a_1, \ldots, a_n], \Phi(a_0, S))$. An action sequence α is a *solution* to $\mathcal{P}$ (a *plan*) if $\widehat{\Phi}(\alpha, \Sigma_{\mathcal{I}}) \neq \emptyset$ and for every $s \in \widehat{\Phi}(\alpha, \Sigma_{\mathcal{I}})$, s satisfies $\mathcal{G}$.

Here, we experiment with planning problems where: (a) actions are deterministic; (b) the domains might or might not contain axioms; and (c) the initial state might be incomplete.

Figure 1 shows a generic algorithm for planning as search, where $\Sigma_{\mathcal{I}}$ is the initial belief state and $\mathcal{G}$ is the goal (we assume that $\Sigma_{\mathcal{I}}$ does not satisfy $\mathcal{G}$).

Algorithm 1: FWDPLAN($\mathcal{D}, \mathcal{I}, \mathcal{G}$)

```
1.  S = Σ_I ; Queue = {(S, [])} ; Visited = {S}
2.  while Queue is not empty
3.      select (S, p) with the best
          heuristic value from Queue
4.      for each action a executable in S
5.        S' = Φ(a, S)
6.        if S' satisfies G then return [p; a]
7.        else if S' ∉ Visited
8.          compute heuristic for S'
9.          insert (S', [p; a]) into Queue
10.         insert S' into Visited
```

Fig. 1. A heuristic forward search algorithm

2.2 Two Sequential Planning Systems

We use two sequential planning systems in our experiments. The two systems have been chosen because of their efficiency, the fact that they are both search-based planners, and the availability of their source code.

The first system, *FF*,[2] is a planner for classical domains, where the initial state is complete. FF is one of the state-of-the-art classical planers [16], and has received several awards for its outstanding performance in international planning competitions. The input of FF is PDDL. This version of FF does not consider axioms. In FF, the next state is computed by the equation $\Phi(a, s) = s \cup e_a^+(s) \setminus e_a^-(s)$ where $e_a^+(s)$ (resp. $e_a^-(s)$) is the set of positive effects (resp. the set of negative effects) of a in the state s. For this reason, the computation of $\Phi(a, s)$ is very fast as it can be done by two set operations. FF also modifies the general algorithm of Fig. 1 by adding an initial phase of local search (based on hill-climbing), and entering the best-first search only upon failure of the local search phase. The outstanding performance of FF *can be attributed to the accuracy of its heuristic*.

The second system used in our study is a modification of the CPA system[3] [24], developed for conformant planning problems (i.e., $|\Sigma_{\mathcal{I}}| \geq 1$). CPA uses $\mathcal{AL}$ as its input language and the number of fulfilled subgoals as heuristic measure (which is known for being not very accurate). CPA cannot match the performance of FF in most of the classical domains. Nevertheless, CPA is competitive with most of the state-of-the-art conformant planners (at the time it was developed). The main difference between CPA and other conformant planners is that it considers axioms directly, and it uses Φ^*, a deterministic approximation of Φ, to deal with non-determinism (caused by axioms) and incompleteness. Informally, given an action a and a set of literals δ approximating the state of the world, the next state of the world $\Phi^*(a, \delta)$ is approximated by a fixpoint of the process that computes *(i)* the set of fluents that cannot possibly be changed or hold δ_i; and *(ii)* the set of direct effects of a in δ_i where $\delta_0 = \delta$. This process might require n^2 steps, where n is the number of propositions in the domain. Detailed definitions of $\Phi^*(a, \delta)$ can be found in [24]. As we will see later, this will be an important source of parallelism.

[2] See `members.deri.at/~joergh/ff.html`.
[3] See `www.cs.nmsu.edu/~tphan/software.htm`

3 The Parallel System — General Schemes

In the rest of the discussion we will use the generic term *computing agents* (or, simply, *agents*) to indicate the concurrent planning engines, concretely implemented as threads or processes. The generic structure of a forward search algorithm, as illustrated in Fig. 1, suggests two natural approaches for the transparent exploitation of parallelism.

3.1 Vertical Parallelism

The algorithm in Fig. 1 explores a search space, described by the possible b-states and the function Φ (or Φ^*). *Vertical Parallelism* arises from the exploitation of parallelism from the non-determinism of the search process—i.e., allowing the concurrent exploration of different (S, p) extracted from the queue (**Line** 3). This is intuitively described in Fig. 2 (left). Effectively, the different agents are exploring alternative ways to reach a goal b-state, by concurrently building distinct plans. The advantage of this approach is the possibility of pursuing alternative plans, which is particularly advantageous when the heuristic function is ineffective in discriminating between b-states to explore. On the other hand, if the different agents use a common representation of the search space (e.g., a single queue), then we run the risk of *(1)* exploring speculative parts of the search space (e.g., b-states with a low heuristic value), and *(2)* modifying the order of exploration of the search space (which might negatively impact the heuristic search).

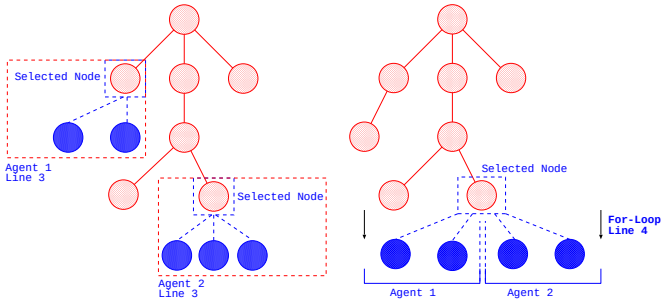


Fig. 2. Vertical and Horizontal Parallelism

3.2 Horizontal Parallelism

Horizontal Parallelism arises from the use of different agents in constructing *one* particular plan—thus, the agents are cooperating in the development of a *single* plan. This is effectively achieved by distributing the iterations of the for-loop of **Line** 4 between the agents, as intuitively illustrated in Fig. 2 (right). The advantage of this approach is the fact that the structure of the search space is unchanged w.r.t. a sequential execution—thus, parallelism does not interfere with the heuristics function. The drawbacks arise from the potentially small granularity of the tasks (e.g., when the states in $\Phi(a, S)$ are "easy" to compute) and the possible contention in the use of a common queue.

4 Systems Description

4.1 mCpA: a Parallel CPA

The parallel versions of CPA have been developed using a common parallel structure. The model adopted relies on a multi-core platform, relying on shared memory for all communication tasks. The agents in charge of performing the computation are represented by concurrent threads; all forms of communication between the computing agents are realized through access and modifications of data structures allocated in the shared memory. Synchronization is required to coordinate access to shared memory (via mutex locks).

We explore alternative implementations, aimed at exploiting specific computational features of the planning domains. VERT is a purely vertical implementation, aimed at domains where the heuristics function is not satisfactory (e.g., many b-states are generated with the same best heuristic value). The horizontal models (HORZ1-HORZ4) are aimed at domains where the computation of the successor states is expensive. In particular, HORZ1 considers the case of large number of actions or actions having comparable "complexity", while HORZ2 and HORZ3 address domains where the number of actions might be large but the cost of applying the actions and determining successor states might widely vary. The HORZ4 prototype considers domains where actions are very simple (not expensive) and overheads should be avoided. The final two models (HYBR1 and HYBR2) address domains where both vertical and horizontal conditions are present (possibly only in moderate terms).

Vertical Parallelism In this implementation, called VERT, we maintain a unique central queue, representing the frontier of the search space, with the b-states ranked according to the heuristics function (as in Fig. 1). The central queue is a priority queue and is allocated in shared memory. We also use a shared table to store visited b-states. Modifications of the central queue and the visited b-states table are critical sections and mutex locks are acquired by the agents for each update.

During the search, each agent P extracts a b-state S with the highest heuristic value from the central queue and determines actions that are executable in S. For each executable action, P computes the successor b-state S' and its heuristic value. If S' satisfies the goal then a solution is returned. Otherwise, if S' is not present in the visited b-states table, P will add S' to the queue and to the visited b-states table. When an agent P satisfies the goal, it sets a flag to notify the others of termination. To detect the situation in which there is no solution, each agent P is associated to a flag that indicates whether P is idle—P is idle if it runs out of work and the central queue is empty. When all agents are idle they stop and report that no solution has been found.

Horizontal Parallelism For horizontal parallelism, we have realized four different implementations—exploring alternative strategies to interact with the central queue.[4]

The HORZ1 implementation relies on the use of a central queue to store open nodes, and a set structure to store visited b-states, both located in shared memory. In HORZ1,

[4] This type of variations have led to significant differences in other parallel systems [14].

we *statically* divide the set of all actions into equal size segments, and assign each segment to a distinct computing agent. During the search, an arbitrary agent, say P_0, extracts a b-state S with the highest heuristic value from the queue, and all agents, including P_0 itself, compute the successor b-states for S. Each agent P computes the successor b-states of S for each applicable action belonging to the segment of actions assigned to P. These b-states are stored locally and only in the end transferred to the central queue (with corresponding update of the visited b-states table)—this guarantees, in most of the cases, that the b-states in the central queue are in the same order as in a sequential execution of CPA.[5]

The HORZ2 implementation is similar to HORZ1, however, instead of assigning to an agent a fixed number of actions to compute the successor b-states, we *dynamically* allocate actions to agents at run time. Given n agents and m actions that need to be tested in computing the successor b-states, if agent P is available, then P will receive a segment containing $\frac{m}{4\times n}$ unexplored actions[6]; at the same time, the value m of unexplored actions is updated to $m-\frac{m}{4\times n}$. This process is repeated until the successor b-states for all actions have been computed. The net effect is to start by assigning coarse tasks to the agents, and refining them to smaller tasks as the computation continues, ultimately creating a better load balancing between the computing agents (since different actions may require a different amount of time). All the computed successor b-states are stored in a local queue, associated to each agent. As in HORZ1, when all the actions have been applied, the content of the local queues is transferred to the central queue.

HORZ3 is similar to HORZ2, with the exception that each agent receives only one action at the time—creating fine grained tasks and facilitating load balancing.

Finally, the HORZ4 is identical to HORZ3, except that the b-states computed by the agents are immediately inserted in the central queue. The goal is to avoid the sequential phase required to transfer b-states from the local queues to the central one. The new b-states may appear in the central queue in an order different from the one of sequential execution (if multiple b-states with the same highest heuristics value are present). Thus, the parallel planner is likely to explore the search space in a different order than sequential execution.

Hybrid Parallelism HYBR1 is a combination of horizontal and vertical parallelism. The agents are divided into groups of equal size (4 in our experiments). An arbitrary agent P of each group extracts the b-state from the top of the central queue, and shares the work of computing successor b-states with all agents in the group, including P itself, as done in HORZ1. Checking visited b-states is done as in the previous implementations. The central queue and the visited b-states table are allocated in shared memory.
In HYBR2, agents are divided into two groups—each with a private queue and visited states table. Unlike HYBR1, where all groups play the same role, in HYBR2, the first group uses the strategy of VERT: at any time during the search each agent in the group extracts the b-state from the top of the group's queue, computes the successor b-states for executable actions and inserts them into the group's queue. The second group works in the same way as HORZ1, where actions are divided into segments of equal size and

[5] There are rare exceptions to this, as discussed later.
[6] This formula has been experimentally chosen.

each segment is associated with an agent in the group to compute successor b-states. The intuition is to provide a balance between trusting and speeding up the original heuristics (pursued by the second team), and allowing the fast exploration of alternative branches.

4.2 mFF: a Parallel FF

The parallelization of FF follows the model of vertical parallelism. The choice of not exploring horizontal parallelism is dictated by the high-efficiency of FF in computing Φ, which would make horizontal parallelism too fine-grained. The implementation of vertical parallelism has been adapted to FF with the following main modifications:

- One agent (*master*) maintains the central queue, and performs best-first search following the sequential FF scheme.
- The other agents (*slaves*) are allowed to extract b-states from the central queue and perform search starting from the given b-state. Given a b-state S, the slave agent proceeds by first attempting a hill-climbing local search starting from S, and entering the best-first search only upon failure of hill-climbing (thus, effectively restarting the FF computation from S as initial state). The best-first search conducted by the slave is limited by a maximum number of steps, and the frontier of the final search tree developed by the slave agent replaces S in the central queue (if a solution is found, it will be reported and the computation will terminate).

Asserting a limit on the best-first search conducted by each slave agent is useful to guarantee that the slave agent does not enter a "low quality" part of the search tree, without requiring excessive interaction with the other agents and the central queue. The number of steps allowed is initially set to an experimentally determined constant (500 in the proposed experiments) and it is adaptive—it is incremented as the search goes deeper into the global search tree.

5 Experimental Evaluation

5.1 Benchmarks

To evaluate the performance of our systems, we use two test suites. The first includes classical planning domains, while the second includes conformant planning domains. The benchmarks are briefly described in Table 1. In the discussion, we loosely use the term *speedup* to denote the ratio between the sequential and the parallel execution time, a measure of how much parallel search contributes to improve the performance of a given sequential system.[7] Note that we do not present all the results (e.g., performance of each form of parallelism for each benchmark) due to lack of space.

5.2 Classical Domains

We experimented with both mCpA and mFF on classical planning domains. In both cases, we selected domains and instances which have been proved challenging for the

[7] This is different from the theoretical notion of speedup, that requires the absolute best sequential time using *any* sequential system.

Benchmark	Instances	Source	Brief Description
	Classical Domains		
Gripper	$n = 100$	AIPS 1998	robot transports n balls between 2 locations
Miconic	$p = 20, f = 20$	AIPS 2000	lift transports p passengers between f floors
Pathways	$p = 1, \ldots, 30$	IPC-5	find sequence of biochemical reactions producing p substances
PipesWorld	$p = 10$	IPC-5	Control flow of oil in a p-node network
Stacks	$p = 30$	IPC-5	Problem in production scheduling
Storage	$p = 17$	IPC-5	moving crates from containers to p depots
	Conformant Domains		
Bomb	$b = 200, t = 20$	[6]	disarm bombs by dunking in the toilet
Cleaner	$n = 10, p = 50$	[24]	robot cleans p objects in n rooms
Ring	$n = 30$	[9]	robot locks windows in n rooms
Cube	$n = 9$	[9]	robot moves in a $n \times n \times n$ grid
Safe	$n = 50$	[6]	open a safe with n possible combinations)
Logistic	$l = 3, c = 3, p = 3$	[6]	transport p packages within l locations in c cities
Coin	$n = 10$	IPC-5	robot collects n coins scattered in a building
Comm	$s = 14/12, p = 11/9$	IPC-5	certify or repair packets

Table 1. Benchmarks

sequential counterparts. Times are reported in seconds and the experiments rely on a 2-hour time limit (*TO* denotes time-out). Each problem was solved four times and average execution times are reported.

`mCpA: mCpA` has been implemented on a Sun multicore shared memory machine, with 8 cores, 4GB of memory, and running Solaris 9. Table 2 reports the experimental results using different numbers of agents (from 1 to 8). The leftmost column of the table shows the problem and the version of `mCpA` used. In each of the other cells, we report the execution time for the corresponding parallel version, and the corresponding number of agents, followed by the ratio of performance improvement w.r.t. sequential CPA.

VERT obtained good performance on the Miconic domain—a speedup of 12.94 using 8 agents in $Mic(20, 20)$. The reason for the super-linear speedup is due to the fact that multiple agents are exploring different branches of the search tree, and VERT determines a shorter plan than CPA. E.g., for $Mic(20, 20)$, the length of the plan that VERT with 8 agents returned is 86, comparing to 166 of the sequential version. The speedups of HORZ1 and HORZ3 on this problem are also fairly good—more than 6 using 8 agents. They are lower due to the relatively fast computation of successor b-states. We observe drops in speedups in the other systems due to contention on locks and lack of sufficient agents to follow the most promising plans.

In the Gripper domain, the length of the plan returned by VERT is not always shorter than that found by CPA. E.g., with 8 agents, the length of the plan returned by VERT is 377 while that of CPA is 319. That explains why the speedup of the vertical parallel implementation VERT is not as good. In general, the speedups obtained by all the versions, on this domain, are stable, gradually increasing as the number of agents increases. This is because the length of the plan found by the sequential version is not far from that of

an optimal plan (i.e., the heuristics of CPA works well on this domain). Thus, in most of the cases, the parallel solutions represent better approximations of the optimal plan. Note that, in general, optimality is desired but not required.

The Pathway domain, with deterministic actions and a large number of actions and fluents and without axioms, reminds us how important heuristic is in searching for a plan. Here , horizontal parallelism does not pay off, while vertical parallelization provides good speedup.

Domain	n=2	n=4	n=8
Mic(20,20)	CPA: 1232		
VERT	396 (3.12)	193 (6.39)	95 (12.94)
HORZ1	637 (1.94)	340 (3.63)	204 (6.04)
HORZ2	693 (1.78)	525 (2.35)	398 (3.10)
HORZ3	639 (1.93)	333 (3.70)	197 (6.25)
HORZ4	1094 (1.13)	291 (4.23)	154 (7.99)
HYBR1	617 (2.00)	400 (3.08)	182 (6.78)
HYBR2	1236 (1.00)	637 (1.94)	198 (6.21)
Gripper(100)	CPA: 5255		
VERT	3095 (1.70)	2109 (2.49)	1032 (5.09)
HORZ1	2711 (1.94)	1395 (3.77)	778 (6.75)
HORZ2	2821 (1.86)	1499 (3.51)	878 (5.99)
HORZ3	2687 (1.96)	1379 (3.81)	763 (6.89)
HORZ4	2638 (1.99)	1414 (3.72)	788 (6.67)
HYBR1	3899 (1.35)	3260 (1.61)	1591 (3.30)
HYBR2	5276 (1.00)	2722 (1.93)	1404 (3.74)
Pathways(4)	CPA: 23.02		
VERT	4.57 (5.04)	7.2 (3.20)	3.39 (6.79)
HORZ1	20.05 (1.15)	17.51 (1.31)	
HORZ2	15.76 (1.46)	12.24 (1.88)	12.77 (1.80)
HORZ3	15.87 (1.45)	11.03 (2.09)	12.01 (1.92)
HORZ4	15.07 (1.53)	10.06 (2.29)	6.75 (3.41)
HYBR1	12.85 (1.79)	7.12 (3.23)	6.98 (3.30)
HYBR2	27.08 (0.85)	5.96 (3.86)	6.23 (3.70)

Table 2. Classical Benchmarks using `mCpA`

`mFF: mFF` has been developed on the same 8-core Sun server, running Solaris 9. The experiments conducted deal with instances of problems (drawn from the IPC-5 competition) that are challenging for the sequential FF system. Table 3 shows the main results. In parentheses we show the ratio between sequential time and parallel time. The important result to underline here is the ability to solve various instances that are intractable by sequential FF (actually, many of the time-outs reported are for instances that took longer than 24 hours). In this context, the benefits are parallelism are two-fold:

- the ability to overlap different *types* of search (standard best-first search and hill-climbing local search)—this is the case of Storage(17), where hill-climbing is time consuming while best-first quickly converges to a solution;

- the ability to concurrently explore branches that have equally high heuristic values (this is the case of Pathways).

Observe that the current implementation is not particularly good in maintaining a low level of communication—we can notice that using 8 agents the performance occasionally degrades (e.g., PipesWorld(15)) due to excessive contentions on the locks of the central queue.

Domain	Instance	Number of Agents			
		FF 2.3	2	4	8
Pathways	9	TO	4.47 (_)	2.71 (_)	3.06 (_)
	11	TO	6.57 (_)	6.55 (_)	3.16 (_)
	12	TO	TO (_)	TO (_)	264.79 (_)
	13	TO	4.64 (_)	3.51 (_)	4.03 (_)
	15	TO	48.39 (_)	4.59 (_)	4.61 (_)
	20	83.08	75.45 (1.10)	21.21 (3.92)	17.83 (4.66)
	30	TO	121.18 (_)	7.09 (_)	7.08 (_)
Stacks	30	202.82	203.16 (0.99)	190.6 (1.06)	185.01 (1.1)
PipesWorld	9	180.64	92.12 (1.96)	67.44 (2.68)	110.01 (1.64)
	10	442.62	48.7 (9.01)	35.28 (12.55)	37.9 (11.68)
	11	64.53	15.49 (4.17)	15.06 (4.28)	12.3 (5.25)
	15	1289.01	1280.15 (1.0)	502.53 (2.57)	1197.8 (1.08)
	20	TO	TO	1393.87 (_)	1259.15 (_)
Storage	17	732.36	8.95 (81.83)	4.93 (148.55)	0.91 (804.4)

Table 3. Results using mFF

5.3 Conformant Domains

The experiments on conformant domains have been performed using mCpA. For reference, we compared our systems with CFF [6] and KACMBP [9], which are two of the fastest conformant planners. Unfortunately, we could not obtain the Solaris executables for these planners, and we had to run them on a Linux machine and scale the timings to our Sun multi-core.[8] The comparison with the other systems is illustrated in Table 4. Some experimental results are shown in Figure 3.

Domain	CPA	CFF	KACMBP
Bomb(200,20)	248	23763	2411
Cleaner(10,50)	390	Maximum length exceeded	TO
Ring(30)	423	TO	< 1
Cube(9)	812	TO	< 1
Safe(50)	1865	68	< 1
Log(3,3,3)	1025	< 1	745
Coin(10)	487	< 1	1502

Table 4. Conformant Benchmarks

[8] For each problem, we compute a time conversion ratio by running CPA on both the Linux and Solaris platforms.

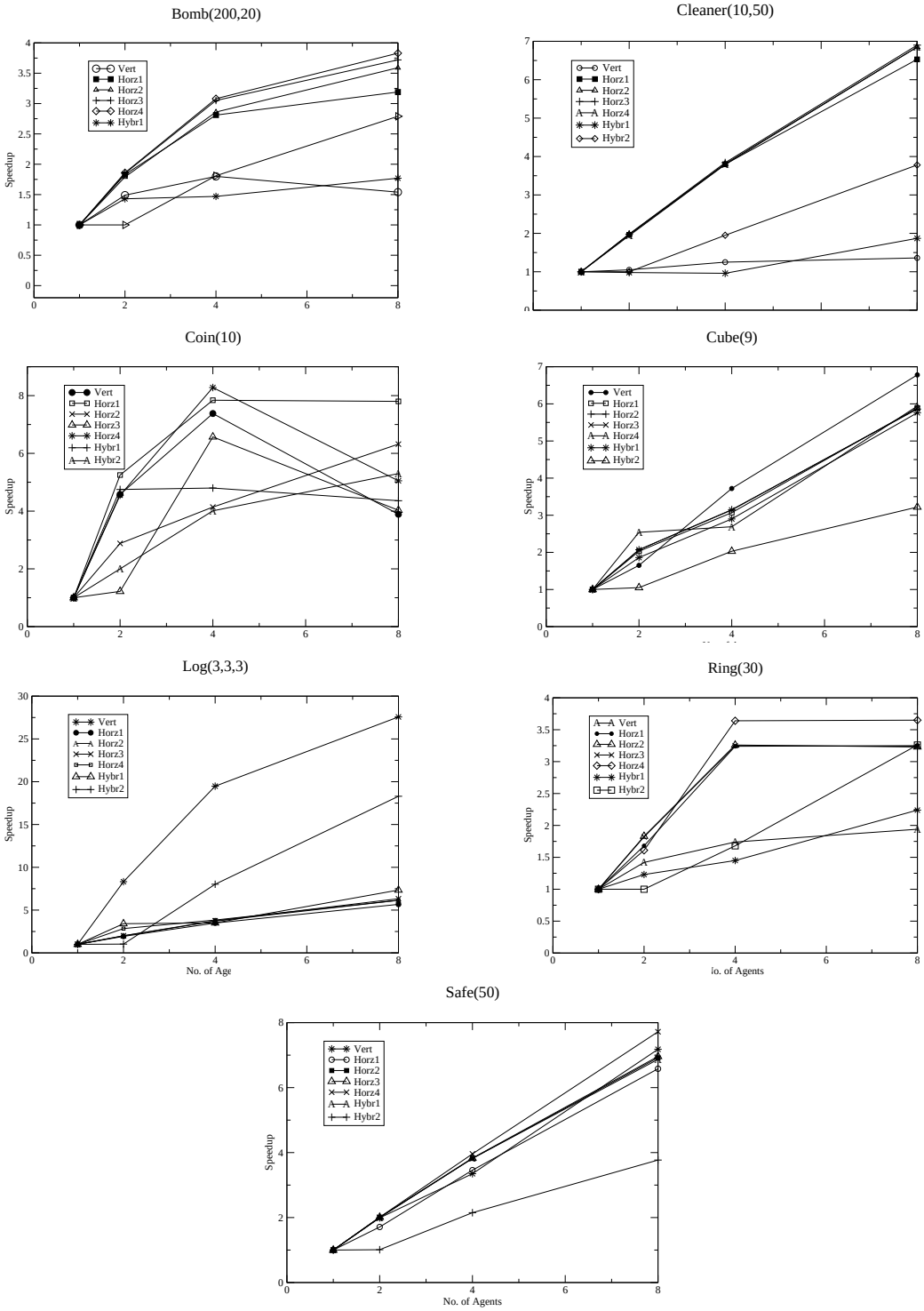


Fig. 3. Speedups on Conformat Domains

In the Bomb and Cleaner domains, we do not see much speedup in VERT when the number of agents increases. This is because the heuristic of CPA works well on this domain. The best parallel implementation on this domain is HORZ4. The reason why HORZ4 is better is because the newly computed successor b-states are immediately inserted in the queue (instead of waiting for all agents to complete). Horizontal parallelism, however, performs very well on the Cleaner domain—since computing successor b-states is expensive.

In the Ring domain, the horizontal parallelism implementations scale well up to 4 agents, and then they become stable. The reason is that the number of actions in the domain is only 4, leaving other agents idle. A similar behavior occurs in the Cube domain, whose number of actions is 6. The speedups of VERT on Cube(9) is good as the system is capable of finding a shorter plan (43 steps, compared to 63 of sequential CPA). In the Safe domain, the speedups of both vertical and horizontal implementations are very good.

The Logistic domain is truly problematic for the sequential version CPA because the heuristic function performs poorly, especially on the Log(4,3,3) problem. Sequential CPA, KACMBP, and most of our parallel implementations, except HORZ2, could not solve it within the time limit. Thanks to parallelism, we could solve this problem (HORZ2) using 8 agents. For the Log(3,3,3) problem, the speedups obtained by VERT and HYBR2 are impressive (more than 25 on 8 agents for the VERT implementation). In the Coin domain, the speedups obtained by our parallel implementations range from 1.22 (HORZ3, 2 agents) to 8.28 (HORZ4, 4 agents).

The occasional super-linear speedups are due to changes in the search pattern caused by parallelism; in the case of vertical parallelism, this is obvious (as multiple paths are concurrently explored). In the case of horizontal parallelism, this may occur because the order of the b-states in the central queue might differ from the sequential execution (the heuristic is currently computed on the first state of a b-state, and this may change during horizontal parallelism).

In summary, on the domains where the heuristic function does not perform well (i.e., various elements receive the highest value, and the one on top of the queue might not lead to the shortest plan), the vertical parallelism is very effective—sometimes we obtain super-linear speedups. In contrast, in the domains where the heuristic function performs well, the speedup obtained by the horizontal approach, although less than linear, is usually good. Furthermore, the more expensive the computation of a successor b-state is, the higher the speedup obtained via horizontal parallelism. The hybrid approach balances these two extremes.

6 Related Work

The work proposed in this paper is in the same spirit as the work of [26], where a parallelization scheme similar to our horizontal parallelism is applied to a STRIPS planner—in their context this is useful to handle the cost of applying operators with variables (while we use it to address the use of axioms), but it will be less effective for domains that have a fast computation of the next state.

Parallel planning as considered in this paper is different from *distributed planning* [21, 12] in that we focus on improving sequential planning systems by distributing the workload of an agent to multiple agents while distributed planning often deals with the problem of coordination between agents to create a plan for all agents. Distributed planning often requires the execution of plans by the agents and agents are often reactive. Agents in our framework share the same goal and representation, stop when one find a plan, and do not execute actions to change the world. Furthermore, efficiency is not the first issue in distributed planning. Issues of parallelization have been explored in this context, by either partitioning actions and goals between agents so that separate plans can be computed and composed (e.g., interaction graphs [17]) or by adopting a hierarchical approach, where distinct "regions" of the plan are given to distinct agents (e.g., [10]). These approaches are "global" versions of horizontal parallelism. Vertical parallelism has been explored in other search-based problems, e.g., [22, 14].

Our approach to planning in this paper is perhaps more closely related to the distributed problem planning in which the planning process is distributed but a centralized plan needs to be found as discussed in [11]. Our proposed approach could be viewed as a special case of system in view of [11] where one agent distributes the work and all agents search for a plan until one is found.

The work presented in this paper is also similar to the path-finding problem in [27] where different search algorithms for finding a path in distributed environment have been proposed. These algorithms do not deal with incomplete information though. Furthermore, we would like to point out that our goal is not to propose a generic algorithms for parallel search.

7 Conclusions and Future Work

Over the years, two main reasons have led to performance improvements of automated planners: new algorithms and faster computers. The latter has allowed to apply the same planning algorithms to solve more complex problems without any changes. This trend is expected to change, as computer manufacturers are moving away from focusing on single-thread performance and focusing on multi-core platforms. In this paper we presented an investigation of alternative methodologies for parallelization of heuristic search-based planners on multi-core platforms. We identified two forms of parallelism and investigated their implementations and interactions. The results are very encouraging, in terms of improved execution time, speedups, and scalability. We are currently exploring the porting of these ideas in a fully distributed platform as well as taking better advantage of the growingly popular hybrid Beowulf clusters—i.e., clusters whose nodes are multi-core platforms.

References

1. F. Bacchus. The AIPS'00 Planning Competition. *AI Magazine*, 22(3), 2001.
2. C. Baral and M. Gelfond. Reasoning agents in dynamic domains. pages 257–279. Kluwer Academic Publishers, 2000.

3. C. Baral et al. Computational complexity of planning and approximate planning in the presence of incompleteness. *AIJ*, 2000.
4. A.L. Blum and M.L. Furst. Fast Planning through Planning Graph Analysis. *AIJ*, 90:281–300, 1997.
5. B. Bonet and H. Geffner. Planning as Heuristic Search. *AIJ*, 129(1–2):5–33, 2001.
6. R. Brafman and J. Hoffmann. Conformant planning via heuristic forward search: A new approach. *ICAPS*, 2004.
7. D. Bryce et al. Planning Graph Heuristics for Belief Space Search. *JAIR*, 26:35–99, 2006.
8. C. Castellini et al. SAT-based Planning in Complex Domains. *Artificial Intelligence*, 147:85–117, July 2003.
9. A. Cimatti et al. Conformant Planning via Symbolic Model Checking and Heuristic Search. *AI Journal*, 159:127–206, 2004.
10. M. desJardins, M. Wolverton. Coordinating Planning Activity and Information Flow in a Distributed Planning System. *AAAI Fall Symp.* 1998.
11. E. Durfee. Distributed Problem Solving and Planning. *Multiagent Systems*, MIT Press. 1999.
12. J. Urban and P. Dasgupta. *The Encyclopedia of Distributed Computing*. Kluwer Pubs., 2003.
13. T. Eiter et al. A Logic Programming Approach to Knowledge State Planning, II. *Artificial Intelligence*, 144(1-2), 2003.
14. G. Gupta et al. Parallel Execution of Prolog Programs: a Survey. *ACM TOPLAS*, 23(4):472–602, 2001.
15. P. Haslum et al. New admissible heuristics for domain-independent planning. In *AAAI*, 1163–1168, 2005.
16. J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *JAIR*, 14:253–302, 2001.
17. M. Iwen and A. Mali. Distributed graphplan. *ICTAI*, IEEE, 2002.
18. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. *AAAI*, pp. 1194–1199, 1996.
19. H. Kitan and J.A. Hendler. *Massive parallel artificial intelligence.* MIT Press, 1994.
20. V. Lifschitz. Answer set programming and plan generation. *AIJ*, 138(1–2), 2002.
21. A.D. Mali and S. Kambhampati. Distributed Planning. *Encyclopedia of Distributed Computing*, Kluwer, 2003.
22. L. Perron. Search Procedures and Parallelism in Constraint Programming. *CP*, Springer Verlag, 346–360, 1999.
23. D. Smith and D. Weld. Conformant graphplan. *AAAI*, 1998.
24. T. C. Son et al. Conformant Planning for Domains with Constraints — A New Approach. In *AAAI*, 1211–1216, 2005.
25. S. Thiebaux, J. Hoffmann, and B. Nebel. In Defense of PDDL Axioms. In *IJCAI*, 2003.
26. D. Vrakas, I. Refanidis, and I. P. Vlahavas. Parallel planning via the distribution of operators. *JETAI*, 13(3):211–226, 2001.
27. M. Yokoo et al. Search Algorithms for Agents. *Multiagent Systems*, MIT Press. 1999.
28. H. Zhang et al. PSATO: a distributed propositional solver and its application. *Journal of Symbolic Computing*, 21(4), 1996.

Memoizing Multi-Threaded Transactions

Lukasz Ziarek Suresh Jagannathan

Department of Computer Science Purdue University
[lziarek,suresh]@purdue.edu

Abstract. There has been much recent interest in using transactions to simplify concurrent programming, improve scalability, and increase performance. When a transaction must abort due to a serializability violation, deadlock, or resource exhaustion, its effects are revoked, and the transaction re-executed. For long-lived transactions, however, the cost of aborts and subsequent re-executions can be prohibitive. To ensure performance, programmers are often forced to reason about transaction lifetimes and interactions while structuring their code, defeating the simplicity transactions purport to provide.

One way to reduce the overheads of re-executing a failed transaction is to avoid re-executing those operations that were unaffected by the violation(s) that induced the abort. Memoization is one way to capitalize on re-execution savings, especially if violations are not pervasive. Abstractly, if a procedure p is applied with argument v within a transaction, and the transaction aborts, p need only be re-evaluated when the transaction is retried if its argument is different from v.

In this paper, we consider the memoization problem for transactions in the context of Concurrent ML (CML) [20]. Our design supports multi-threaded transactions which allow internal communication through synchronous channel-based communication. The challenge to memoization in the context is ensuring that communication actions performed by memoized procedures in the original (aborted) execution can be satisfied when the transaction is retried.

We validate the effectiveness of our approach using STMBench7 [9], a customizable transaction benchmark. Our results indicate that memoization for CML-based transactions can lead to substantial reduction in re-execution costs (up to 45% on some configurations), with low memory overheads.

1 Introduction

Concurrency control mechanisms, such as transactions, rely on efficient control and state restoration mechanisms for performance. When a transaction aborts, due to a serializability violation [8], deadlock, transient fault [24], or resource exhaustion [11], its effects are typically undone, and the transaction retried. A long-lived transaction that aborts represents wasted work, both in terms of the operations it has performed whose effects must now be erased, and in terms of overheads incurred to implement the concurrency control protocol; these overheads include logging costs, read and write barriers, contention management, etc. [12].

Transactional abstractions embedded in functional languages (e.g., AtomCaml [21], proposals in Scheme48 [14], or STM Haskell [11]) benefit from having relatively few stateful operations, but when a transaction is aborted, the cost of re-execution still remains. One way to reduce this overhead is to avoid re-executing those operations that

would yield the same results produced in the original failed execution. Consider a transaction T that performs a set of operations, and subsequently aborts. When T is re-executed, many of the operations it originally performed may yield the same result, because they were unaffected by any intervening global state change between the original (failed) and subsequent (retry) execution. Avoiding re-execution of these operations reduces the overhead of failure, and thus allows the programmer more flexibility and leeway to identify regions that would benefit from being executed transactionally.

Static techniques for eliminating redundant code, such as subexpression elimination or partial redundancy elimination, are ineffective here because global runtime conditions dictate whether or not an operation is redundant. Memoization [15, 18] is a well-known dynamic technique used to eliminate calls to pure functions. If a function f supplied with argument v yields result v', then a subsequent call to f with v can be simply reduced to v' without re-executing f's body, *provided* that f is effect-free.

In this paper, we consider the design and implementation of a memoization scheme for an extension of Concurrent ML [20] (CML) that supports multi-threaded transactions. CML is particularly well-suited for our study because it serves as a natural substrate upon which to implement a variety of different transactional abstractions [7]. In our design, threads executing within a transaction communicate through CML synchronous events. Isolation and atomicity among transactions are still preserved. Multi-threaded transactions can, thus, be viewed as a computation which executes atomically and in isolation instead of a simple code block. Our goal is to utilize memoization techniques to avoid re-execution overheads of long-lived multi-threaded transactions that may be aborted.

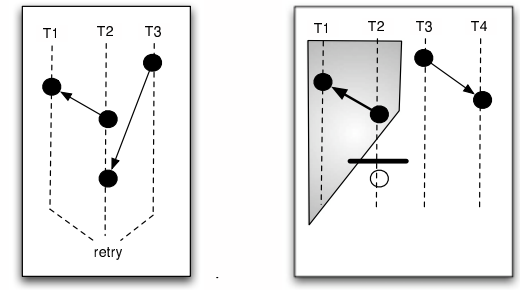
The paper is organized as follows. In the next section, we describe our programming model, and introduce issues associated with memoization of synchronous communication actions. Section 3 provides additional motivation. We introduce *partial memoization*, a refinement that has substantial practical benefits in Section 4. Implementation details are given in Section 5. We present a case study using STM-Bench [9], a highly-concurrennt transactional benchmark, in Section 6. Related work and conclusions are given in Section 7.

2 Programming Model

Our programming model supports multi-threaded closed nested transactions. An expression wrapped within an `atomic` expression is executed transactionally. If the value yielded by a transaction is `retry`, the transaction is automatically re-executed. A transaction may `retry` because a serializability violation was detected when it attempted to commit, or because it attempts to acquire an unavailable resource [11]. An executing transaction may create threads which in turn may communicate with other threads executing within the transaction using synchronous message passing expressed via CML selective communication abstractions. To enforce isolation, communication between threads executing within different transactions is not permitted. A transaction attempts to commit only when all threads it has spawned complete. Updates to shared channels performed by a transaction are not visible to other transactions until the entire transaction completes successfully.

We are interested in allowing multi-threaded transactions primarily for reasons of composability and performance. A computation wrapped within an atomic section may invoke other procedures that may spawn threads and have these threads communicate with one another. This is especially possible when considering long-lived transactions that encapsulate complex computations involving multiple layers of abstraction. Prohibiting such activity within an atomic section would necessarily compromise composability. Moreover, allowing multi-threaded computation may improve overall transaction performance; this is certainly the case in the benchmark study we present in Section 6. Inter-thread communication within a transaction is handled through dynamically created channels on which threads place and consume values. Since communication is synchronous, a thread wishing to communicate on a channel that has no ready recipient must block until one exists. Communication on channels is ordered.

2.1 Memoization



(a) (b)

Fig. 1. Threads are represented as dotted lines while circles define communication points. The shaded area depicts computation which can be memoized based on communications which are satisfiable.

A transaction may spawn a collection of threads that communicate with one another via message-passing (see Fig. 1(a)). When the transaction is retried (see Fig. 1(b)), some of these communication events may be satisfiable when the procedures in which the events occurred are invoked (e.g., the first communication event in the first two threads), while others are not (e.g., the second communication action performed by thread T2 to thread T3). The shaded region indicates the pure computation in T1 and T2 that may be avoided when the transaction is re-executed. Note that T2 must resume execution from the second communication action because the synchronization action with T3 from the aborted execution is not satisfiable when the transaction is retried; since the send action by T3 was received by T4, there is no other sender available to provide the same value to T2.

In this context, deciding whether an application of procedure f can be avoided based on previously recorded memo information depends upon the value of its arguments, the

communication actions performed by f, threads f spawns, and f's return value. Thus, the memoized return value of a call to f can be used if (a) the argument given matches the argument previously supplied; (b) recipients for values sent by f on channels in an earlier call are still available on those channels; (c) a value that was consumed by f on some channel in an earlier call is again ready to be sent by another thread; and (d) threads created by f can be spawned with the same arguments supplied in the memoized version. Ordering constraints on all sends and receives performed by the procedure must also be enforced.

To avoid performing the pure computation within a call, a send action performed within the applied procedure, for example, will need to be paired with a receive operation executed by some other thread. Unfortunately, there may be no thread currently scheduled that is waiting to receive on this channel. Consider an application that calls a memoized procedure f which (a) creates a thread T that receives a value on channel c, and (b) sends a value on c computed through values received on other channels that is then consumed by T. To safely use the memoized return value for f nonetheless still requires that T be instantiated, and that communication events executed in the first call can still be satisfied (e.g., the values f previously read on other channels are still available on those channels). Ensuring these actions can succeed involves a systematic exploration of the execution state space to induce a schedule that allows us to consider the call in the context of a global state in which these conditions are satisfied.

Because such an exploration may be infeasible in practice, our formulation considers a weaker alternative called *partial* memoization. Rather than requiring global execution to reach a state in which *all* constraints in a memoized application are satisfied, partial memoization gives implementations the freedom to discharge some fraction of these constraints, performing the rest of the application as normal.

3 Tracking Communication Actions

The key requirement for effective memoization of procedures executing within CML transactions is the ability to track communication actions performed among procedures. Provided that the global state would permit these same actions to succeed if a procedure is re-executed with the same inputs, memoization can be employed to reduce re-execution costs.

```
atomic(fn () =>
       let val  (c1, c2) = (mkCh(), mkCh())
           fun  f() = (...; send(c1, v1); ...)
           fun  g() = (recv(c1); send(c2,v2))
       in spawn(f()); spawn(g()); recv(c2)
       end)
```

Fig. 2. The call to `f` can always be memoized since there is only a single receiver on channel `c1`.

Consider the example code presented in Fig. 2 that spawns two threads to execute procedures `f` and `g` within an atomic section. Suppose that the section fails to commit,

and must be retried. To correctly utilize `f` 's memoized version from the original failed execution, we must be able to guarantee the value sent on channel `c1` has a recipient. At the time the memoization check is performed, the thread computing `g` may not even have been scheduled. However, by delaying the memoization decision for `f` 's call until `g` is ready to receive a value on `c1`, we guarantee that memoized information stored for `f` can be successfully used to avoid performing the pure computation within its body.

```
atomic(fn () =>
        let val (c1, c2, c3) =
                (mkCh(), mkCh(), mkCh())
            fun f() = (...; send(c1,v1); recv(c2))
            fun g() = (recv(c1); recv(c2))
            fun h() = (send(c2,v2);
                       send(c2,v3);
                       send(c3,()))
        in (spawn(f()); spawn(g()); spawn(h());
            recv(c3))
        end)
```

Fig. 3. Because there may be multiple possible interleavings that pair synchronous communication actions among concurrently executing threads, memoization requires dynamically tracking these events.

Unfortunately, reasoning about whether an application can leverage memoized information is usually more difficult. Consider a slightly modified version of the program shown in Fig. 3, that introduces an auxiliary procedure `h`. Procedure `f` communicates with `g` via channel `c1`. It also either receives value `v2` or `v3` from `h` depending upon its interleaving with `g`. Suppose that when this section is first executed, `g` receives values `v2` from `h` and `f` receives value `v3`. If the section must be re-executed, the call to `f` can be avoided only if the interleaving of actions between `g` and `h` allow `f` to receive `v3`. Thus, a decision about whether the call to `f` can be elided requires also reasoning about the interactions between `h` and `g`, and may involve enforcing a specific schedule to ensure synchronous operations mirror their behavior under the aborted execution.

Notice that if `v2` and `v3` are equal, the receive in `f` can be paired with either send in `h`. Thus, memoization can be leveraged even under different schedules than a prior execution. Unlike program replay mechanisms [23], no qualifications are made on the state of the thread with which a memoization candidate communicates. Consequently, an application can utilize a memoized version of a procedure under a completely different interleaving of threads and need not communicate with the same threads or operations it did during its previous execution.

4 Approach

To support memoization, we must record, in addition to argument and return values, synchronous communication actions, thread spawns, channel creation etc. as part of

the memoized state. These actions define a set of constraints that must be satisfied at subsequent applications of a memoized procedure. To record constraints, we require expressions to manipulate a *memo store*, a map that given a procedure identifier and an argument value, returns the set of effects performed by the procedure when invoked with that argument. If the set of constraints returned by the memo store is satisfied in the current state, then the return value can be used, and the application elided. For example, if there is a communication constraint that expects the procedure to receive value `v` on channel `c`, and at the point of call, there exists a thread able to send `v` on `c`, evaluation can proceed to a state in which the sender's action is discharged, and the receive constraint is considered satisfied.

If the current constraint expects to send a value `v` on channel `l`, and there exists a thread waiting on `l`, the constraint is also satisfied. A send operation can match with any waiting receive action on that channel. The semantics of synchronous communication allows us the freedom to consider pairings of sends with receives other than the one it communicated with in the original memoized execution. This is because a receive action places no restriction on either the value it reads, or the specific sender that provides that the value. Similarly, if the current constraint records the fact that the previous application of the function spawned a new thread, or channel, then those actions must be performed as well. Thus, if all recorded constraints, which represent effects performed within a procedure *p*, can be satisfied in the order in which they occur, pure computation within the *p*'s body can be elided at its calls.

4.1 Partial Memoization

Determining whether all memoization constraints can be satisfied may require performing a potentially unbounded number of evaluation steps to yield an appropriate global state. However, even if it is not readily possible to determine if all constraints necessary to elide the pure computation within an application can be satisfied, it may be possible to determine that some prefix of the constraint sequence can be discharged. Partial memoization allows us to avoid re-executing any pure computation bracketed by the first and last elements of this prefix.

Consider the example presented in Fig 4. Within the atomic section, we apply procedures `f`, `g`, `h` and `i`. The calls to `g`, `h`, and `i` are evaluated within separate threads of control, while the application of `f` takes place in the original thread. These different threads communicate with one other over shared channels `c1` and `c2`.

Suppose the atomic section aborts, and must be re-executed. We can now consider whether the call to `f` can be elided when the section is re-executed. In the initial execution of the atomic section, spawn constraints would have been added for the threads responsible for executing `g`, `h`, and `i`. Second, a send constraint followed by a receive constraint, modeling the exchange of values `v1` and either `v2` or `v3` on channels `c1` and `c2` would have been included in the memo store for `f`. For the sake of the discussion, assume that the send of `v2` by `h` was consumed by `g` and the send of `v3` was paired with the receive in `f`.

The spawn constraints for the different threads are always satisfiable, and when discharged, will result in the creation of new threads which will begin their execution by trying to apply `g`, `h` and `i`, consulting their memoized versions to determine if all

```
atomic(fn () =>
   let val (c1,c2) = (mkCh(),mkCh())
       fun f ()  = (send(c1,v1); ... recv(c2))
       fun g ()  = (recv(c1); ... recv(c2))
       fun h () =  (...
                    send(c2,v2);
                    send(c2,v3));
       fun i () =  recv(c2)
   in spawn(g); spawn(h); spawn(i);
      f();  send(c2, v3)
      ...
      retry
   end)
end
```

Fig. 4. Determining if an application can be memoized may require examining an arbitrary number of possible thread interleavings.

necessary constraints can be satisfied. The send constraint associated with `f` matches the corresponding receive constraint associated found in the memo store for `g`. Determining whether the receive constraint associated with `f` can be matched requires more work. To match constraints properly, we need to force a schedule that causes `g` to receive the first send by `h` and `f` to receive the second, causing `i` to block until `f` completes.

Fixing such a schedule is tantamount to examining an unbounded set of interleavings. Instead, we could *partially* elide the execution of `f`'s call on re-execution by satisfying the send constraint (that communicates `v1` on `c1` to `g`), avoiding the pure computation following (abstracted by "..."), allowing the application of `f` to begin execution at the `recv` on `c2`. Resumption at this point may lead to the communication of `v2` from `h` rather than `v3`; this is certainly a valid outcome, but different from the original execution.

5 Implementation

Our implementation is incorporated within MLton [16], a whole-program optimizing compiler for Standard ML. The main changes to the underlying compiler and library infrastructure are the insertion of write barriers to track channel updates, barriers to monitor procedure arguments and return values, hooks to the CML library to monitor channel based communication, and changes to the Concurrent ML scheduler. The entire implementation is roughly 5K lines of SML: 3K for the STM, and 300 lines of changes to CML.

5.1 STM Implementation

Our STM implementation implements an eager versioning, lazy conflict detection protocol [4, 22]. References are implemented as "servers" operating across a set of channels; each channel has one server receiving from it and any number of channels sending to it. Our implementation uses both exclusive and shared locks to optimize read-only

transactions. If a transaction aborts or yields (`retry`), it first reverts any value it has changed based on a per-transaction change log, and then releases all locks it currently holds. The transaction's log is not deleted as it contains information utilized for memoization purposes.

Recall our design supports nested, multi-threaded transactions. A multi-threaded transaction is defined as a transaction whose processing is split among a number of threads. Transactions that perform a collection of operations on disjoint objects can have these operations be performed in parallel. The threads which comprise a multi-threaded transaction must synchronize at the transaction's commit point. Namely, the parent thread will wait at its transaction boundary until its children complete. We allow spawned threads and the parent transaction to communicate through CML message passing primitives. Synchronization invariants among concurrent computation within a transaction must be explicitly maintained by the application. The transaction as a whole, however, is guaranteed to execute atomically with the rest of the computation.

5.2 Memoization

A memo is first created by capturing the procedure's argument at the call site. For each communication within the annotated procedure, we generate a constraint. A constraint is composed of a channel identifier and the value that was sent or received on the channel. In the case of a spawn, we generate a spawn constraint which simply contains the procedure expression which was spawned. Constraints are ordered and augment the parent transaction's log. When a procedure completes, its return value is also added to the log. To support partial memoization, continuations are captured with the generated constraints.

Unlike traditional memoization techniques, it is not readily apparent if a memoized version of a procedure can be utilized at a call site. Not only must the arguments match, but the constraints which were captured must be satisfied in the order they were generated. Thus, we delay a procedure's execution to see if its constraints will be matched. Constraint matching is similar to channel communication in that the delayed procedure will block on each constraint. Constraints can be satisfied either by matching with other constraints or by exchanging and consuming values from channels. Constraints are satisfied if the value passed on the channel matches the value embedded in the constraint. Therefore, constraints ensure that a memoized procedure both receives and sends specific values and synchronizes in a specific order. Constraints make no qualifications about the communicating threads. Thus, a procedure which received a specific value from a given thread may be successfully memoized as long as its constraint can be matched with *some* thread.

If constraint matching fails, pure computation within the application cannot be fully elided. Constraint matching can only fail on a receive constraint. A receive constraint obligates a function to read a specific value from a channel. To match a constraint on a channel with a regular communication event, we are not obligated to remove values on the channel in a specific order. Since channel communication is blocking, a constraint that is being matched can choose from all values whose senders are currently blocked on the channel. This does not violate the semantics of CML since the values blocked on a channel cannot be dependent on one another; in other words, a schedule must

exist where the matched communication occurs prior to the first value blocked on the channel.

Unlike a receive constraint, a send constraint can never fail. CML receives are ambivalent to the value they remove from a channel and thus any receive on a matching channel will satisfy a send constraint. If no receives or sends are enqueued on a constraint's target channel, a re-execution of the function will also block. Therefore, failure to fully discharge constraints by stalling memoization on a presumed unsatisfiable constraint does not compromise global progress. This observation is critical to keeping memoization overheads low.

In the case that a constraint is blocked on a channel that contains no other communications or constraints, memoization induces no overheads, since the thread would have blocked regardless. However, if there exist communications or constraints that simply do not match the value the constraints expects, we can fail, and allow the thread to resume execution from the continuation stored within the constraint. To identify such situations, we have implemented a simple yet effective heuristic. Our implementation records the number of context switches to a thread blocked on a constraint. If this number exceeds a small constant (two in our current implementation), memoization stops, and the thread continues execution within the procedure body at that communication point.

Our memoization technique relies on efficient equality tests for performance and expressivity. We extend MLton's poly-equal function to support equality on reals and closures. Although equality on values of type real is not algebraic, built-in compiler equality functions were sufficient for our needs. To support efficient equality on procedures, we approximate function equality as closure equality. Unique identifiers are associated with every closure and recorded within their environment; runtime equality tests on these identifiers are performed during memoization.

5.3 CML hooks

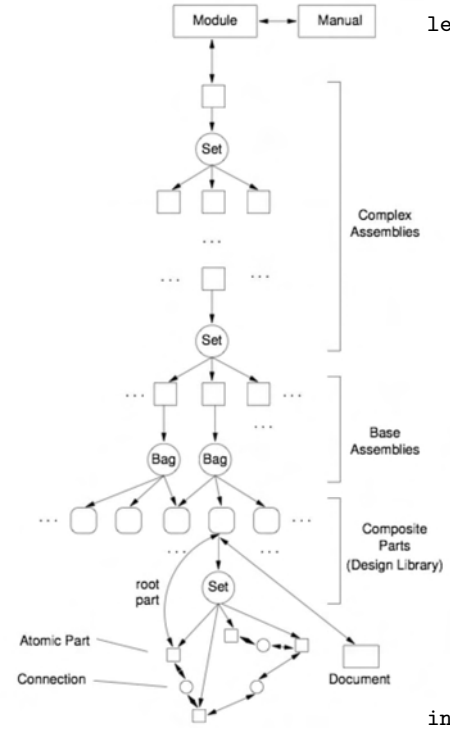
The underlying CML library was also modified to make memoization efficient. The bulk of the changes were hooks to monitor channel communication and spawns, and to support constraint matching on synchronous operations. Successful communications occurring within transactions were added to the log in the form of a constraints, as described previously. Selective communication and complex composed events were also logged upon completion. A complex composed event simply reduces to a sequence of communications that are logged separately.

The constraint matching engine also required a modification to the channel structure. Each channel is augmented with two additional queues to hold send and receive constraints. When a constraint is being tested for satisfiability, the opposite queue is first checked (e.g. a send constraint would check the receive constraint queue). If no match is found, the regular queues are checked for satisfiability. If the constraint cannot be satisfied immediately it is added to the appropriate queue.

6 Case Study - STMBench7

As a realistic case study, we consider STMBench7 [9], a comprehensive, tunable multi-threaded benchmark designed to compare different STM implementations and designs. Based on the well-known 007 database benchmark [5], STMBench7 simulates data storage and access patterns of CAD/CAM applications that operate over complex geometric structures (see Fig. 5).

STMBench7 was originally written in Java. We have implemented a port to Standard ML (roughly 1.5K lines of SML) using our channel based STM. In our implementation, all nodes in the complex assembly structure and atomic parts graph are represented as servers with one receiving channel and handles to all other adjacent nodes. Handles to other nodes are simply the channels themselves. Each server thread waits for a message to be received, performs the requested computation, and then asynchronously sends the subsequent part of the traversal to the next node. A transaction can thus be implemented as a series of channel based communications with various server nodes.




```
let fun findAtomicPart(object, pid) =
      let val assembly =
               travCAssembly(object, pid)
          val bag = travAssembly(assembly, pid)
          val component = travBag(bag, pid)
          val part = traveComp(component, pid)
      in part
      end
    fun sclHgt(object, pid, c) =
      let val part = findAtomicPart(object, pid)
          val newHeight = height(part)*recv(c)
          val _ = changeHeight(part, newHeight)
       in send(c,newHeight)
       end
    fun Traversal(object, pid1, pid2, height) =
      atomic(fn () =>
              let val c1 = channel()
                  val c2 = channel()
                  val _ = spawn(sclHgt(object,
                                       pid1,
                                       c1))
                  val _ = spawn(sclHgt(object,
                                       pid2,
                                       c2))
              in send(c1, height);
                 send(c2, recv(c1));
                 recv(c2
                 ... retry ...)
              end)
in Traversal()
end
```

Fig. 5. The figure on the left shows the overall structure of structure of a CAD/CAM object. The code on the right illustrates a multi-threaded atomic traversal of these objects.

At its core, STMBench7 builds a tree of assemblies whose leafs contain bags of components; these components have a highly connected graph of atomic parts and design documents. Indices allow components, parts, and documents to be accessed via their properties and IDs. Traversals of this graph can begin from the assembly root or any index and sometimes manipulate multiple pieces of data.

The program on the right side of Fig. 5 shows a code snippet that is responsible for modifying the height parameters of a building's structural component. A change made by the procedure `Traversal` affects two components of a design, but the specific changes to each component are disjoint and amenable for concurrent execution. Thus, the modification can easily be expressed as disjoint traversals, expressed by the procedure `findAtomicPart`. The `sclHgt` procedure shown in Fig. 5) changes the height parameter of distinct structural parts. Observe that although the height parameter of $\texttt{pid}_2$ depends on the new height of $\texttt{pid}_1$, the traversal to find the part can be executed in parallel. Once $\texttt{pid}_1$ is updated, the traversal for $\texttt{pid}_2$ can complete.

Consider what would happen if the atomic section is unable to commit. Observe that much of the computation performed within the transaction are graph traversals. Given that most changes are likely to take place on atomic parts, and not on higher-level graph components such as complex or base assemblies, the traversal performed by the re-execution is likely to overlap substantially with the original traversal. Of course, when the transaction executes, it may be that some portion of the graph has changed. Without knowing exactly which part of the graph has been modified by other transactions, the only obvious safe point for re-execution is the beginning of the traversal.

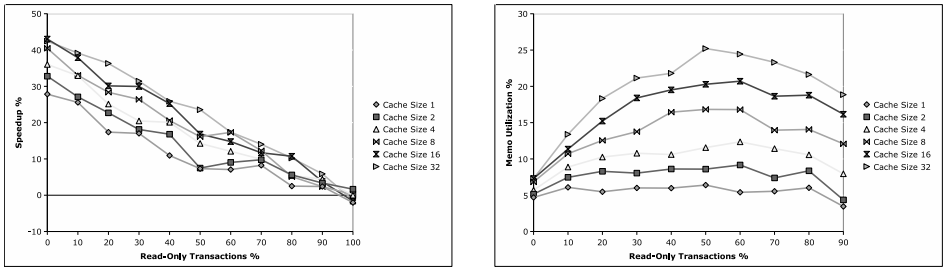
6.1 Results

To measure the effectiveness of our memoization technique, we executed two configurations of the benchmark, and measured overheads and performance by averaging results over ten executions. The *transactional* configuration uses our STM implementation without any memoization. The *memoized transactional* configuration implements partial memoization of aborted transactions. The benchmarks were run on an Intel P4 2.4 GHz machine with one GByte of memory running Gentoo Linux, compiled and executed using MLton release 20051202. Our experiments are not executed on a multiprocessor because the utility of memoization for this benchmark is determined by performance improvement as a function of transaction aborts, and not on raw wallclock speedups.

All tests were measured against a graph of over 1 million nodes. In this graph, there were approximately 280k complex assemblies and 1400K assemblies whose bags referenced one of 100 components; by default, each component contained a parts graph of 100 nodes. Our tests varied two independent variables: the read-only/read-write transaction ratio (see Fig. 6) and part graph size (see Fig. 7). The former is significant because only transactions that modify values can cause aborts. Thus, an execution where all transactions are read-only or which never `retry` cannot be accelerated, but one in which transactions can frequently abort or `retry` offers potential opportunities for memoization. In our experiments, the atomic parts graph (the graph associated with each component) is modified to vary the length of transactions. By varying the number

of atomic parts associated with each component, we significantly alter the number of nodes that each transaction accesses, and thus lengthen or shorten transaction times.

For each test, we varied the maximum number of memos (labeled cache size in the graphs) stored for each procedure. Tests with a small number experienced less memo utilization than those with a large one. Naturally, the larger the size of the cache used to hold memo information, the greater the overhead. In the case of read-only non-aborting transactions (shown in Fig. 6), performance slowdown is correlated to the maximum memo cache size.

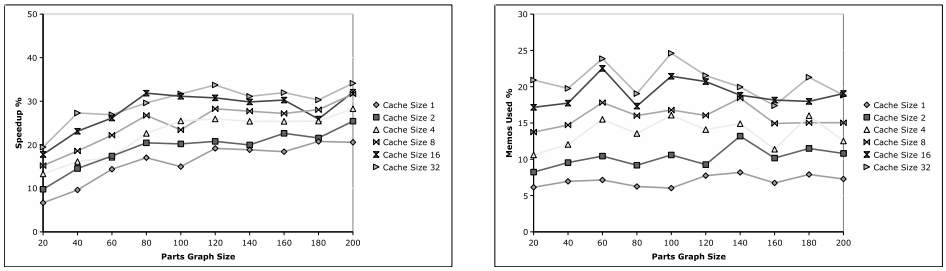


(a) (b)

Fig. 6. (a) presents normalized runtime speedup with a varying read to write ratio. (b) shows the average percent of transactions which are memoizable as read/write ratios change.

Our experiments consider four different performance facets: (a) runtime improvements for transactions with different read-write ratios across different memo cache sizes (Fig. 6(a)); (b) the amount of memoization exhibited by transactions, again across different memo cache sizes (Fig. 6(b)); (c) runtime improvements as a function of transaction length and memo cache size (Fig. 7(a)); and, (d) the degree of memoization utilization as a function of transaction length and memo cache size (Fig. 7). Memory overheads were measured by utilizing MLton's profiler and GC statistics. Memory overheads were proportional to cache sizes and averaged roughly 15% for caches of size 16. Runs with cache sizes of 32 had overheads of 18%.

Memoization leads to substantial performance improvements when aborts are likely to be more frequent. For example, even when the percentage of read-only transactions is 60%, we see a 20% improvement in runtime performance compared to a non-memoizing implementation. The percentage of transactions that utilize memo information is related to the size of the memo cache and the likelihood of the transaction aborting. In cases where abort rates are low, for example when there is a sizable fraction of read-only transactions, memo utilization decreases. This is because a procedure is applied potentially many times, with the majority of applications not requiring memoization because they were not in aborted transactions. Therefore, its memo utilization will be much lower than a procedure in a transaction that aborted once and which was able to leverage memo information when subsequently re-applied.



(a) (b)

Fig. 7. (a) shows normalized runtime speedup compared to varying transactional length. (b) shows the percentage of aborted transactions which are memoizable as transaction duration changes.

To measure the impact of transaction size on performance and utilization, we varied the length of the random traversals in the atomic parts graph. As Fig. 7(a) illustrates, smaller transactions offer a smaller chance for memoization (they are more likely to complete), and thus provide less opportunities for performance gains; larger transactions have a greater chance of taking advantage of memo information. Indeed, we see a roughly 30% performance improvement once the part size becomes greater than 80 when the memo cache size is 16 or 32. As transaction sizes increase, however, the amount of the transaction that is memoizable decreases slightly (Fig. 7(b)). Larger transactions have a higher probability that some part of their traversal has changed and are thus not memoizable. After a certain size, an increase in the traversal length of the atomic parts graph no longer impacts the percent of memos used. This is because the majority of the transaction that is memoizable is found in the initial traversal through the assembly structure, and not in the highly-contented parts components.

As expected, increasing the memoization cache size leads to an increase in both run-time speed up as well as the percent of the transactions that we are able to memoize. Unfortunately, as a result our memoization overheads are also increased both due to the larger amount of memos taken during execution as well as increased time to discover which memo can be utilized at a given call site. Memory overheads increase proportionally to the size of the memo cache.

7 Related Work and Conclusions

Memoization, or function caching [15, 17, 13], is a well understood method to reduce the overheads of function execution. Memoization of functions in a concurrent setting is significantly more difficult and usually highly constrained [6]. We are unaware of any existing techniques or implementations that apply memoization to the problem of optimizing execution for languages that support first-class channels and dynamic thread creation.

Self adjusting mechanisms [2, 3, 1] leverage memoization along with change propagation to automatically alter a program's execution to a change of inputs given an existing execution run. Selective memoization is used to identify parts of the program which

have not changed from the previous execution while change propagation is harnessed to install changed values where memoization cannot be applied. The combination of these techniques has provided an efficient execution model for programs which are executed a number of times in succession with only small variations in their inputs. However, such techniques require an initial and complete run of the program to gather needed memoization and dependency information before they can adjust to input changes.

New proposals [10] have been presented for self adjusting techniques to be applied in a multi-threaded context. However, these proposals impose significant constraints on the programs considered. References and shared data can only be written to once, forcing self adjusting concurrent programs to be meticulously hand crafted. Additionally such techniques provide no support for synchronization between threads nor do they provide the ability to restore to any control point other than the start of the program.

Reppy and Xiao [19] present a program analysis for CML that analyzes communication patterns to optimize message-passing operations. A type-sensitive interprocedural control-flow analysis is used to specialize communication actions to improve performance. While we also use CML as the underlying subject of interest, our memoization formulation is orthogonal to their techniques.

Our memoization technique shares some similarity with transactional events [7]. Transactional events require arbitrary look-ahead in evaluation to determine if a complex composed event can commit. We utilize a similar approach to formalize memo evaluation. Unlike transactional events, which are atomic and must either complete entirely or abort, we are not obligated to discover if an application is completely memoizable. If a memoization constraint cannot be discharged, we can continue normal execution of the function body from the failure point.

References

1. Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An Experimental Analysis of Self-Adjusting Computation. In *PLDI*, pages 96–107, 2006.
2. Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *POPL*, pages 247–259, 2002.
3. Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective Memoization. In *POPL*, pages 14–25, 2003.
4. Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *PLDI*, pages 26–37, 2006.
5. Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The 007 benchmark. *SIGMOD Record*, 22(2):12–21, 1993.
6. Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A Concurrent Logical Framework II: Examples and Applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002.
7. Kevin Donnelly and Matthew Fluet. Transactional Events. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 124–135, 2006.
8. Jim Gray and Andreas Reuter. *Transaction Processing*. Morgan-Kaufmann, 1993.
9. Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: a Benchmark For Software Transactional Memory. In *Eurosys*, 2007.

10. Matthew Hammer, Umut A. Acar, Mohan Rajagopalan, and Anwar Ghuloum. A Proposal for Parallel Self-Adjusting Computation. In *DAMP*, 2007.
11. Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *PPoPP*, pages 48–60, 2005.
12. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software Transactional Memory for Dynamic-Sized Data Structures. In *ACM Conference on Principles of Distributed Computing*, pages 92–101, 2003.
13. Allan Heydon, Roy Levin, and Yuan Yu. Caching Function Calls Using Precise Dependencies. In *PLDI*, pages 311–320, 2000.
14. Richard Kelsey, Jonathan Rees, and Michael Sperber. The Incomplete Scheme 48 Reference Manual for Release 1.1, July 2004.
15. Yanhong A. Liu and Tim Teitelbaum. Caching Intermediate Results for Program Improvement. In *PEPM*, pages 190–201, 1995.
16. MLton. http://www.mlton.org.
17. William Pugh. An Improved Replacement Strategy for Function Caching. In *LFP*, pages 269–276, 1988.
18. William Pugh and Tim Teitelbaum. Incremental Computation via Function Caching. In *POPL*, pages 315–328, 1989.
19. John Reppy and Yingqi Xiao. Specialization of CML Message-Passing Primitives. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 315–326, 2007.
20. John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
21. Michael F. Ringenburg and Dan Grossman. AtomCaml: First-Class Atomicity via Rollback. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 92–104, 2005.
22. Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a High-Performance Software Transactional Memory system for a Multi-Core Runtime. In *PPoPP*, pages 187–197, 2006.
23. Andrew P. Tolmach and Andrew W. Appel. Debuggable Concurrency Extensions for Standard ML. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 120–131, 1991.
24. Lukasz Ziarek, Philip Schatz, and Suresh Jagannathan. Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 136–147, 2006.

On Supporting Parallelism in a Logic Programming System

Vítor Santos Costa[1]

[1]CRACS and DCC-FCUP
Universidade do Porto
Portugal
vsc@dcc.fc.up.pt

Abstract. Logic Programming is a declarative approach to programming where one can specify a problem in a high-level fashion. Several major approaches to implicit and explicit parallelism have been proposed for logic programming in Prolog. But, arguably, the last few years have seen most interest in the explicit parallelization of Prolog.
With the advent of multi-core processors, parallelism is just available. One boring, but useful approach, is bag-of-tasks parallelism. We believe that the challenge facing parallel logic programming is to make all forms of parallelism as boring as possible. To do so, we propose some principles from our experience with previous work in Parallel Logic Programming, discuss how much a Prolog system needs to be adapted to support these principles, and present an application.

1 Introduction

Logic Programming is a declarative approach to programming where one can specify a problem in a high-level fashion. Arguably, Prolog is the most popular logic programming language. Early progress on Prolog compilation, leading to the WAM abstract machine [44], showed Prolog to be useful in a wide variety of practical applications. Prolog and Logic Programming have been widely used ever since, in a surprising large number of diverse applications.

The high-level nature of Logic Programming has made Prolog a natural target for parallelization. Several major approaches have been proposed. In *explicit parallelism* the programmer extends the language with a number of primitives that enable the creation and management of separate tasks. In *implicit parallelism* the Prolog system is largely responsible to detect and exploit the available parallelism [16].

Implicit and explicit parallelism have been well studied in logic programming, but, arguably, the last few years have seen most interest in explicit parallelization. The field of Inductive Logic Programming (ILP), within Machine Learning, has been an example motivation for some of this work. Systems such as April [14] and distributed versions of Aleph [22] were designed to run on clusters and apply MPI [4, 5] in a rather direct fashion. Thread libraries were used to parallelize Aleph in a conventional shared-memory machine. Machine Learning in general

tends to generate computationally demanding tasks, and ILP is particular is highly computationally demanding. In order to support this need, Prolog systems, such as Ciao and YAP, have been adapted to support bag of tasks style execution so that they can exploit massive parallelism in grid systems [6, 12].

```
top - 16:48:38 up 43 days,  1:41,  1 user,  load average: 4.00, 4.00, 4.00
Tasks: 139 total,   6 running, 133 sleeping,   0 stopped,   0 zombie
Cpu(s): 50.0% us,  0.0% sy,  0.0% ni, 50.0% id,  0.0% wa,  0.0% hi,  0.0% si
Mem:  16409824k total, 10484280k used,  5925544k free,   174136k buffers
Swap:  2040244k total,        0k used,  2040244k free,  7525096k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
19577 vitor     25   0  682m 522m 1560 R  100  3.3  19365:53 yap
19581 vitor     25   0  682m 522m 1560 R  100  3.3  19368:01 yap
19580 vitor     25   0  682m 522m 1560 R  100  3.3  19368:44 yap
19582 vitor     25   0  682m 522m 1560 R  100  3.3  19368:24 yap
```

Fig. 1. A Multi-Core in Action

The last few years have seen a major change in this picture. With multi-core processors, *parallelism is just available*. Fig. 1 shows an example of usage of a dual processor machine, with each CPU being four-core, for a total of eight available cores. The machine is also used as a workstation, so only four cores are being used for background tasks [1].

Arguably, such bag-of-tasks parallelism is just *boring*: there is hardly much challenge in launching four processes and waiting for their outcome. But, it is very *useful*, as one is, at least in principle, four times faster (and still has a useful desktop machine), with very little work.

We believe that the challenge facing declarative programming, and we will discuss logic programming here, is to make other forms of parallelism as boring as possible. But to do so, some issues have to be debated first:

- Although there are good reasons to want newer languages, work such as XSB-Prolog and Ciao has shown that incremental progress in current logic programming language design is possible. This makes it the onus of the new language designer(s) to prove that new wine is there, or in other words, that their new approach is widely applicable.
- The last few years have shown the contrast between implicit and explicit parallelism to be largely artificial. Explicit parallelism can be a useful building tool in creating higher-level parallel systems [7]. And implicit parallelism benefits from annotations and other forms of user aid. One can therefore feel rather confident in arguing that there is a continuum of alternatives, and that ideally it should be possible for the programmer to move smoothly in choosing the combination that better suits her or his needs.

[1] unfortunately, this is not the author's machine!

- Work such as KLIC [9] &-Prolog [19], Aurora [25], Muse [1], JAM [11], Andorra-I [41], Penny [26], ACE [29, 28] and the DASWAM [42] addressed research issues and advanced technology, resulting in sophisticated and powerful systems. Which, unfortunately, have proven to be rather hard to maintain. This argues for simpler systems built from reusable-blocks, as in recent work for Ciao [20] and, in a different context, ASP-Prolog [13].
- Logic Programming systems run user tasks, and therefore must interact with the user's environment. This often includes Input/Output and data-base operations, which therefore may be key to efficient execution. It is arguably the case that side-effects have been seen as an obstacle in the race for parallelism and either ignored or set aside [18, 21, 27, 17]. But it does not need to be thus. A good example is data-base support for tabling, which is usually implemented by storing tabled solutions in *tries*. In this case, a parallel implementation can understand the goal of a data-base operation, and exploit parallelism, with excellent results [33]. We believe that it is critical to progress in this direction.
- At the end of the day, it will be the ability to actually run real applications that will decide whether the work will be worthwhile. It has been argued that parallel logic programming had no real applications. This is unfair, as a number of applications have been developed: knowledge-based systems [2], natural language processing [30], multimedia [34], and model checking [32]. More to the point, one can argue that such applications did not include some of the major applications of LP and that they had to compete for scarce parallel resources. As the latter problem transforms from a problem into a motivation, research on the former becomes all the more important.

In a nutshell, we believe that this discussion distills itself into three simple commandments:

- Thou shall use Modular System Construction, so that thou shall be able to maintain and reuse thou code!
- Thou shall Provide High-Level Data Structures, so that the user needs shall be apparent to thou!
- Though shall study and understand Real Applications, so that they shall be the salt of thou work!

Can we apply those commandments in a principled way? We would like to discuss the application of the three commandments but we will focus on the first commandment in this work. We discuss how a specific Prolog system, the YAP Prolog system [37], has been adapted to support parallelism. In a similar fashion to other Prolog systems such as SICStus Prolog [3] and Ciao [20], YAP included some support for shared-memory implicit parallelism, in this case or-parallelism. Building on this support, it was possible to implement a thread library that supports explicit parallelism. On a different vein, the system has also been used for distributed programming and for grid style computation. We compare the costs of the three approaches, and study how our simple commandments can be obeyed, if at all.

2 The YAP Prolog System

The YAP Prolog system was originally developed by Luís Damas and Vítor Santos Costa towards being a high-performance Prolog system [37]. The system includes a number of components, described in Fig. 2. We distinguish three particularly complex components: the abstract emulator (marked as red), the compiler (marked as blue), and the memory management routines, including the garbage collector (marked as gray). Edges show how components depend on each-other.

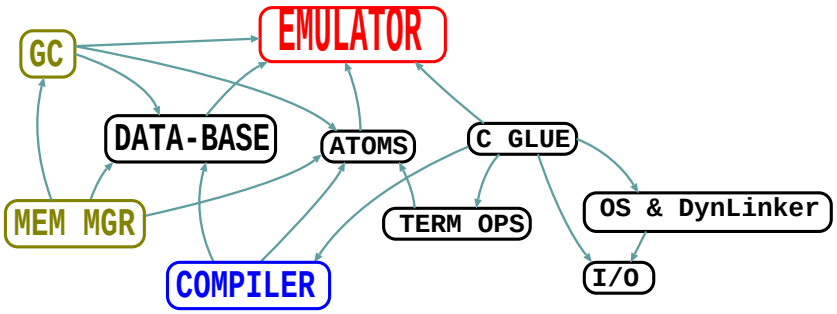


Fig. 2. The Structure of the YAP System

The core component of the system is a WAM abstract machine emulator [36]. Currently, the emulator implements close to 400 different instructions. The emulator interacts directly with three key components:

1. The *C-Glue* provides the interface between the abstract engine and most libraries. It relies on a a set of abstract types, such as Constant, Compound Term, Term, Atom, Functor and a set of constructors and destructors. These primitives are used by all components of the system.
2. The *Atom-Table* is organized as a hash-table and maintains all constants in the system. As in traditional LISP systems [15], it is used as a starting point for all other property lookup.
3. The *Data-Base* supports compiled code, both dynamic and static, and a term database, accessed through the `record` family of built-ins. There are also smaller data-bases: an operator data-base, and a small constant data-base.

These components require dynamically allocated memory:

1. The *Garbage Collector* [8] and the closely-associated *Stack Shifter* can interrupt the engine to compress stacks, clean up dead code, and expand memory regions.

2. The Memory Allocator provides memory allocation and deallocation services for the system. YAP includes three different allocators: **(i)** the original allocator asks the system for a large chunk of memory, allocates big chunks for the stacks, and manages the rest of memory with a greedy algorithm; **(ii)** the default one allocates a huge chunk of memory and expands it dynamically, but uses the Doug Lea allocator to manage memory [23]; **(iii)** last, YAP can just use the standard library routines for memory allocation.
   We have found out that the greedy allocator will not work well on large applications. The Doug Lea allocator has allocator better performance, and is in fact traditionally used by several system libraries.

The full functionality of the system requires extra modules:

1. Input/Output operations are obviously required for the system to be useful.
2. Term operations such as term comparison, are important in actual applications.
3. The Operating System interface allows access to a number of important features. Access to the Dynamic Linker allows run-time extensibility.

Finally, the YAP compiler supports clause-level compilation and dynamic compilation of indices [40].

3 Supporting Parallelism in YAP

Our first commandment says that one should be able to extend the system modularly. In other words, ideally we should implement parallelism by extending the system with a new module, and would not need to rewrite code on the remaining of the system. Unfortunately, and as often is the case, such a commandment may be hard to obey. YAP is an ideal platform to study the problem as it has been used to implement a large number of different approaches. Next, we discuss some of these approaches: we shall start from the methodologies associated with coarser grain-size.

Grid-Support With Condor YAP can be run in grid systems using two approaches: in the common *as is* approach the system is just transfered without changes; otherwise, the system can be adapted to support libraries such as condor [43]. The condor library is particularly interesting because it allows "transparent" check-pointing and migration of jobs, which may become useful as idle cores in networks of workstations become more and more available [12]. Fig. 3 shows where changes were necessary: notice that these changes correspond support the `universe` condor environment on a circa 2004 version of condor.

The changes are required to operate under the more limited functionality available in the condor universe. They essentially drop some of the memory manager functionality, as YAP now **has** to use the standard library; drop dynamic linking as condor requires static linking; and change some time and file access primitives, again due to limitations in the condor programming environment.

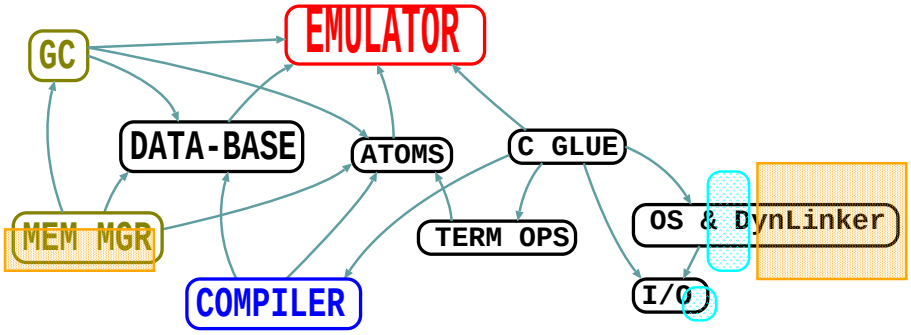


Fig. 3. The Structure of the YAP System with Condor support: square boxes correspond to dropped functionality, round boxes correspond to changes

Arguably, such changes are mostly modular. On the other hand, they are more about *dropping* modules than actually extending a system with new modules. In that sense, they can be seen as a minor, but necessary, sin.

Distributed Processing with MPI The YAP system includes support for two different MPI libraries: MPICH [4] and LAM [5]. The two interfaces were developed independently but operate under similar principles. They provide a low-level interface that allows one to use basic MPI functionality, while exporting and importing Prolog terms as messages. The implementation is shown in Fig. 4.

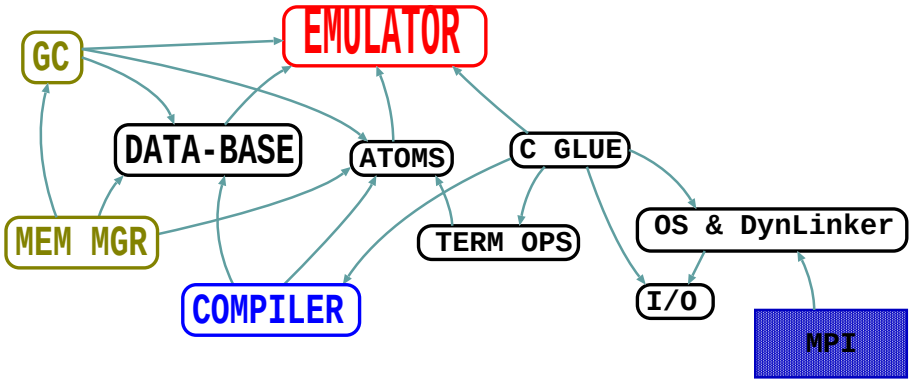


Fig. 4. The Structure of the YAP System with MPI support: the grey square box corresponds to the new functionality

As the picture shows, both interfaces operate as a new module that links to the system through the C-interface. The interfaces did not require changes

to Prolog, although they would benefit from functionality in the term libraries. Arguably, such an implementation is not the most efficient, but it is the easier to update and maintain, and follows perfectly our first commandment.

Threads The YAP system includes support for Posix p-threads in the style of the SWI-Prolog thread library [45]. In this approach, threads run on separate stacks but share access to the data-base. The implementation is presented in Fig. 5.

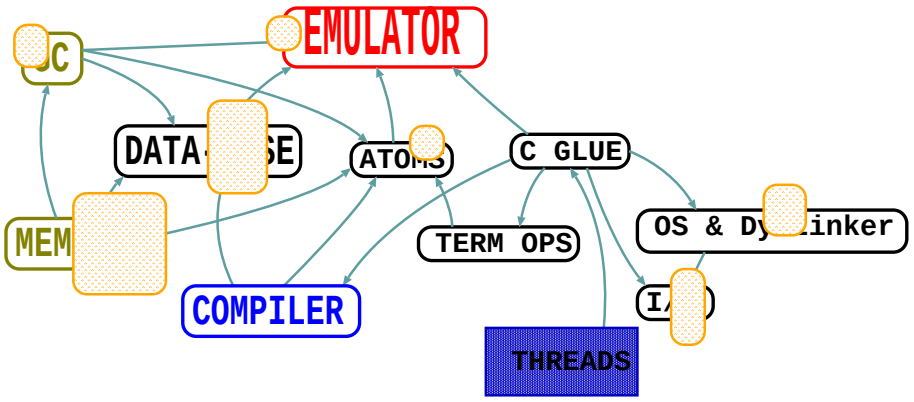


Fig. 5. The Structure of the YAP System with thread support: the grey square box corresponds to the new functionality, the round corner boxes correspond to significant changes in the pre-existing code

As the picture shows, supporting threads requires major changes throughout the system. Most of these changes have to do with the need to synchronize access to the data-base and the Input/Output. In some more detail:

1. One needs a new module for threading. Notice that this module is now plugged in more closely to the system, as communication is expected to be quite more intensive.
2. One needs major changes to the Data-Base, due to the need to synchronize access to dynamic structures, such as the index trees [40] and dynamic predicates. Such data-structures are quite important in large programs. These changes then ripple down to the garbage collector and to the emulator.
3. The memory allocator needs to support concurrency and multiple stacks. In fact, the easiest solution is to rely on the system library, as for condor.
4. The Atom-Table needs to support concurrent access.

Most of the complexity stems from the concurrent accesses to the data-base. The changes are quite intrusive and hard to debug, as it is often the case with

concurrent systems. Memory management is a second problem: threads require extra memory to support locks, that depending on the grain size, may be quite frequent. Note that several approaches are possible, ranging from a big central lock to fine–grained access with specialized data-structures, such as read-write locks. Supporting threads is therefore not modular, but we have been able to provide most of the functionality in the non-threaded system (except for the atom garbage collector). As threads may be used to build other primitives, they may arguably be a necessary sin.

Native Or-Parallelism YAP includes code for a native implementation of three different models of or-parallelism: COWL, stack-copying, and the Sparse Binding Arrays [39]. The implementation is further complicated by the need to support tabling [32]. Although the implementation is not currently being actively developed, it is still in the code. The implementation is presented in Fig. 3.

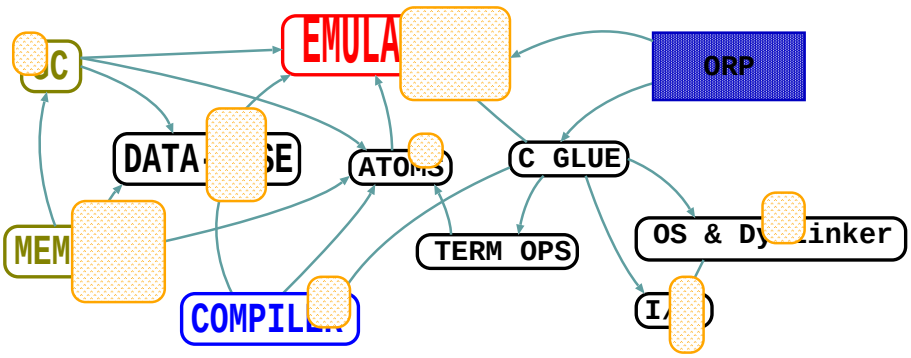


Fig. 6. The Structure of the YAP System with native or-parallelism: the grey square box corresponds to the new functionality, the round corner boxes correspond to significant changes in the pre-existing code

It should be clear that there is extensive overlap between native or-parallelism and threads: this is because both implementations can perform concurrent access to the data-base and concurrent I/O, although concurrent access in or-parallel systems may be unnecessary if one requires a sequential order [18, 17]. Or-parallel requires major changes to:

1. The memory allocator needs several low-level optimizations to support or-parallelism, namely for stack-copying.
2. The emulator needs several new instructions to run or-parallelism. Furthermore, some useful invariants in the sequential system will now be broken. The trailing mechanism is particularly affected.

The actual impact of these changes depends on the particular model, with the process-based model of the COWL being the least intrusive [35] and the shared-

memory approach of the SBA being the most intrusive [10], as it requires to share the stacks and to create private images. All the models do require new instructions and major changes to key data-structures, such as choice-points.

3.1 Evaluation

Table 1 shows how pervasive the changes are in YAP's `C`-code. We use Nuno Fonseca's LAM interface in this study. The first column shows the number of `define` directives needed, and the second column shows how many files were affected (out of a total of 104). The third column shows how much new code was needed for the new functionality. Up to a third of all system files (including most header files) need to be changed in some way to support threads or implicit parallelism. The changes to support *condor* are much less extensive, and they mostly have to with the need to support a different memory allocator. As expected, although or-parallelism requires more changes than threads, the changes are largely on the same files. In fact, around 141 defines are shared between or-parallelism and threads.

	#`define` needed	Files Affected	New Lines
`CONDOR`	3 (47)	3	0
`MPI`	0	0	1652
`! THREADS`	231 (47)	33	1280
`ORP`	680	33	4222

Table 1. Number of `define` used to separate parallelism specific code, number of files where such defines were used and extra code needed (including comments). Changes required to use the system allocator are in brackets.

It is unsurprising that implementing parallelism requires more extra code. Of this code, about 300 lines of code implement a locking library which is shared with the thread implementation. So the actual cost of writing the thread library is under a thousand lines of code, mostly on packing and unpacking arguments when calling `p-thread` functions. This is smaller than the MPI interface, as the latter has to copy and receive terms from messages.

Is there a runtime cost for this functionality? Previous reports indicate costs on the order of 5% for applications with SWI-Prolog [45], and with or-parallelism in YAP [32]. An interesting worst possible situation is the case where almost every operation needs to be protected by a lock. In order to study this setting, we compare YAP without and with thread support on small, data-base intensive examples. Our comparison was run on a dual-CPU machine, where each CPU is a 4 core Intel(R) Xeon(R) CPU E5345 running at 2.33GHz. The machine has 16GB memory, and runs RedHat Entreprise Linux release 4 with kernel 2.6.9. We use Yap-5.1.2 compiled for the `x86_64` architecture. The machine was connected

to the network, and the file-system was NFS; the experiments were performed through `ssh` access, the machine was otherwise idle. The tests are as follows: **(i)** `t1` accesses a dynamic predicate with a single fact; this requires holding a lock: **(ii)** `t2` asserts and retracts a dynamic predicate with a single fact; **(iii)** `t3` asserts a fact of the form `t3(RandomNumber)`; **(iv)** `t4` retracts a fact of the form `t4(RandomNumber)`; **(v)** `t5` asserts a fact where the argument is a list of length up to 10 and branching factor up to 500; **(vi)** `t6` asserts a fact where the argument is a list of length up to 10 and branching factor up to 2 (hence there will be more repeated facts).

Results are shown in Table 2. We compare two versions with threads: in the first version, threads are implemented using `C`-code from the Linux kernel that uses hardware instruction. In the second version, we call the `P-Thread` routines. The results show that there is indeed an overhead, that the overhead can be very large if the application just updates the data-base, but that it tends to reduce as operation cost increases. The results are also somewhat disappointing in that we would expect locking on the `P-Thread` library to have substantially improved in newer thread libraries. This does not seem to be the case: locking performance is still very much under par.

	t1	t2	t3	t4	t5	t6
NO-THREADS	11	642	298	792	1354	1379
THREADS-USER LOCKING	11	1144	480	1558	1520	1527
THREADS-PTHREAD LOCKING	25	1717	767	1851	2116	2163

Table 2. Running time for 500000 iterations of simple data-base access predicates, using a native implementation, user code for locking, and the p-thread library locking routines.

The results also show that locking can indeed decrease system performance. The data-base is controlled by reader-writer locks, which are called at 263 points in the code. Standard locks are called at 113 points in the code. Locking is required whenever accessing a dynamic predicate. YAP does not need to lock the indexing code for static procedures because it is write once [40].

3.2 Parallel Execution

The usefulness of the techniques discussed here clearly depends on how much the underlying computer architecture can support them. To validate whether it is worthwhile to exploit these machines, we experimented a number of simple benchmarks on the multi-core machine. The 3 experiments include just accessing a static and a dynamic fact, building a long list, and randomly accessing a very large compound term through `arg/3`. Experiments are run in by *executing the same code at N threads*: the only communication is when accessing the read-lock that protects the dynamic predicate in the second experiment. All

other experiments have *no* synchronization within YAP. The ideal result would be constant-time, and except for the second benchmark, slowdowns should be caused by limitations in the multi-core architecture.

Cores	1	2	3	4	5	6	7	8
`a(1)`	1771	1780	1795	1794	1772	1793	1798	1833
`a(1)` (dynamic)	1635	1623	1636	1629	1617	1625	1623	1684
`mklist(10000,_)`	435	438	437	436	443	443	450	491
`mklist(100000,_)`	4540	4628	4660	4438	4737	5071	5970	6658
`mklist(1000000,_)`	473232	47605	47692	46674	50014	55351	62401	70228
`arg(Rand,10000,_)`	693	1523	6914	12885	13480	14469	14297	15235
`arg(Rand,100000,_)`	866	2765	7713	14108	15420	16646	16137	17245
`arg(Rand,1000000,_)`	1075	3203	7869	14688	16302	16978	17275	18469

Table 3. Running time in msec for 20,000,000 iterations of a simple query, for 200 iterations of the `mklist/2` predicate, and for 2,000,000 random accesses to a large term. Every thread repeats the original task, hence ideal performance would be constant time.

The results show a complex story. First, they show almost perfect parallelism for the simple query: in this case, the cores can happily process away in their local caches, and performance is excellent even when all cores are in use. Second, they show that a limited amount of synchronization has no impact on system performance: the static and dynamic versions of the code execute in much the same way.

The results for `mklist` show excellent performance for the two list smaller lists, with 160KB and 1600KB. Performance drops somewhat for larger number of cores when we construct the 16MB list. In this case, we have a slowdown of 60% when the 8 cores construct the 8MB lists in parallel.

The `arg/3` results were the most surprising. The simple benchmark was chosen as an example of stressing the cache system. It does its task well. With two cores, it is just faster to run the benchmarks sequentially than to run them in the separate cores. The picture grows worse for 3 cores and for 4 cores: in fact, adding the 4th core consistently halves performance! It is also interesting that after 4 cores performance degrades more gracefully (maybe because contention is now bad enough). The size of the term is not particularly important: the effects are very clear with a 800KB terms, and there is only a small price to access an 80MB term.

4 Future Work

We started this work assuming that multi-cores will make parallel programming boring, and hoping that parallel logic programming would follow. We will have to wait: multi-core architectures are complex, with memory performance far more

critical than for traditional shared memory machines. Programming these machines may require understanding memory access patterns, admittedly a harder task in the context of declarative languages.

This leads us to commandment number two: providing high-level data structures that are well understood and that can be profiled and analised with confidence may be what makes parallel logic programming successful. And this in turn leads to commandment number three, and to the question we should have started from: what do logic programmers need?

There is not a single answer to this question. Our experience shows that often people just want to run similar tasks. In this case, the question is "how does Prolog access memory?", and this is an hard problem by itself [24]. But, one can go one step further by looking at actual applications. In the author's case, at the time of writing this paper, he was interested in two main applications in the area of Statistical Relational Learning.

The Problog language combines logic with probabilities by saying that a clause may be true [31]. The probability of a goal is evaluated by combining the probabilities of all paths that prove the goal. This is very close to traditional or-parallelism, except that successful (and interrupted) proofs must be stored away. Doing this sequentially would kill parallelism, But there is no real reason to do so. Solving this problem is thus a question of designing a data-structure that can store proofs and that allows concurrent updates, such as, say, a trie [33]!

The CLP($\mathcal{BN}$) language can be used to specify graphical models, such as Hidden Markov Models [38]. One interesting query in these models is to find the most likely explanation to a sequence of observations. The Viterbi algorithm is the main tool for this task. The algorithm implements dynamic programming and proceeds in two steps. Computation is dominated by the forward step where it steps across all nodes in the graph following a dominance order. The first CLP($\mathcal{BN}$) implementation used constraints to represent the node, topologically sorted them, and then run the algorithm. This was elegant, but expensive. A recent implementation does not generate the nodes. Instead, it generates a set of instructions describing the graph and applies them to every element in the sequence. Even so, the application can take seconds for larger networks. The application is an example of data-flow parallelism, but we have observed that for large networks chunks tend to be somewhat independent, so sub-computation can proceed in parallel as long as we have shared access to the state. Can we exploit independent and-parallelism in this context?

It would be a mistake to ignore the huge amount of work in parallel logic programming: much progress was made, and many lessons were learned. We understand the main issues in implicit parallelism, and we know where the main pitfalls wait for us. We should thus start from the lessons we learned. And we should build something new, and better.

Acknowledgments

The author gratefully acknowledges the support of the project STAMPA, FCT Project PTDC/EIA/67738/2006. The work has been partially supported by

funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC. The work was motivated by discussions at the CICLOPS07 pannel: the author thanks B. Demoen, D. S. Warren, E. Pontelli, F. Silva, G. Gupta, I. Dutra, M. Hermenegildo, R. Rocha, and S. Abreu for sharing their ideas.

References

1. K. A. M. Ali and R. Karlsson. The Muse approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162 (or 129–160??), Apr. 1990.
2. K. A. M. Ali and R. Karlsson. OR-Parallel Speedups in a Knowledge Based System: on Muse and Aurora. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 739–745, ICOT, Japan, 1992. Association for Computing Machinery.
3. J. Andersson, S. Andersson, K. Boortz, M. Carlsson, H. Nilsson, T. Sjoland, and J. Widén. SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science, November 1997. SICS Technical Report T93-01.
4. G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. Mpich-v: toward a scalable fault tolerant mpi for volatile nodes. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.
5. G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.
6. M. Carro and M. Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *1999 International Conference on Logic Programming*, pages 320–334. MIT Press, Cambridge, MA, USA, November 1999.
7. A. Casas, M. Carro, and M. Hermenegildo. Towards High-Level Execution Primitives for And-parallelism: Preliminary Results. In *CICLOPS 2007: 7^{th} Colloquium on Implementation of Constraint and LOgic Programming Systems*, pages 102–116, Porto, 2007.
8. L. F. Castro and V. Santos Costa. Understanding Memory Management in Prolog Systems. In *Proceedings of Logic Programming, 17th International Conference, ICLP 2001*, volume 2237 of *Lecture Notes in Computer Science*, pages 11–26, Paphos, Cyprus, November 2001.
9. T. Chikayama, T. Fujise, and H. Yashiro. A portable and reasonably efficient implementation of KL1. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, page 833, Budapest, Hungary, 1993. The MIT Press.
10. M. E. Correia, F. Silva, and V. Santos Costa. The SBA: Exploiting orthogonality in OR-AND Parallel Systems. In *Proceedings of the 1997 International Logic Programming Symposium*, pages 117–131. MIT Press, October 1997. Also published as Technical Report DCC-97-3, DCC - FC & LIACC, Universidade do Porto, April, 1997.
11. J. A. Crammond. The abstract machine and implementation of parallel parlog. *New Generation Computing*, 10(4):385–422, 1992.
12. I. C. Dutra, D. Page, V. Santos Costa, J. W. Shavlik, and M. Waddell. Towards automatic management of embarassingly parallel applications. In *Proceedings of*

Europar 2003, volume 2790 of *Lecture Notes in Computer Science*, pages 509–516, Klagenfurt, Austria, August 2003. Springer Verlag.

13. O. El-Khatib, E. Pontelli, and T. C. Son. Integrating an answer set solver into prolog: Asp-prolog. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Logic Programming and Nonmonotonic Reasoning, 8th International Conference, LPNMR 2005, Diamante, Italy, September 5-8, 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*, pages 399–404. Springer, 2005.
14. N. A. Fonseca, F. Silva, and R. Camacho. April - An Inductive Logic Programming System. In F. M, V. W, K. B, and L. A, editors, *Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA06)*, volume 4160 of *Lecture Notes in Artificial Intelligence*, pages 481–484, Liverpool, September 2006. Springer-Verlag.
15. R. P. Gabriel. *Performance and evaluation of Lisp systems*. MIT Press, 1985.
16. G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):1–131, 2001.
17. G. Gupta and V. Santos Costa. Cuts and Side-Effects in And-Or Parallel Prolog. *Journal of Logic Programming*, 27(1):45–71, April 1996.
18. B. Hausman, A. Ciepielewski, and A. Calderwood. Cut and Side-Effects in Or-Parallel Prolog. In *International Conference on Fifth Generation Computer Systems 1988*, pages 831–840. ICOT, 1988.
19. M. V. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.
20. M. V. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the ciao system preprocessor). *Sci. Comput. Program.*, 58(1-2):115–140, 2005.
21. L. V. Kalé, D. A. Padua, and D. C. Sehr. OR-Parallel execution of Prolog with side effects. *The Journal of Supercomputing*, 1988.
22. S. T. Konstantopoulos. A data-parallel version of Aleph. In R. Camacho and A. Srinivasan, editors, *Proc. of the Workshop on Parallel and Distributed Computing for Machine Learning, ECML/PKDD 2003*, 2003.
23. D. Lea. *A Memory Allocator*.
24. R. Lopes, L. F. Castro, and V. Santos Costa. From Simulation to Practice: Cache Performance Study of a Prolog System. In *ACM SIGPLAN Workshop on Memory System Performance, Berlin, Germany*, June 2002. SIGPLAN Notices vol 38(2), February 2003, pages 56–64.
25. E. Lusk, R. Butler, T. Disz, R. Olson, R. A. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.
26. J. Montelius and K. A. M. Ali. An And/Or-Parallel Implementation of AKL. *New Generation Computing*, 14(1), 1996.
27. K. Muthukumar and M. V. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 80–97. MIT Press, June 1989.
28. E. Pontelli and G. Gupta. Data and-parallel logic programming in &ace. In *7th IEEE Symposium on Parallel and Distributed Processing*. IEEE Computer Society, 1995.

29. E. Pontelli, G. Gupta, and M. V. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.
30. E. Pontelli, G. Gupta, J. Wiebe, and D. Farwell. Natural language multiprocessing: A case study. In *AAAI/IAAI*, pages 76–82, 1998.
31. L. D. Raedt, A. Kimmig, and H. Toivonen. Problog: A probabilistic prolog and its application in link discovery. In M. M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2462–2467, 2007.
32. R. Rocha, F. Silva, and V. S. Costa. On Applying Or-Parallelism and Tabling to Logic Programs. *Theory and Practice of Logic Programming Systems*, 5(1-2):161–205, 2005.
33. R. Rocha, F. Silva, and V. Santos Costa. Achieving Scalability in Parallel Tabled Logic Programs. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPPDPS02), Fort Lauderdale, Florida, USA*, April 2002.
34. S. W. Ryan and A. K. Bansal. A scalable distributed multimedia knowledge retrieval system on a cluster of heterogeneous high performance architectures. *International Journal on Artificial Intelligence Tools*, 9(3):343–367, 2000.
35. V. Santos Costa. Cowl: Copy-on-write for logic programs. In *Proceedings of the IPPS/SPDP99*, pages 720–727. IEEE Computer Press, May 1999.
36. V. Santos Costa. Optimising bytecode emulation for prolog. In *LNCS 1702, Proceedings of PPDP'99*, pages 261–267. Springer-Verlag, September 1999.
37. V. Santos Costa, L. Damas, R. Reis, and R. Azevedo. *YAP User's Manual*, 2002. http://www.ncc.up.pt/~vsc/Yap.
38. V. Santos Costa, C. D. Page, and J. Cussens. *Probabilistic Inductive Logic Programming*, chapter CLP($\mathcal{BN}$): Constraint Logic Programming for Probabilisti c Knowledge. Springer-Verlag, 2007. (to appear).
39. V. Santos Costa, R. Rocha, and F. Silva. Novel Models for Or-Parallel Logic Programs: A Performance Analysis. In *Proceedings of EuroPar2000, LNCS 1900*, pages 744–753, September 2000.
40. V. Santos Costa, K. Sagonas, and R. Lopes. Demand-driven indexing of prolog clauses. In V. Dahl and I. Niemelä, editors, *Proceedings of the 23rd International Conference on Logic Programming*, volume 4670 of *Lecture Notes in Computer Science*, pages 305–409. Springer, 2007.
41. V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming PPOPP*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.
42. K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3), 1996.
43. D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.
44. D. H. D. Warren. *Applied Logic—Its Use and Implementation as a Programming Tool*. PhD thesis, Edinburgh University, 1977. Available as Technical Note 290, SRI International.
45. J. Wielemaker. Native preemptive threads in swi-prolog. In C. Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 331–345. Springer, 2003.

Toward a parallel implementation of Concurrent ML

John Reppy and Yingqi Xiao

University of Chicago

Abstract. Concurrent ML (CML) is a high-level message-passing language that supports the construction of first-class synchronous abstractions called events. This mechanism has proven quite effective over the years and has been incorporated in a number of other languages. While CML provides a concurrent programming model, its implementation has always been limited to uniprocessors. This limitation is exploited in the implementation of the synchronization protocol that underlies the event mechanism, but with the advent of cheap parallel processing on the desktop (and laptop), it is time for Parallel CML.
We are pursuing such an implementation as part of the Manticore project. In this paper, we describe a parallel implementation of Asymmetric CML (ACML), which is a subset of CML that does not support output guards. We describe an optimistic concurrency protocol for implementing CML synchronization. This protocol has been implemented as part of the Manticore system.

1 Introduction

Concurrent ML (CML) [1, 2] is a statically-typed higher-order concurrent language that is embedded in Standard ML [3]. CML extends SML with synchronous message passing over typed channels and a powerful abstraction mechanism, called *first-class synchronous operations*, for building synchronization and communication abstractions. This mechanism allows programmers to encapsulate complicated communication and synchronization protocols as first-class abstractions, which encourages a modular style of programming where the actual underlying channels used to communicate with a given thread are hidden behind data and type abstraction. CML has been used successfully in a number of systems, including a multithreaded GUI toolkit [4], a distributed tuple-space implementation [2], and a system for implementing partitioned applications in a distributed setting [5]. The design of CML has inspired many implementations of CML-style concurrency primitives in other languages. These include other implementations of SML [6], other dialects of ML [7], other functional languages, such as HASKELL [8], SCHEME [9], and our own MOBY language [10], and other high-level languages, such as JAVA [11].

One major limitation of CML is that its implementation is *single-threaded* and cannot take advantage of multicore or multiprocessor systems.[1] We are incorporating the CML concurrency primitives into the functional parallel-programming language *Manticore* [12, 13], so this limitation must be addressed. In this paper, we take a major step in that direction by describing a parallel implementation of a subset of CML, which

[1] In fact, almost all of the existing implementations of events have this limitation.

```
type 'a event

val choose : ('a event * 'a event) -> 'a event
val wrap : 'a event * ('a -> 'b) -> 'b event
val guard : (unit -> 'a event) -> 'a event
val withNack : (unit event -> 'a event) -> 'a event

val sync : 'a event -> 'a

val never : 'a event
val always : 'a -> 'a event

type 'a chan
val recvEvt : 'a chan -> 'a event
val sendEvt : ('a chan * 'a) -> unit event
```

Fig. 1. The core features of CML

we call *Asymmetric Concurrent ML* (ACML). This subset of CML includes the full set of CML combinators, but does not support *output guards* (*i.e.*, send operations in a choice). We try to provide both an intuitive explanation of the synchronization protocol that underlies ACML, as well as enough of the nitty-gritty details to help other implementors. Because of space constraints, much of the implementation is omitted, but an extended version of this paper will be available as technical report [14].

2 A CML overview

Concurrent ML is a higher-order concurrent language that is embedded into Standard ML [1, 2]. It supports a rich set of concurrency mechanisms, but for purposes of this paper we focus on the core mechanisms of communication and events, which are shown in Figure 1. Communication in CML is based on synchronous message passing on typed channels. Because channels are synchronous, both the send and receive operations are blocking.

To support more complicated interactions, CML provides *event values*, which are first-class synchronous abstractions. Base events constructed by `sendEvt` and `recvEvt` describe simple communications on channels. There are also two special base-events: `never`, which is never enabled and `always`, which is always enabled for synchronization. These events can be combined into more complicated event values using the event combinators:

- Event wrappers (`wrap`) for post-synchronization actions.
- Event generators (`guard` and `withNack`) for pre-synchronization actions and cancellation (`withNack`).
- Choice (`choose`) for managing multiple communications. In CML, this combinator takes a list of events as its argument, but we restrict it to be a binary operator here. Choice of a list of events can be constructed using `choose` as a "cons" operator and `never` as "nil."

```
type 'a queue

val queue : unit -> 'a queue
val isEmptyQ : 'a queue -> bool
val enqueue : ('a queue * 'a) -> unit
val dequeue : 'a queue -> 'a option
```

Fig. 2. Specification of queue operations

To use an event value for synchronization, we apply the `sync` operator to it.

Event values are pure values similar to function values. When the `sync` operation is applied to an event value, a dynamic instance of the event is created, which we call a *synchronization event*. A single event value can be synchronized on many times, but each time involves a unique synchronization event.

In this paper, we describe an implementation ACML, which differs from the interface in Figure 1 in that it does not have the `sendEvt` event constructor. Instead, sending a message is supported using the function

```
val send : ('a chan * 'a) -> unit
```

This function is still blocking, but does not support sending a message in a choice context.

3 Preliminaries

We present our implementation using SML syntax with a few extensions. To streamline the presentation, we elide several aspects of the actual implementation, such as thread IDs and processor affinity.

3.1 Queues

Our implementation uses queues to track pending messages and waiting threads in channels. We omit the implementation details here, but give the interface to the queue operations that we use in Figure 2. These operations have the expected semantics.

3.2 Threads and thread scheduling

As in the uniprocessor implementation of CML, we use first-class continuations to implement threads and thread-scheduling. The continuation operations have the following specification:

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
val throw  : 'a cont -> 'a -> 'b
```

We represent the state of a suspended thread as a `unit` continuation

```
type thread = unit cont
```

The interface to the scheduling system is represented by two atomic operations:

```
val enqueueRdy : thread -> unit
val dispatch : unit -> 'a
```

The first enqueues a ready thread in the scheduling queue and the second transfers control to the next ready thread in the scheduler queue.

3.3 Compare and swap

Our implementation also relies on the atomic *compare-and-swap* instruction. We use the following SML specification for this operation:

```
val cas  : ('a ref * 'a * 'a) -> 'a
```

Note that `cas` does not follow the SML equality semantics in that it performs pointer equality. With this operation, we build spinlocks:

```
val spinLock   : bool ref -> unit
val spinUnlock : bool ref -> unit
```

For purposes of this paper, we assume that threads are not preempted, so spinlocks are a reasonable locking mechanism.

4 A parallel implementation of PCML

Our parallel implementation is based on a core subset of the CML event operations, called *Primitive CML* (PCML). This subset has an event type with a minimal set of combinators, a condition-variable type used for signaling, and support for channels with input events. The signature of PCML is given in Figure 3. Note that unlike full CML (see Figure 1), there are no `guard` or `withNack` combinators. As we discuss in Section 5, these can be implemented on top of PCML.

4.1 The synchronization protocol

The heart of the implementation is the protocol for synchronization on a choice of events. This protocol is split between the `sync` operator and the base-event constructors (*e.g.*, `waitEvt` and `recvEvt`). Each base event is represented by a record of three functions: `pollFn`, which tests to see if the base-event is enabled (*e.g.*, there is a message waiting); `doFn`, which is used to synchronize on an enabled event; and `blockFn`, which is used to block the calling thread on the base event. In the single-threaded implementation of CML [15, 2], we rely heavily on the fact that `sync` is executed as an atomic operation. The single-threaded protocol is as follows:

1. Poll the base events in the choice to see if any of them are enabled. This phase is called the *polling phase*.

```
signature PRIM_CML =
  sig

  (* events *)
    type 'a evt
    val never : 'a evt
    val always : 'a -> 'a evt
    val choose : ('a evt * 'a evt) -> 'a evt
    val wrap : 'a evt * ('a -> 'b) -> 'b evt
    val sync : 'a evt -> 'a

  (* condition variables *)
    type  cvar
    val new : unit -> cvar
    val set : cvar -> unit
    val waitEvt : cvar -> unit evt

  (* channels *)
    type 'a chan
    val channel : unit -> 'a chan
    val recvEvt : 'a chan -> 'a evt
    val send : ('a chan * 'a) -> unit

  end
```

Fig. 3. Primitive CML

2. If one or more base events are enabled, pick one and synchronize on it using its `doFn`. This phase is called the *commit phase*.
3. If no base events are enabled we execute the *blocking phase*, which has the following steps:
   (a) Enqueue a continuation for the calling thread on each of the base events using its `blockFn`.
   (b) Switch to some other thread.
   (c) Eventually, some other thread will complete the synchronization.

We use the term *synchronization setup* for steps 1, 2, and 3(a) of this protocol.

Because the implementation of `sync` is atomic, the single-threaded implementation does not have to worry about the state of a base event changing between when we poll it and when we invoke the `doFn` or `blockFn` on it. In a parallel implementation, however, the global lock would be a bottleneck, so we must design a more complicated protocol. This design is further constrained by the fact that a given event may involve multiple occurrences of the same event. For example, the following code nondeterministicly tags the message received from `ch` with either `1` or `2`:

```
sync (choose (
  wrap (recvEvt ch, fn x => (1, x)),
  wrap (recvEvt ch, fn y => (2, y))
))
```

We must also avoid deadlock when multiple threads are simultaneously attempting communication on the same channel. For example, if thread P is executing

```
sync (choose (recvEvt ch1, recvEvt ch2))
```

at the same time that thread Q is executing

```
sync (choose (recvEvt ch2, recvEvt ch1))
```

we have a potential deadlock if the implementation of `sync` attempts to hold a lock on both channels simultaneously (*i.e.*, where P holds the lock on `ch1` and attempts to lock `ch2`, while Q holds the lock on `ch2` and attempts to lock `ch1`).

Our approach to avoiding these pitfalls is to use an *optimistic* protocol that does not hold a lock on more than one channel at a time and avoids locking whenever possible. The basic protocol has a similar structure to the sequential one described above, but it must deal with the fact that the state of a base event can change before the synchronization setup is complete. This fact means that the commit phase may fail and that the blocking phase may commit. The parallel synchronization protocol is as follows:

- The protocol starts with the polling phase, which is done in a *lock-free* way.
- The If one or more base events are enabled, pick one and *attempt* to synchronize on it using its `doFn`. This attempt may fail because of changes in the base-event state since the polling was done.
- If there are no enabled base events (or all attempts to synchronize failed), we enqueue a continuation for the calling thread on each of the base events using its `blockFn`. When blocking the thread on a particular base event, we may discover that synchronization is now possible, in which case we can synchronize immediately.

This design is guided by the goal of minimizing synchronization overhead and maximizing concurrency.

4.2 The PCML event type

A primitive-event value is represented as a binary tree, where the internal nodes represent choice and the leaves represent single synchronous operations. This canonical representation of events relies on the following equivalences:

$$\texttt{wrap}(\texttt{wrap}(ev, g), f) = \texttt{wrap}(ev, f \circ g)$$
$$\texttt{wrap}(\texttt{choose}(ev_1, ev_2), f_1) = \texttt{choose}(\texttt{wrap}(ev_1, f), \texttt{wrap}(ev_2, f))$$

We use this equivalence to maintain a canonical representation of events as trees in which the leaves are wrapped base-event values and the interior nodes are choice operators. Figure 4 illustrates the mapping from a nesting of `wrap` and `choose` combinators to its canonical representation.

Another issue that we must deal with is that another thread may attempt to complete the synchronization before setup is finished. We solve this problem by piggybacking on the mechanism used in the single-threaded implementation to do "garbage collection" of completed events. For each synchronization event, we allocate an event-state reference to hold the state of the synchronization.

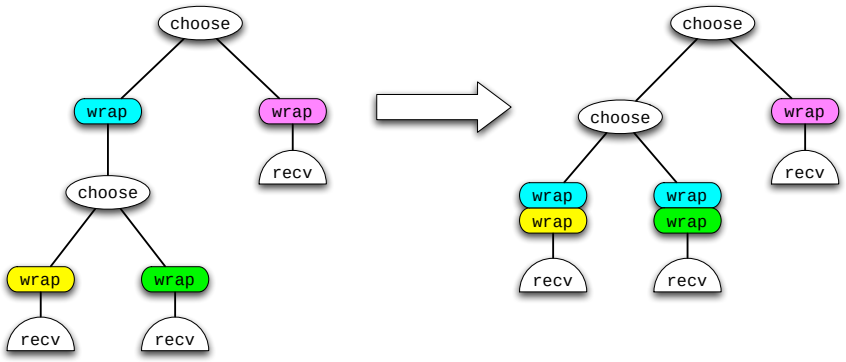


Fig. 4. The canonical-event transformation

```
datatype event_status = INIT | WAITING | SYNCHED
type event_state = event_status ref
```

The `INIT` state denotes that event setup is in progress, `WAITING` denotes that setup is complete and the event is available for synchronization, and `SYNCHED` denotes that the event has been synchronized on.

The canonical-event representation is implemented by the following datatype:

```
datatype 'a evt
  = BEVT of {
        pollFn : unit -> bool,
        doFn : 'a cont -> unit,
        blockFn : (event_state * 'a cont) -> unit
      }
  | CHOOSE of 'a evt * 'a evt
```

In this type, wrapped base events are represented by three functions: the `pollFn` is used to poll an event to test for its availability, the `doFn` is used to synchronize on an enabled event, and the `blockFn` is used to enqueue a suspended thread on the event. Both `doFn` and `blockFn` take resumption continuations as arguments. These continuations are used to return from the invoking `sync` operation. Note also that the `blockFn` takes a state flag as an argument. This flag is enqueued along with the resume continuation in the waiting queue maintained by the underlying communication object.

4.3 The PCML `sync` operation

The implementation of the `sync` operation is given in Figure 5. It is structured as three functions that correspond to the items in the protocol description above. Each of these functions does a walk over the tree representation of the event value to apply its operation to the base events at the leaves. The `poll` function polls each base event and returns a list of `doFn` functions for the base events that were enabled. The `doEvt` function, which is applied to this list, attempts to complete the synchronization

```
fun sync ev = callcc (fn resumeK => let
    (* optimistically poll the base events *)
      fun poll (BEVT{pollFn, doFn, blockFn}, enabled) =
            if pollFn()
              then doFn::enabled
              else enabled
        | poll (CHOOSE(ev1, ev2), enabled) =
            poll(ev2, poll(ev1, enabled))
    (* attempt to complete an enabled communication *)
      fun doEvt [] = blockThd()
        | doEvt (doFn::r) = (
            doFn resumeK;
          (* if we get here, that means that the
           * attempt failed, so try the next one
           *)
            doEvt r)
    (* record the calling thread's continuation in the
     * event waiting queues
     *)
      and blockThd () = let
            val flg = ref INIT
            fun block (BEVT{blockFn, ...}) =
                  blockFn(flg, resumeK)
              | block (CHOOSE(ev1, ev2)) = (
                  block ev1; block ev2)
            in
              block ev;
            (* if we get here, then setup is complete *)
              flg := WAITING;
              dispatch()
            end
      in
        doEvt (poll (ev, []))
      end)
```

Fig. 5. The primitive `sync` operation

on one of the base event's using its `doFn` function. Since the state of the base event might have changed since it was polled, it possible for the `doFn` to fail, in which case it returns. Otherwise, it will transfer control to the resume continuation. If `doEvt` is unable to complete the synchronization of any of the enabled events (or there were no enabled events), then it calls `blockThd`. This function allocates the state flag and then calls the `blockFn` of each of the base events to enqueue the resumption continuation. If the state of the base event has changed since polling (*i.e.*, it has become enabled), then the `blockFn` will complete the synchronization, otherwise it returns. If all of the `blockFn`s return, then the event's state is changed to `WAITING` and some other thread is dispatched.

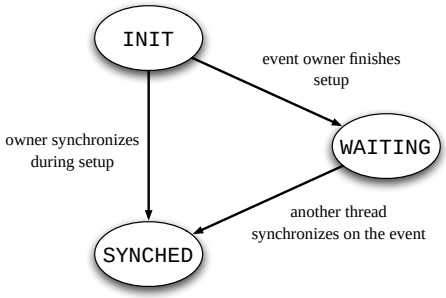


Fig. 6. The state-transitions of a synchronization event.

Because `sync` does not hold locks on the underlying communication objects, it is possible that some other thread may attempt to synchronize on one of the base events before `blockThd` has completed its work. Our policy is to only allow the owner thread of a synchronization event (*i.e.*, the caller of the `sync` operation) to change its state from `INIT`, as is shown in Figure 6. To implement this policy, non-owners use the following utility function to change the state:

```
fun claimEvent flg = (case cas(flg, WAITING, SYNCHED)
       of WAITING => true
        | INIT => claimEvent flg
        | SYNCHED => false
      (* end case *))
```

This function forces its caller to wait until setup is complete before being allowed to synchronize on the event. If the state is already `SYNCHED`, then it returns false.

An obvious simplification of this design would be to combine `pollFn` and `doFn` into a single function. There is a disadvantage of merging these two functions, however, which is that by polling all of the base events first, it is possible to impose an ordering on enabled events, such as priorities or to support fairness [2].

4.4 The PCML event combinators

The implementation of the primitive-event combinators is largely straightfoward, with the exception of `wrap`, which involves both the continuation hacking needed to hook in the wrapper function and event canonicalization. The implementation of `wrap` is given in Figure 7. When applied to a base-event value, we need to arrange for the wrapper function (`f`) to be applied to the values thrown to the resumption continuation by `doFn` and `pollFn`. When applied to a `CHOOSE` value, it pushes the wrapper down into both branches as described by the equivalence in Section 4.2.

4.5 PCML channels

The other half of the synchronization protocol is implemented in the base-event values for the communication objects. The representation of a channel consists of a spinlock, a

```
fun wrap (BEVT{pollFn, doFn, blockFn}, f) = BEVT{
        pollFn = pollFn,
        doFn = fn k => callcc (fn retK =>
          throw k (f (callcc (fn k' => (doFn k';
                                        throw retK ()))))),
        blockFn = fn (flg, k) => callcc (fn retK =>
          throw k (f (callcc (fn k' => (blockFn(flg, k');
                                        throw retK ())))))
      }
  | wrap (CHOOSE(ev1, ev2), f) =
      CHOOSE(wrap(ev1, f), wrap(ev2, f))
```

Fig. 7. The primitive `wrap` combinator

queue of blocked senders (with messages), and a queue of blocked receivers (with their owner's event state).

```
datatype 'a chan = Ch of {
    lock : bool ref,
    sendq : ('a * unit cont) queue,
    recvq : (event_state * 'a cont) queue
  }
```

The code for `recvEvt` is a non-trivial example of a base-event implementation and is given in Figure 8. The `pollFn` checks to see if the channel's `sendq` is empty. Since this operation only involves reading the state of the queue, it can be done without locking. Even if the results are erroneous because of conflicts with other threads, the fallback code in the `doFn` and `blockFn` will ensure correct behavior. The `doFn` is called when the `sendq` is expected to be nonempty. It locks the channel, removes an item from the `sendq` and then releases the lock. If the queue was empty (*i.e.*, `NONE` was returned), then the `doFn` returns. Otherwise, it enqueues the blocked sender and throws the message to the resume continuation of the `sync` operation. The `blockFn` is called when the `sendq` is expected to be empty. It also locks the channel and then checks the `sendq` in case its state has changed since polling. If there is an item available, then it is used to complete the synchronization. Otherwise, the resume continuation and event-state flag are enqueued in the channel's `recvq`.

The `send` operation on channels is given in Figure 9. The body of this function is a loop that examines the `recvq` for waiting events. If it finds one, then it completes the synchronization, otherwise it enqueues its resume continuation and message on the `sendq`.

5 Implementing full CML

In this section, we sketch how to build an implementation of the full set of CML event combinators from the `PRIM_CML` interface that we implemented in the previous section. The basic idea, which was suggested by Matthew Fluet [16], is to move the bookkeeping used to track negative acknowledgments out of the implementation of sync and

```
fun recvEvt (Ch{lock, sendq, recvq}) = let
      fun pollFn () = not(isEmptyQ(sendq))
      fun doFn k = let
            val _ = spinLock lock
            val item = dequeue sendq
            in
              spinUnlock lock;
              case item
               of NONE => ()
                | SOME(msg, sendK) => (
                    enqueueRdy sendK;
                    throw k msg)
              (* end case *)
            end
      fun blockFn (flg : event_state, k) = (
            spinLock lock;
          (* if we are lucky, a sender may have arrived
           * on the channel since we polled it.
           *)
            case dequeue sendq
             of SOME(msg, sendK) => (
                (* there is a matching send *)
                  spinUnlock lock;
                  flg := SYNCHED;
                  enqueueRdy sendK;
                  throw k msg)
              | NONE => (
                  enqueue (recvq, (flg, k));
                  spinUnlock lock)
            (* end case *))
      in
        BEVT{pollFn = pollFn, doFn = doFn, blockFn = blockFn}
      end
```

Fig. 8. The `recvEvt` event constructor

into guards and wrappers. Space does not permit a complete description of this layer, but we cover the highlights.

In this implementation, negative acknowledgments are signaled using the condition variables (cvars) provided by PCML. Since we must create these variables at synchronization time, we represent events as suspended computations (or *thunks*). The event type has the following definition:

```
datatype 'a event
  = E of (cvar list * (cvar list * 'a thunk) PCML.evt) thunk
```

where the `thunk` type is

```
type 'a thunk = unit -> 'a
```

```
fun send (Ch{lock, sendq, recvq}, msg) = callcc (fn sendK => let
      val _ = spinLock lock
      fun tryLp () = (case dequeue recvq
             of SOME(flg, recvK) =>
                (* there is a matching recv, but we must
                 * check to make sure that some other
                 * thread has not already claimed the event.
                 *)
                  if claimEvent flg
                    then ( (* we got it *)
                      spinUnlock lock;
                      enqueueRdy sendK;
                      throw recvK msg)
                    else (* someone else got the event *)
                      tryLp ()
              | NONE => (
                  enqueue (sendq, (msg, sendK));
                  spinUnlock lock;
                  dispatch ())
            (* end case *))
      in
        tryLp ()
      end)
```

Fig. 9. The `send` operation

The outermost thunk is a suspension used to delay the evaluation of guards until synchronization time. When evaluated, it produces a list of cvars and a primitive event. The cvars are used to signal the negative acknowledgments for the event. The primitive event, when synchronized, will yield a list of those cvars that need to be signaled and a thunk that is the suspended wrapper action for the event. With this representation, the `sync` operation is straightforward.

```
fun sync (E thunk) = let
      val (_, ev) = thunk()
      val (cvs, act) = PCML.sync ev
      in
        List.app PCML.set cvs;
        act()
      end
```

We start by evaluating the top-level thunk to get the primitive event value, which we then synchronize on. The result of synchronization will be a list of cvars that need to be signaled and the wrapper thunk. We signal the nacks by setting the cvars and then evaluate the wrapper thunk.

The two combinators that are at the heart of the bookkeeping for negative acknowledgments are `withNack` and `choose`. The former creates a new cvar when its thunk is evaluated. This cvar is passed as an argument to `withNack`'s argument and is added to the list of cvars for its result.

```
fun withNack f = let
      fun thunk () = let
            val nack = PCML.new()
            val E thunk' = f (baseEvt (PCML.waitEvt nack))
            val (cvs, ev) = thunk' ()
            in
              (nack::cvs, ev)
            end
      in
        E thunk
      end
```

The purpose of negative acknowledgments is to signal that some other event in a choice was chosen, which means that the `choose` combinator must associate the cvars of its left side with the synchronization result of its right side (and vice versa).

```
fun choose (E thunk1, E thunk2) = let
      fun thunk () = let
            val (cvs1, ev1) = thunk1()
            val (cvs2, ev2) = thunk2()
            in (
              cvs1 @ cvs2,
              PCML.choose (
                PCML.wrap(ev1, fn (cvs, th) => (cvs @ cvs2, th)),
                PCML.wrap(ev2, fn (cvs, th) => (cvs @ cvs1, th)))
            ) end
      in
        E thunk
      end
```

Space does not permit a description of the other mechanisms, but they can be found in a forthcoming technical report [14].

6 Related work

Various authors have described implementations of choice protocols using message passing as the underlying mechanism [17–20]. While these protocols could, in principle, be mapped to a shared-memory implementation, we believe that our approach is both simpler and more efficient.

Russell described a monadic implementation of CML-style events on top of Concurrent Haskell [8]. His implementation uses Concurrent Haskell's M-vars for concurrency control and he uses an ordered two-phase locking scheme to commit to communications. A key difference in his implementation is that choice is biased to the left, which means that he can commit immediately to an enabled event during the polling phase. This feature greatly simplifies the implementation, since it does not have to handle changes in event status between the polling phase and the commit phase. Russell's implementation did not support multiprocessors (because Concurrent Haskell did not support them at the time), but presumably would work on a parallel implementation of

Concurrent Haskell. Donnelly and Fluet have implemented a version of events that support transactions on top of Haskell's STM mechanism [16]. Their mechanism is quite powerful and, thus, their implementation is quite complicated.

In earlier work, we reported on specialized implementations of CML's channel operations that can be used when program analysis determines that it is safe [21]. Those specialized implementations fit into our framework and can be regarded as complementary.

7 Conclusion

We have described a new protocol for implementing Asymmetric CML on multiprocessors. This implementation consists of a primitive layer that provides basic synchronous operations, non-deterministic choice, and post-synchronization wrappers. This layer is implemented using a new optimistic-concurrency protocol. The full set of CML event combinators is then constructed on top of this primitive layer. One advantage of this architecture is that the more complicated upper layer does not directly use locks or thread scheduling operations.

We have implemented the primitive layer in the Manticore system using the Manticore compiler's BOM intermediate representation [13]. This implementation must also deal with preemption, which we do by locally masking preemption. Unfortunately, Manticore is not yet stable enough to be able to run meaningful performance tests, although we have been able to test the correctness of the implementation on an 8-way parallel system. We expect that the basic performance of the primitives will be good when channels are used to implement point-to-point communications (as is common), but the interesting question will be how they perform in a situation with many senders or receivers sharing a single channel. We plan to provide preliminary performance results in a forthcoming technical report [14].

In the longer term, we want to extend the PCML layer to support output guards (*i.e.*, `sendEvt`). In our protocol, adding this event constructor complicates the implementation in a couple of significant ways. First, it becomes possible to write code that has matching communications in a single choice, as in the following example:

```
sync (choose (
  recvEvt ch,
  wrap (sendEvt(ch, 1), fn () => 2)))
```

The implementation must detect such cases and avoid having a thread communicate with itself. The second problem is that committing to a synchronization will require atomically updating the states of two different synchronization events. Two-phase locking is one possible solution, but it requires introducing a linear order on synchronization events to avoid deadlock. Instead, we are exploring the use of implementation techniques from STM [22], but we have not worked out the details.

References

1. Reppy, J.H.: CML: A higher-order concurrent language. In: PLDI '91, New York, NY, ACM (June 1991) 293–305

2. Reppy, J.H.: Concurrent Programming in ML. Cambridge University Press, Cambridge, England (1999)
3. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). The MIT Press, Cambridge, MA (1997)
4. Gansner, E.R., Reppy, J.H. In: A Multi-threaded Higher-order User Interface Toolkit. Volume 1 of Software Trends. John Wiley & Sons (1993) 61–80
5. Young, C., YN, L., Szymanski, T., Reppy, J., Pike, R., Narlikar, G., Mullender, S., Grosse, E.: Protium, an infrastructure for partitioned applications. In: HotOS-X. (January 2001) 41–46
6. MLton: Concurrent ML Available at http://mlton.org/ConcurrentML.
7. Leroy, X.: The Objective Caml System (release 3.00). (April 2000) Available from `http://caml.inria.fr`.
8. Russell, G.: Events in Haskell, and how to implement them. In: ICFP '01. (September 2001) 157–168
9. Flatt, M., Findler, R.B.: Kill-safe synchronization abstractions. In: PLDI '04. (June 2004) 47–58
10. Fisher, K., Reppy, J.: The design of a class mechanism for Moby. In: PLDI '99. (May 1999) 37–49
11. Demaine, E.D.: Higher-order concurrency in Java. In: WoTUG20. (April 1997) 34–47 Available from http://theory.csail.mit.edu/ edemaine/papers/WoTUG20/.
12. Fluet, M., Rainey, M., Reppy, J., Shaw, A., Xiao, Y.: Manticore: A heterogeneous parallel language. In: DAMP '07, New York, NY, ACM (January 2007) 37–44
13. Fluet, M., Ford, N., Rainey, M., Reppy, J., Shaw, A., Xiao, Y.: Status report: The Manticore project. In: ML '07, New York, NY, ACM (October 2007) 15–24
14. Reppy, J., Xiao, Y.: Toward parallel CML (extended version). Technical report, Department of Computer Science, University of Chicago *Forthcoming*.
15. Reppy, J.H.: First-class synchronous operations in Standard ML. Technical Report TR 89-1068, Dept. of CS, Cornell University (December 1989)
16. Donnelly, K., Fluet, M.: Transactional events. In: ICFP '06, New York, NY, ACM (2006) 124–135
17. Buckley, G.N., Silberschatz, A.: An effective implementation for the generalized input-output construct of CSP. ACM TOPLAS **5**(2) (April 1983) 223–235
18. Bornat, R.: A protocol for generalized occam. SP&E **16**(9) (September 1986) 783–799
19. Knabe, F.: A distributed protocol for channel-based communication with choice. Technical Report ECRC-92-16, European Computer-industry Research Center (October 1992)
20. Demaine, E.D.: Protocols for non-deterministic communication over synchronous channels. In: Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP'98). (March 1998) 24–30 Available from http://theory.csail.mit.edu/ edemaine/papers/IPPS98/.
21. Reppy, J., Xiao, Y.: Specialization of CML message-passing primitives. In: POPL '07, New York, NY, ACM (January 2007) 315–326
22. Shavit, N., Touitou, D.: Software transactional memory. In: PODC '95, New York, NY, ACM (1995) 204–213