Models for Implicitly Parallel Execution

Petr Krajča

Department of Computer Science,
Palacky University, Olomouc, Czech Republic

Programming Languages in Multi-core Era

Paradigm shifts

- hardware (8+ cores)
- software (hoped for)

Explicit Parallelism

- well-established methods and tools
- support from programming languages, operating systems
- still not get what we want

Implicit Parallelism

- partial success (loop parallelization, instruction level parallelism)
- functional programming: great expectations

Schemik: Introduction

- implicitly parallel dialect of Scheme
- testbed for our research
- parallel execution of programs is done independently of the programmer
- returns always the same results
- roots in functional programming
- handles side-effects correctly using Software Transactional Memory
- supports various features (higher-order functions, macros, continuations as first-class elements, etc.)
- transfer of experience to similar programming languages (e.g., JavaScript)

Schemik

- implicitly parallel dialect and interpreter of Scheme (R5RS)
- lexically scoped, tail-calls, macros (lispish), continuations, compatible standard library
- S-expressions, prefix notation
- e.g., $1 + 2 \times 3 \Longrightarrow$ `(+ 1 (* 2 3))`
- stack-based model of evaluation

Evaluation Model (Outline)

- evaluation is described by pushdown automaton having two stacks:
  - *execution stack* – contains operation to be done
  - *result stack* – stores objects playing the role of intermediate results and operands
- operation is a tuple $\langle$operation-name, arg, $\mathcal{E}$, flag$\rangle$
- transitions of an automaton are made according to the operation on the top of the execution stack
- for instance, we consider the following stack operations:
  - EVAL – initiates evaluation of given (sub)expression
  - INSPECT – controls the order of evaluation of arguments
  - FUNCALL – performs function application
  - SET – redefines binding of lexical variable
  - FEVAL – initiates evaluation in a parallel branch

Evaluation Model: An Example (1 of 3)

1. E: $\langle$EVAL 42$\rangle$ $\rrbracket$
   R: $\rrbracket$

Evaluation Model: An Example (1 of 3)

1. E: $\langle$EVAL 42$\rangle$ $\rrbracket$
   R: $\rrbracket$
2. E: $\rrbracket$
   R: 42 $\rrbracket$

Evaluation Model: An Example (1 of 3)

1. E: $\langle$EVAL 42$\rangle$ $\rrbracket$
   R: $\rrbracket$
2. E: $\rrbracket$
   R: 42 $\rrbracket$

1. E: $\langle$EVAL $*\rangle$ $\rrbracket$
   R: $\rrbracket$

Evaluation Model: An Example (1 of 3)

1. E: $\langle$EVAL 42$\rangle$ $\rrbracket$
   R: $\rrbracket$
2. E: $\rrbracket$
   R: 42 $\rrbracket$

1. E: $\langle$EVAL $*\rangle$ $\rrbracket$
   R: $\rrbracket$
2. E: $\rrbracket$
   R: *primitive func. ** $\rrbracket$

Evaluation Model: An Example (2 of 3)

1. E: ⟨EVAL `(* 2 3)`⟩ ⟧
   R: ⟧

Evaluation Model: An Example (2 of 3)

1. E: $\langle$EVAL `(* 2 3)`$\rangle$ $\rrbracket$
   R: $\rrbracket$
2. E: $\langle$EVAL `*`$\rangle$ $\langle$INSPECT `(2 3)`$\rangle$ $\rrbracket$
   R: $\rrbracket$

Evaluation Model: An Example (2 of 3)

1. E: ⟨EVAL `(* 2 3)`⟩ ⟧
   R: ⟧
2. E: ⟨EVAL ∗⟩ ⟨INSPECT `(2 3)`⟩ ⟧
   R: ⟧
3. E: ⟨INSPECT `(2 3)`⟩ ⟧
   R: *primitive func. ** ⟧

Evaluation Model: An Example (2 of 3)

1. E: ⟨EVAL (* 2 3)⟩ ⟧
   R: ⟧
2. E: ⟨EVAL *⟩ ⟨INSPECT (2 3)⟩ ⟧
   R: ⟧
3. E: ⟨INSPECT (2 3)⟩ ⟧
   R: *primitive func. ** ⟧
4. E: ⟨EVAL 2⟩ ⟨EVAL 3⟩ ⟨FUNCALL 2⟩ ⟧
   R: *primitive func. ** ⟧

Evaluation Model: An Example (2 of 3)

1. E: ⟨EVAL (* 2 3)⟩ ⟧
   R: ⟧
2. E: ⟨EVAL *⟩ ⟨INSPECT (2 3)⟩ ⟧
   R: ⟧
3. E: ⟨INSPECT (2 3)⟩ ⟧
   R: *primitive func. ** ⟧
4. E: ⟨EVAL 2⟩ ⟨EVAL 3⟩ ⟨FUNCALL 2⟩ ⟧
   R: *primitive func. ** ⟧
5. E: ⟨EVAL 3⟩ ⟨FUNCALL 2⟩ ⟧
   R: 2 *primitive func. ** ⟧
6. E: ⟨FUNCALL 2⟩ ⟧
   R: 3 2 *primitive func. ** ⟧

Evaluation Model: An Example (2 of 3)

1. E: ⟨EVAL (* 2 3)⟩ ⟧
   R: ⟧
2. E: ⟨EVAL *⟩ ⟨INSPECT (2 3)⟩ ⟧
   R: ⟧
3. E: ⟨INSPECT (2 3)⟩ ⟧
   R: *primitive func.* * ⟧
4. E: ⟨EVAL 2⟩ ⟨EVAL 3⟩ ⟨FUNCALL 2⟩ ⟧
   R: *primitive func.* * ⟧
5. E: ⟨EVAL 3⟩ ⟨FUNCALL 2⟩ ⟧
   R: 2 *primitive func.* * ⟧
6. E: ⟨FUNCALL 2⟩ ⟧
   R: 3 2 *primitive func.* * ⟧
7. E: ⟧
   R: 6 ⟧

Parallelization

- each operation EVAL may be performed in an independent evaluator
- each evaluator has an external entity *scheduler* acting as *deus ex machina* and converting EVAL operations into the new evaluators and the FEVAL operations

Parallelization

- each operation EVAL may be performed in an independent evaluator
- each evaluator has an external entity *scheduler* acting as *deus ex machina* and converting EVAL operations into the new evaluators and the FEVAL operations

$$EV_1: \begin{array}{l} \text{E: } \cdots \ \langle \text{EVAL } \mathit{object} \rangle \ \cdots \ \rrbracket \\ \text{R: } \cdots \ \rrbracket \end{array}$$

Parallelization

- each operation EVAL may be performed in an independent evaluator
- each evaluator has an external entity *scheduler* acting as *deus ex machina* and converting EVAL operations into the new evaluators and the FEVAL operations

$$
\begin{array}{ll}
\mathit{EV}_1: & \text{E: } \cdots \ \langle \text{EVAL } \mathit{object} \rangle \ \cdots \ \rrbracket \\
 & \text{R: } \cdots \ \rrbracket \\
 & \qquad\qquad \Downarrow \\
\mathit{EV}_1: & \text{E: } \cdots \ \langle \text{FEVAL } \langle \mathit{EV}_2, \mathit{object} \rangle \rangle \ \cdots \ \rrbracket \\
 & \text{R: } \cdots \ \rrbracket \\
\mathit{EV}_2: & \text{E: } \langle \text{EVAL } \mathit{object} \rangle \ \rrbracket \\
 & \text{R: } \rrbracket
\end{array}
$$

Parallelization

- each operation EVAL may be performed in an independent evaluator
- each evaluator has an external entity *scheduler* acting as *deus ex machina* and converting EVAL operations into the new evaluators and the FEVAL operations

$$
\begin{array}{ll}
EV_1: & \text{E: } \cdots \ \langle \text{EVAL } \textit{object} \rangle \ \cdots \ \rrbracket \\
 & \text{R: } \cdots \ \rrbracket \\
 & \qquad\qquad \Downarrow \\
EV_1: & \text{E: } \cdots \ \langle \text{FEVAL } \langle EV_2, \textit{object} \rangle \rangle \ \cdots \ \rrbracket \\
 & \text{R: } \cdots \ \rrbracket \\
EV_2: & \text{E: } \langle \text{EVAL } \textit{object} \rangle \ \rrbracket \\
 & \text{R: } \rrbracket
\end{array}
$$

- an invocation of FEVAL represents merging of two branches of the execution (stacks from the referenced evaluator are appended to the corresponding stacks of the evaluator processing the FEVAL operation)
- evaluators form a tree (hierarchy)

Hierarchy of Evaluators

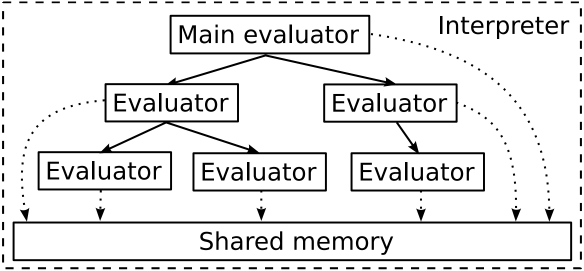


Figure : Structure of the interpreter

Issues

- inherently sequential algorithms
- destructive object mutations (software transactional memory)
- expressions worth parallelizing (heuristics)
- **performance**

Just-in-Time Compilation

- many transitions of the automaton (even for simple expressions)
  $\rightarrow$ opportunity for compilation

- automatic parallelization relies on knowledge of the program execution structure (execution stack)
  $\rightarrow$ compilation ruins this feature

Just-in-Time Compilation

- many transitions of the automaton (even for simple expressions)
  → opportunity for compilation

- automatic parallelization relies on knowledge of the program execution structure (execution stack)
  → compilation ruins this feature

Solution

- compile only expressions insignificant for parallelization

Compilable Expression

Definition

Expression E is *compilable* if

(1) E is either an atom (number, symbol, etc.),

(2) or E is and expression of a form $(E_1 \ E_2 \ \ldots E_n)$ where E_1 is primitive function or special operator and $E_2, \ldots, E_n$ are *compilable* expressions.

- Examples: `(+ 1 a)`, `(car (cdr a))`
- Recursive nature of the definition is used to incrementally compile expressions.
- How to resolve that E_1 is a primitive function?

Compilation (1 of 3): Picking Candidates

1. reader (parser) marks all lists consisting solely of atoms as candidates for compilation
2. operation EVAL checks if its argument
    - has associated machine code that can be executed,
    - or, is candidate for compilation and can be enqueued into a queue of expressions waiting for compilation;
    - if no machine code is available, operation EVAL proceeds as usually
3. compiler tries to compile each expression in its queue and if it succeeds
    - it attaches machine code to the expression (+ its high-level intermediate representation)
    - marks parent expression as a candidate for compilation

Compilation (2 of 3): Intermediate Representations

High-level Intermediate Representation (HIR)

- similar to three-address code
- instructions, registers, constants, blocks
- template (registers may be shifted by offset)
- instruction examples:
  - `set` R_i, *value*
  - `eval-symbol` R_i, *symbol*
  - `car` R_i, R_j
  - `add` R_i, R_j, *value*
  - `putarg` i, *source*
  - `funcall` R_i, *function*
  - ...
- allows traditional optimizations (copy propagation, constant folding, etc.)

Low-level Intermediate Representation (LIR)

- optional, RISC-like instruction set

Compilation (3 of 3): Sketch of the Algorithm

- expression is not compiled directly
- function generating HIR is created instead
- serves as a template
- allows for incremental compilation

Sketch of the algorithm . . .

CompileHIR(E, $base$):

return procedure $\mathrm{HIR}(i)$ such that:

if E is a constant (e.g., number) **then**
  emit operation `set` R_{base+i}, E

if E is a symbol **then**
  emit operation `eval-symbol` R_{base+i}, E

if E has attached HIR code **then**
  invoke $\mathrm{HIR}(base + i)$

if E is an expression `(`fun E_2 `...` E_n`)` where fun is a primitive function **then**
  for all E_j where $j \in \{2, \ldots, n\}$ **do**
    invoke $\textsc{CompileHIR}(E_j,\ base + i + j - 1)$
  if fun is primitive function **+** **then**
    invoke $\textsc{CompileAddition}(base + i,\ n)$
  else
    invoke $\textsc{CompileFuncall}(base + i,\ n,\ fun)$

if E is expression `(if` E_{cond} E_1 E_2`)` **and** E_{cond} is compilable **then**
  invoke $\textsc{CompileIf}(base + i,\ E_{cond},\ E_1,\ E_2)$

if E is quotation `(quote` val`)` **then**
  emit operation `set` R_{base+i}, val

otherwise abort compilation

Example

CompileHIR(`(foo (+ a 1))`, 10):

Procedure $\mathrm{HIR}(i)$:

emit operations:

`eval-symbol` R_{12+i}, `a`
`set` R_{13+i}, 1
`add` R_{11+i}, R_{12+i}, R_{13+i}

`prepare` 1
`putarg` 1, R_{11+i}
`funcall` R_{10+i}, *foo*

Conditionals

Operator `if`

- `(if (< a 0) (- a) a)`
- `(if (< a 0) (- a) (foo a))`
- allowed to directly manipulate with stacks

Compiling conditions

if E_{cond} has attached HIR code without the `exct-push` operation **then**
  invoke $\mathrm{HIR}(i)$
else
  abort compilation
end if
for all E_j where $j \in \{1, 2\}$ **do**
  // *create code block* $\textsc{Branch}_j$ *such that:*
  if E_j has attached HIR code **then**
    $\textsc{Branch}_j \leftarrow \mathrm{HIR}(i)$
  else
    $\textsc{Branch}_j \leftarrow$ operation `exct-push` E_j
  end if
end for
if E_1 has attached HIR code **and** $\textsc{Branch}_2$ contains `exct-push` **then**
  append to $\textsc{Branch}_1$ operation `rslt-push` R_1.
end if
if E_2 has attached HIR code **and** $\textsc{Branch}_1$ contains `exct-push` **then**
  append to $\textsc{Branch}_2$ operation `rslt-push` R_2
end if
emit operation `if` R_i, $\textsc{Branch}_1$, $\textsc{Branch}_2$

Implementation

- compiler is implemented in Schemik itself
- significant reduction in code size
- can run in parallel
- tends to compile itself first
- MyJIT library emits machine code
  - emits machine code for i386, AMD64, SPARC processors
  - intermediate language $\Rightarrow$ RISC-like ISA
  - written in ANSI C
  - thread-safe
  - easy to use and easy to extend design (future optimizations)
  - GNU LGPL v.3
  - `http://myjit.sourceforge.net`
- HIR and machine code attached to expressions (lists) in a form similar to p-list (meta-data)

Additional Optimizations

Inlining

- function consisting merely of an expression which is compilable
- `(define (cadr a) (car (cdr)))`
- directly inlined

Specialization

- dynamically typed programming language
- tagged unions

```
typedef struct scm_value {
  scm_type type;
  union {
    int integer;
    char *symbol;
  } value;
} scm_value;
```

- and tagged pointers representing objects

```
#define scm_new_int(__val) ((scm_value *)(1 | ((__val) << 1)))
#define SCM_INT(x)      (int)((long)x >> 1)
```

Specilization (cntd.)

- lots of boxing and unboxing (testing, allocations, shifting)
- often unnecessary, e.g.,

```
(define (fib n)
   (if (< n 3) n
       (+ (fib (- n 1)) (fib (- n 2)))))
(fib 10)
(fib 10.0)
```

- only two distinct code paths
- for each compiled expression multiple versions are generated
- *generic code* (fallback)
- *specialized* code for specific types of values
  - type checking performed at the begining of the code block
  - if the specialized version is not available, the code is enqueued for processing by the compiler, generic version is used
  - more optimizations – condition elimination (expensive operations), dead code elimination
  - boxing and unboxing only on entry and on exit from the compiled code

Which expressions should be picked by scheduler?

- assumption: compiled expressions are not suitable for parallelization
- scheduler picks expressions which are not compilable
- expressions near to the bottom tend to be more complex

Providing hints to the runtime environment

- new calling convention *call-by-future*
- `(lambda (a b (future c)) ... )`
- called function creates a *future* (may be an independent thread)
- there is no need for `force` operation
- implicitly creates a transaction
- called function controls execution (speculative execution)
- allows to abort computation

Software Transactional Memory: Main Ideas (1 of 2)

- inspiration from RDBMS
- allows to split execution of the program into logical blocks (transaction; ACI)
- in our case STM is not a language construct
- mean which allows to consistently access main memory and detect collisions
- each thread has its own image of the memory (transaction); hierarchy of nested transaction
- transaction only encapsulates access to the memory
- transactions are committed in the logical order (left-to-right)
- no contention manager; each transaction always commits

Software Transactional Memory: Main Ideas (2 of 2)

- any object may be updated (no information in advance)
- mutations are (should be) rare $\Rightarrow$ functional programming
- mutations should have minimal side-effects (loops, local assignments, etc.)

Software Transactional Memory: Main Ideas (2 of 2)

- any object may be updated (no information in advance)
- mutations are (should be) rare $\Rightarrow$ functional programming
- mutations should have minimal side-effects (loops, local assignments, etc.)
- **"Think globally! Act within local variable scope!"**

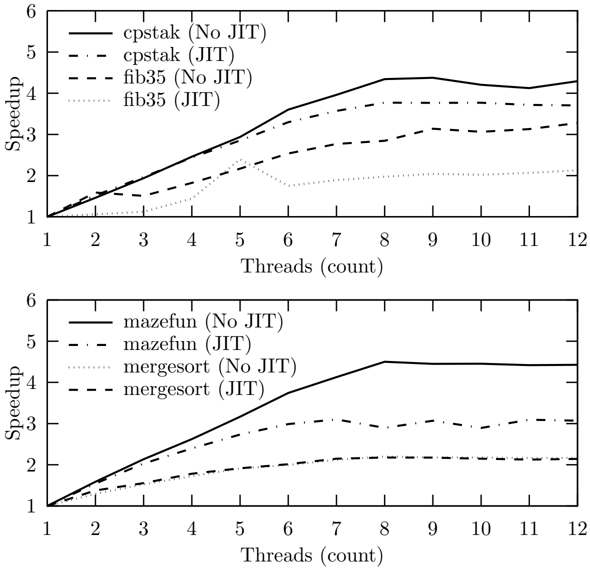
Call-by-future

- allow to impose other convenient macros and functions
- `(parallel-let ((a foo) (b bar)) code)` →
```
 ((lambda ((future a) (future b))
     code)
  foo bar)
```
- `(future a)` → `(lambda ((future x)) x)`
- `(parallel-if cond then else)` →
```
((lambda ((future t) (future e))
    (if cond
        (begin (abort e) t)
        (begin (abort t) e))
  then else)
```
- additional functions for controlling transactions
- `abort` – aborts transaction (future)
- `retry` – retries transaction (future)
- `stalled?` – waiting for an operation with side-effect
- `interrupted?` – interrupted due to the collision

Evaluation

	1 thread		8 threads		
	No JIT	JIT	No JIT	JIT	Guile
bubblesort	5.77	3.27	5.81	3.29	**1.35**
combinations	2.74	1.62	1.33	**0.94**	1.84
cpstak	8.77	5.19	2.08	**1.40**	2.79
fib30	1.23	0.49	0.43	**0.30**	0.31
fib33	5.24	2.03	1.69	**0.87**	1.27
fib35	13.62	5.19	4.49	**2.42**	3.32
mazefun	7.62	3.55	1.69	**1.15**	2.24
mergesort	6.53	3.70	2.99	1.69	**0.14**
nqueens	3.68	1.80	1.30	1.08	**0.64**
powerset	2.41	1.53	1.78	**0.95**	1.97
primes	8.65	3.63	4.11	3.48	**1.86**
quicksort	6.33	2.24	3.79	**1.60**	3.70
sum	8.08	2.78	3.33	2.84	**2.31**
tak	5.72	1.49	1.31	**0.81**	1.73

Scalability



Thank You!