Distributed Partial Evaluation

Michael Sperber Herbert Klaeren Peter Thiemann
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
Sand 13, D-72076 Tübingen, Germany
{sperber,klaeren,thiemann}@informatik.uni-tuebingen.de

Abstract

Partial evaluation is an automatic program transformation that optimizes programs by specialization. We speed up the specialization process by utilizing the natural coarse-grained parallelism inherent in the partial evaluation process. We have supplemented an existing partial evaluation system for the Scheme programming language by a farm-of-workers model for parallel partial evaluation in a network of loosely coupled workstations. Our implementation speeds up specialization by a factor of 2–3 on 6 processors.

Keywords functional programming, automatic program transformation, partial evaluation

1 Introduction

Partial evaluation is a powerful program-specialization technique based on constant propagation. Given the *static* (known) parameters of a source program, partial evaluation constructs a *residual program*—an optimized, specialized version of the program, which on application to the remaining *dynamic* parameters produces the same result as the original program applied to all parameters.

Partial evaluation has a number of applications in computer graphics, scientific computation, operating systems, and metaprogramming. In particular, partial evaluation can automatically generate efficient parser generators from general parsers [23] as well as compilers from interpreters [15, 24]. For typical applications the speedup achieved by partial evaluation ranges between 2 and 10.

It is important to improve the speed of partial evaluators for the following reasons:

1. Partial evaluation enables the construction of general and highly parameterized software systems without sacrificing efficiency. Specialization turns the general system into a specialized, efficient one for specific parameter setting.
2. Automatic compiler generation is a particularly promising application of partial application. The idea is to specialize a language definition in the form of an interpreter with respect to a program to be compiled. The specialized program can then be regarded as a compiled program. Speeding up (and parallelizing) partial evaluation therefore implies speeding up (and parallelizing) compilation.
3. Specialization can be performed on demand (yielding run-time code generation [3]) so parallelized partial evaluation amounts to parallelized just-in-time compilation. In this area speed is of tantamount importance.

Building on an existing partial evaluation system for the Scheme language [13, 22] by one of the authors [26, 27], we present a model for parallelizing partial evaluation, along with an implementation of the model in a loosely coupled distributed environment. Even in this situation, the parallelization pays off since the parallelism is rather coarse-grained. The only shared resource is a central specialization cache.

In our preliminary experiments, typical speed-ups running on Ethernet networks of 6 Unix machines range between 2 and 3.

Overview The next section provides some technical background on partial evaluation. Section 3 outlines our approach in general terms and Section 4 describes some specifics of our implementation. In Section 5 we assess the performance of the implementation. Finally, we discuss related work (Section 6) and conclude (Section 7).

2 Partial Evaluation

This section provides some background on the standard partial evaluation techniques relevant to the present work. The focus is on *offline partial evaluation*, the prevailing methods to perform partial evaluation.

2.1 Offline Partial Evaluation

Offline partial evaluation [8, 15] consists of two phases, *binding-time analysis* (BTA) and *specialization*. The *binding time* of an expression describes at what time its value is available. The two basic binding times are *static* and *dynamic*. The partial evaluator can compute the values of all static expressions in a program whereas it has to produce residual code for the dynamic ones—their values are only available when the residual program runs.

The binding-time analysis automatically provides binding-time annotations for all expressions in a program, based on the binding times of its input parameters. This simplifies the second phase to mere interpretation of annotated programs. It applies the program to the static input, evaluating static expressions and building a residual program from the dynamic ones.

A simple example for partial evaluation is a procedure for computing the nth power of a number x, specialized to a static exponent n. The binding-time analysis produces an annotated program:

```
(define (power n x)
  (if (= n 0)
      1
      (* x (power (- n 1) x))))
```

In this listing, underlining is the annotation for "dynamic"—those expressions of the program that will end up in the residual program. Specializing the above program with respect to a value yields a residual program like the following (for $\mathtt{n} = 3$):

```
(define (power_n=3 x)
  (* x (* x (* x 1))))
```

As demonstrated in the example, the specializer unfolds procedure calls by default. This approach works in the example, since a static `if` controls the recursion of `power`. However, unfolding is not always desirable as it may lead to infinite specialization. It is often necessary to produce recursive residual programs for recursion under dynamic control. To this end, the binding-time analysis (or the user) marks certain procedures as *specialization functions*. Specialization functions are never unfolded, but the specializer generates specialized residual procedures from them. The specializer keeps track of (memoizes) the residual procedures (*specializations*) already created. Consequently, the specializer generates only one residual procedure for each set of equivalent calls to a specialization function. Specialization functions typically enclose dynamic conditionals or dynamic abstractions, as these constructs may give rise to dynamic recursion.

As an example, consider `power`, again, but this time with static x and unknown n. The corresponding annotated program indicates that the specializer cannot evaluate the `if`:

```
(define (power n x)
  (if (= n 0)
      1
      (* x (power (- n 1) x))))
```

Specializing this program for `x = 15` yields a recursive program that exhibits a structure almost identical to the original program.

```
(define (power_x=15 n)
  (if (= n 0)
      1
      (* 15 (power_x=15 (- n 1)))))
```

The specializer maintains a *specialization cache* in order to identify equivalent calls to a specialization function. The cache is necessary to ensure that specialization finished. The cache is a mapping from specialization functions and the static parts of their argument lists to residual function names.

The specialization cache works as follows: Suppose the specializer enters a specialization function f with arguments $\overline{a}$. First, the specializer extracts the *static skeleton* $\overline{s}$ and *the dynamic parts* $\overline{d}$ (determined by the BTA) from $\overline{a}$. Then, the specializer checks whether the cache already contains $f_{\overline{s}}$. If that is not the case, the specializer enters $f_{\overline{s}}$ into the cache and specializes f's body. In any case, the specializer generates a residual function call $f_{\overline{s}}(\overline{d})$. The combination of a specialization function and an appropriate static skeleton is called a *static configuration*, since that is what the specializer needs to create a specialization.

This scheme is called *depth-first specialization* because it amounts to a depth-first traversal of the dynamic call graph of a specialization process. As soon as the specializer encounters a specialization function, it descends into a new specialization unless the specialization cache already contains a matching static configuration.

For effective parallelization, the earlier breadth-first specialization strategy [16] is more convenient. In this scheme, the specialization of a function's body is completed before any new specialization starts. The specialization cache consists of two parts, a *pending list* and a *done cache*. The pending list contains static configurations for which no specializations have been created yet while the done cache contains calls whose specialization has been completed. Checking the cache—as in the preceding description—means checking for an entry in both lists.

2.2 Typical Applications

Typical applications that benefit from partial evaluation (and gain from parallelization) are the automatic generation of efficient parsers from general parsers, and compilation. Thus, our work paves the road to automatic generation of parallel parser generators and compilers.

2.2.1 Parser Generation

Conceptually, a parser is a function that accepts as arguments a grammar and an input string and returns some by-product of the construction of a derivation of the input string. It is straightforward to write such general parsers both for the LL and LR method [9, 19, 20, 23]. Partially evaluating a general parser with respect to a static grammar yields highly efficient residual parsers with performance on par with hand-tuned parser generators such as yacc or Bison [23].

Additionally, the same partial evaluation technology automatically yields efficient parser generators [1, 15, 26].

2.2.2 Compilation

Interpreters for realistic programming languages are typically easier to write than efficient compilers. Furthermore, an interpreter can be viewed as a semantics definition for a programming language. Partial evaluation can generate a compiler from an interpreter: Since an interpreter is (simply put) a function from a program and its input to an output, a partial evaluator can specialize an interpreter with respect to a static program and thereby generate a residual compiled program with all interpretive overhead removed [15]. This semantics-directed method can lead to efficient and highly optimizing compilers for realistic languages [17, 24].

2.3 Concrete Context

The present work builds on Thiemann's PGG system [26, 27], a partial evaluation system for the full Scheme language. One of its distinguishing features is its ability to perform static computations that involve side effects. The use of side effects is completely orthogonal to parallelization.

3 Distributed Partial Evaluation

We present a basic model for distributed partial evaluation, extending it stepwise to get reasonable performance by minimizing communication overhead. Our first approach still uses one synchronous message per specialization to coordinate the specialization process, using essentially RPC [2]. We also present an asynchronous model. The synchronous method never performs work twice whereas the asynchronous method speculatively starts a specialization that may already be performed elsewhere in the network.

3.1 Basic Concepts

The basic idea behind making partial evaluation amenable to parallelization is simple: Every single specialization only depends on its static configuration. It may generate new requests for specialization in the form of static configurations, but it is independent from every other specialization. Therefore, it is natural to consider distributing the work of creating the specializations to distinct computational agents called *specialization servers*. Each specialization server can perform work on an arbitrary specialization, given its static configuration.

As specialization produces more and more static configurations, it is necessary to distribute the work among the specialization servers present in the scenario. This is only possible centrally, as the configurations generated by the different servers may overlap in unpredictable ways. Hence, a designated computational agent, the *memoization master*, keeps a central memoization cache and a pending list. It administers these data structures and keeps the specialization servers busy. This model bears some similarity to the farm-of-workers model [12, 28].

The sole task of the memoization master is to serve as a monitor for the central specialization cache. The specialization servers specialize away and store the specializations to be collected after completion.

The specialization servers understand three basic messages:

initialize This message directs the specialization server to commence work, that is, to initialize its residual program store and to start asking the memoization master for work.

specialize *config* Specialize starting at some static configuration *config* provided along with the **specialize** message. Store the resulting residual procedure locally.

yield-residual-program Return all specializations collected since the last **initialize** message.

The memoization master initially accepts two kinds of message from the specialization servers:

server-is-idle A server sends this message when it has completed work on a specialization.

register-static-config *config* A server sends this message when it has encountered a call to a specialization function. The corresponding static configuration *config* is part of the message. The master simply adds the static configuration to its pending list unless it has processed the static configuration before.

During startup, the master initializes the specialization servers by sending **initialize** messages, puts the main function and its static parameters as an initial static configuration entry in the pending list, processes **register-static-config** messages, and answers each incoming **server-is-idle** message by a **specialize** message back to the respective specialization server. This happens asynchronously; several threads

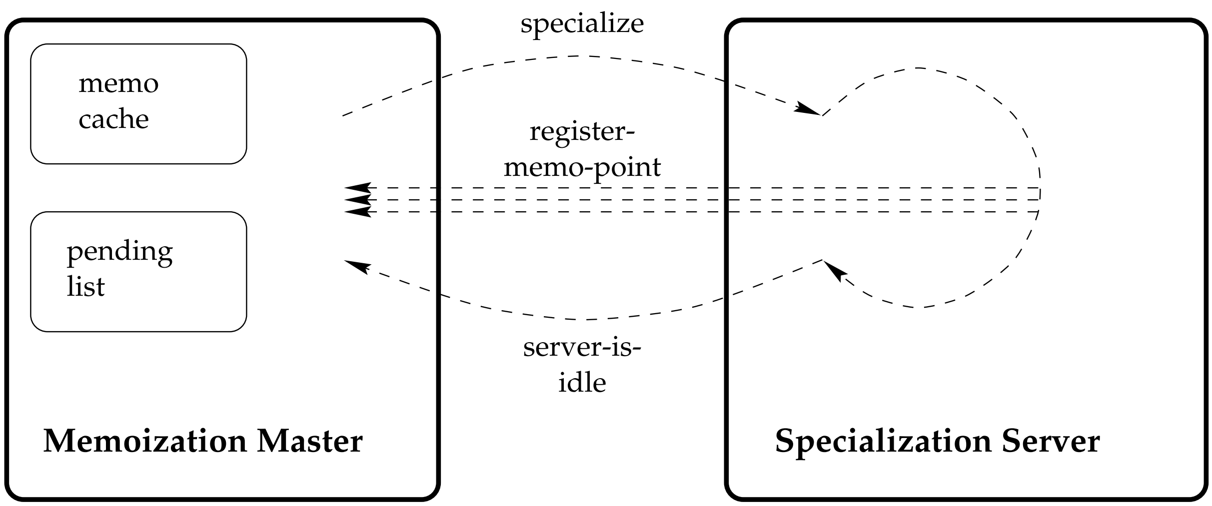

Figure 1: Basic model for distributed partial evaluation

may be active simultaneously on the memoization master accessing the cache. When all servers are idle and the pending list is empty, specialization has completed; the memoization master sends **yield-residual-program** messages to all specialization servers and combines the returned program fragments to yield a complete residual program.

A specialization server, after having received an **initialize** message, sends a **server-is-idle** message to the master and waits for further instructions—**specialize** messages which it processes, storing the specializations. Hence, a server has at most one active thread at any given time.

Figure 2 illustrates the basic protocol as described above. The dashed lines indicate asynchronous messages. However, the causal relationship between the messages entails what is effectively synchronous communication.

3.2 Localizing Information

The above model works in principle, but is too simplistic for realistic usage. It does not take into account that, on realistic networks, message passing is typically more expensive than computation. Unfortunately, computing a specialization in the basic model always involves one synchronous communication: a pair of a **server-is-idle** message from a server to the memoization master and a **specialize** message back to the server. A server considers itself idle immediately after finishing one specialization. This makes for poor performance.

At worst, the basic model involves sending static configurations back and forth between specialization server and memoization master: the specialization server works on a specialization, registers a static configuration, finishes, sends the **server-is-idle** message, and may get the same static configuration back. Sending a **specialize** message involves transmitting a static configuration, and thus a potentially large static data structure.

A first remedy for this problem is to keep specialization as local as possible on the specialization servers: A specialization server now manages a local pending list and specialization cache. It keeps track of all static configurations encountered by the server and assigns a short, unique local identifier (local id) to each of them. With the **specialize** message, the server informs the memoization master not only of the static configuration, but also of the local id it has assigned to it. Then, whenever a server becomes idle, it can refer to its own pending list for more work. However, it may happen that a different specialization server has encountered an equivalent static configuration in the meantime. Therefore, the server still needs to check back with the memoization master if it should commence work on a given static configuration. Fortunately, this only involves transmitting the (short) local id.

Consequently, the memoization master now understands a new message:

can-server-work-on *local-id* A local identifier *local-id* accompanies the message. The server, when sending this message, waits for a boolean answer specifying whether it should commence work on the static configuration that belongs to the local identifier.

Also, the **register-static-config** message is extended to also carry a local identifier as an additional component.

Hence, a server only sends a **server-is-idle** message when its local pending list becomes empty. Otherwise, it considers entries from the pending list until it encounters one where the memoization master positively acknowledges a **can-server-work-on** message.

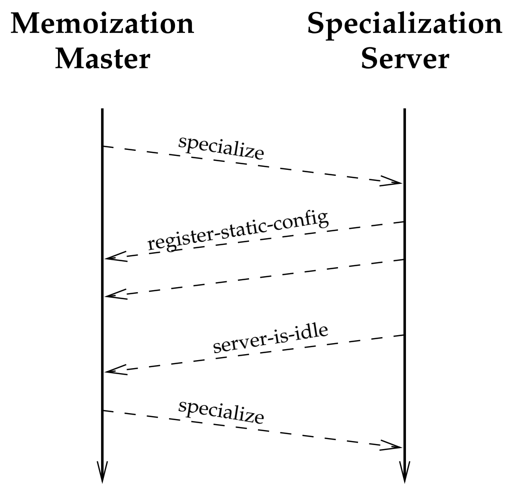

Figure 2: Basic Synchronous Specialization Protocol

3.3 Preempting Work

When the memoization master assigns a static configuration to a server different from the one that generated the configuration, it can inform all servers that have equivalent static configurations in the pending list. This information prevents the overhead of a **can-server-work-on** message later on. Naively, this would involve sending an asynchronous message from the memoization master to the respective servers of the form **kill-local-id** along with a local id.

Unfortunately, this also involves sending one message for each of the local ids to be killed—thus saving only the time difference between a synchronous and an asynchronous message. For a specific specialization server, the information about local ids that other servers have processed becomes only relevant when it sends the (synchronous) **can-server-work-on** message to the master. Thus, it is easy to extend the return value of the **can-server-work-on** message to also include a list of local ids whose static configurations the master has assigned to other servers since the last **can-server-work-on** message. The master merely needs to keep track of these static configurations.

Figure 3 shows the more developed model for distributed partial evaluation. The solid pair of lines describes the synchronous message passing necessary for **can-server-work-on** messages.

Figure 4 illustrates the protocol outlined above with one memoization master and two specialization servers.

3.4 Caching Static Skeletons

Both of the applications mentioned in Sec. 2.2 share the fact that one data structure stays constant throughout the specializations: The generic parser always passes around the grammar; the interpreter needs to keep track of the entire source program. Nevertheless, the current specialization model passes these data structures anew with each **specialize** and each **register-static-config** message.

In compilation, this is especially undesirable as the source program can get large; the problem amounts to retransmitting the source program each time a specialization server registers and receives new work from the memoization master.

Hence, it is necessary to cache the elements of a static skeleton both on the memoization master (to avoid retransmission with **specialize** messages) and the specialization server (to avoid retransmission with **register-static-config** messages). Since these data structures are typically values of top-level variables in the static skeleton, a simple caching mechanism suffices.

3.5 Choosing Work

In order to keep computation as local as possible, the memoization master and the specialization servers keep track of a *preferred* specialization function which each specialization server will exhaust before starting work on other specialization functions. Conversely, the memoization master will tries to avoid assigning static configurations to servers which belong to a specialization function preferred by a a different server.

3.6 Speculative Specialization

An obvious weak point in the above model is the synchronous communication needed until a server can commence work on a specialization. It is possible, however, to also convert this into an asynchronous communication: The server still selects an entry from its local pending list but also sends a message **server-works-on** with that entry and

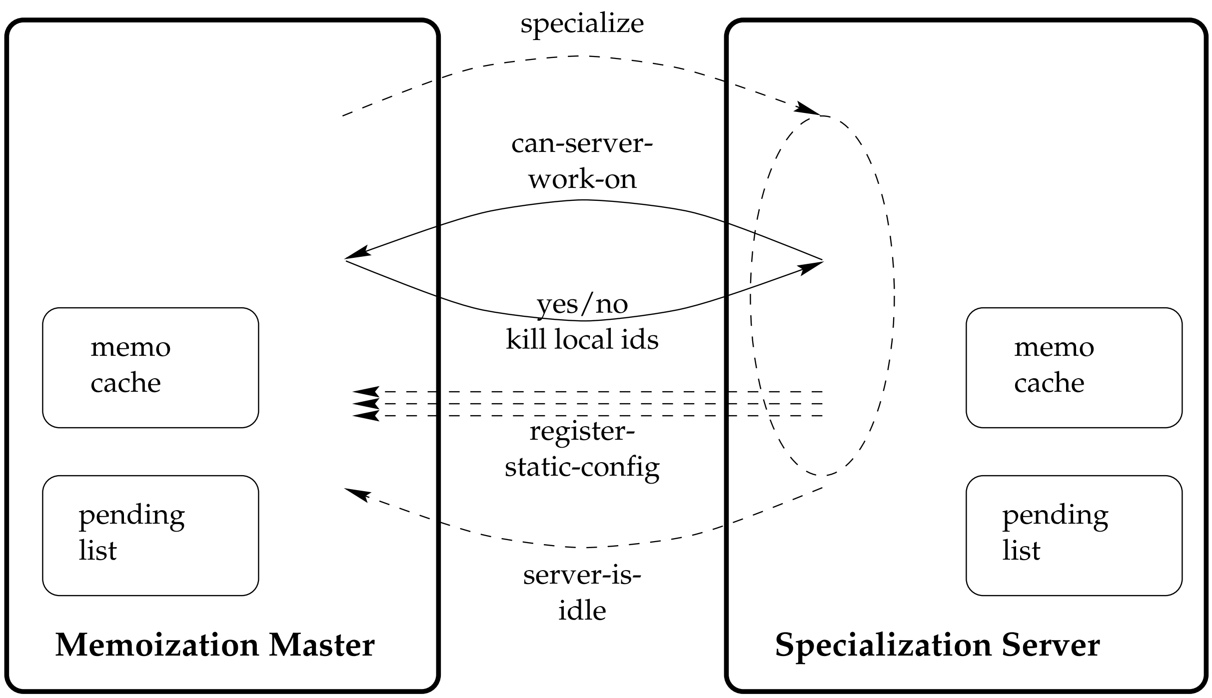

Figure 3: Synchronous model for distributed partial evaluation

the server's id to the master **without waiting for the result**. The master consults the global cache and either removes the entry from the pending list or (if some other client is already working on that particular specialization) it tries to kill the specialization on the specialization server.

In both cases when a specialization has terminated (successfully or due to a kill message) the specialization server continues with the next entry from the pending list or sends **server-is-idle** to the master if the pending list is empty.

Unfortunately, speculative specialization does not yield the expected gains, as our experiments in Sec. 5 show.

3.7 Sizes of Messages

The only sizeable messages are:

- **specialize** specialization requests from the master and
- `register-static-config` static configuration messages from the specialization server to the master.

Both messages involve the transmission of entire static configurations, which may become arbitrarily large. The messages **server-works-on** and **can-server-work-on** only send unique ids, which are established with **register-static-config** messages. The sizes of the large messages are decreased using the techniques outlined in Section 3.4. The result is that large data structures are only transmitted once, afterwards only a globally unique id is transmitted in their place.

3.8 Detecting Termination

The memoization master keeps track of the number of specialization servers that have been started and of the number of specialization servers that are currently idle. As soon as these numbers are equal the computation has been completed.

4 Implementation

Our implementation of the distributed model builds on Kali [6], a distributed implementation of the Scheme programming language. This section gives an overview of Kali's distributed environment, and then briefly describes how the PGG system makes use of it.

4.1 Kali

Kali is an extension of the Scheme 48 system [18], an advanced byte-code implementation of Scheme. Scheme 48 already provides a sequential implementation of preemptive threads.

Kali calls a computational agent in a distributed computing environment an *address space*. Each address space corresponds to a Kali process which may reside anywhere in a network. All address spaces are pairwise connected by TCP/IP stream connections.

Address spaces are first-class objects in Kali and may thus be bound to variables, passed to procedures, and re-

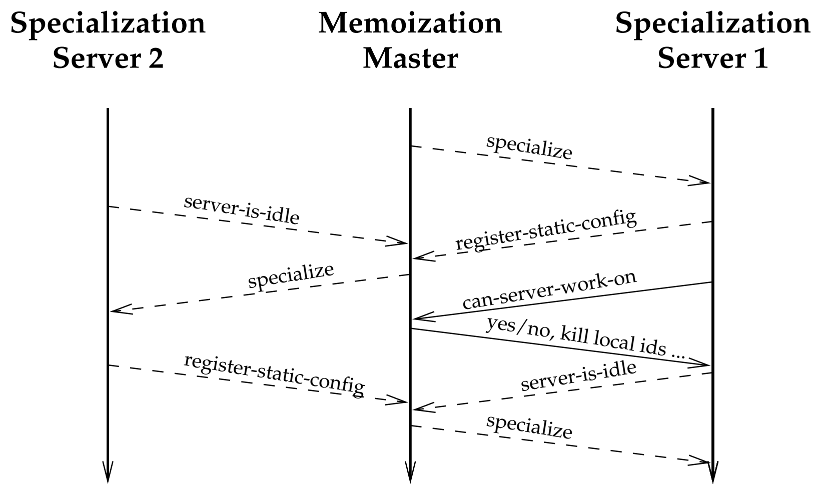

Figure 4: Synchronous Specialization Protocol

turned from procedure calls. One address space may send a message to another address space simply by calling a procedure on the remote address space. Kali provides a `remote-run!` operation which starts a procedure `proc` on arguments `a1` ... `an` on an arbitrary address space `aspace`:

```
(remote-run! aspace proc a1 ... an)
```

The transmission of both the code of the procedure and the values of the arguments is completely transparent. Most values are simply copied to the remote address space. Sharing is respected within a single message, but not across different messages (as it would be the case, e.g., in Linda's tuple space [4,5,14]). Some special values are assigned global unique identifications (uids) and are transferred only once. Among these are

procedures: the compiled code of a procedure is only transmitted once while the environment part (containing the values of the free variables) is transmitted every time.

proxies: a *proxy* is conceptually a distributed array which is indexed by address spaces. A proxy has a local value for each address space, but it also holds information identifying the address space that created the proxy. The procedures `proxy-local-ref` and `proxy-creator` provide access to this information. Proxies are also transmitted using unique uid's.

Furthermore, the thread system provides *placeholders* which serve as semaphores and also the necessary locking primitives to grant exclusive access rights locally. On top of these abstractions, the Kali system provides remote procedure calls with `remote-apply`, thread migration, user-specified load-balancing, and more [6].

4.2 Adapting the PGG

The changes in the PGG system boil down to replacing the serial implementation of memoization [27] by the distributed one outlined above. Due to the modular design of the system (taking advantage of Scheme 48's module system [21]) only the memoization module has to be replaced, everything else remains unchanged.

All messages are simply asynchronous (`remote-run!`) or synchronous (`remote-apply`) procedure calls. For caching static skeletons, proxies provide a straightforward mechanism.

4.3 Implementation Problems

A problematic issue is symbol generation. In the course of each specialization many new identifiers are generated for bound variables in the residual program. The implementation language Scheme uses symbols for variables and the standard solution is to provide a symbol generator that creates new symbols on the fly. However, in the current implementation of Kali, locally created symbols do not have a globally unique identity. Hence, our system replaces symbols by "hand-made" globally unique numbers. The master converts these numbers into symbols upon collecting the residual program.

5 Performance

We have run benchmarks on a cluster of six RS/6000 workstations running AIX connected by an Ethernet local area network. Specifically, we have run an LR(1) parser generator [23] and performed compilation of a large automatically generated Mixwell program [16].

# processors	runtime	CPU time	speedup
seq	24.88	24.48	
1	24.06	6.57	1.0
2	10.88	6.16	2.2
3	10.78	6.61	2.2
4	8.56	7.09	2.8
5	8.17	6.78	2.9
6	8.72	7.39	2.8

Table 1: Parser generation on a cluster of RS/6000 workstations

# processors	runtime	CPU time	speedup
seq	6.19	6.22	
1	7.44	0.49	1.0
2	4.49	1.00	1.7
3	4.12	1.44	1.8
4	3.46	1.77	2.2
5	4.08	1.83	1.8
6	3.82	1.89	1.9

Table 2: Compilation on a cluster of RS/6000 workstations

Tables 1 and 2 show the run times of parser generation and the respective speed-ups. The "CPU time" column shows the CPU time on the memoization master. Note that the tables only show up to six processors—the seventh is the memoization master. It is not clear whether the tables indicate any saturation on the part of the master—more sufficiently similar machines were not available to us. The first line of each table shows the timings for the purely sequential version of the system. Obviously, the initial message overhead of the parallel version is already offset by the work division between the memoization master and the single specialization server in the one-processor case. Note also that the CPU utilization of the master does not ultimately change significantly with the addition of more specialization servers. It does, however, present a lower bound for the run time of the specialization. This is an indication that some optimization on our (currently fairly straightforward) memoization master may yield higher maximum speedups.

All of these benchmarks use the synchronous model. Even though we expected much smaller improvements due to the high costs of synchronous communication, the results here were much better than with speculative specialization, where our current implementation only yields negligible speed-ups. Here, the high communication latencies usually prevent kill messages from the master to reach the server in time to stop any superfluous work done. The time gained by avoiding synchronous communication is offset by the time spent on duplicate work.

6 Related Work

The notion of partial evaluation and its application to automatic program generation stems from Futamura's work [10]. Since then, compiler generation has been among the main fields of interest for researchers in partial evaluation. This led to the discovery of offline partial evaluation and the construction of practical compiler generators [16].

Consel and Danvy [7] have implemented a self-applicable partial evaluator for the purely functional subset of Scheme on a shared memory multi-processor machine. Their implementation exploits features of Mul-T, a dialect of Scheme with futures [11]. More precisely, they assign one dedicated semaphore to each specialization function. Therefore the speedup of their method is limited by the number of specialization functions in the program. However, in a shared-memory machine there is no need to transmit static configurations and to assign unique identifiers as we do.

Our approach to parallelizing the PGG is inspired by the farm-of-workers model [12, 28]. Our implementation benefits fundamentally from Kali's approach to a distributed higher-order language [6].

7 Conclusions and Future Work

We have demonstrated that partial evaluation has some potential for effective parallelization, giving rise to numerous applications. We intend to extend our system in the following directions:

- Recently, one of the authors has developed a sequential implementation of incremental specialization and specialization on demand [27]. This implementation has an intrinsic potential for parallelism: whereas the sequential implementation interleaves specialization with running the specialized program, a parallel implementation could continue specialization during execution of the specialized program. A combination with run-time code generation which is also already part of the PGG system [25] can lead to just-in-time compilation.

- We believe that one of the limiting factors is the lack of globally shared data structures in the Kali Scheme system. This lack gives rise to a large communication overhead if the specializer deals with large data that changes during specialization. It would be interesting to perform similar experiments with a system like TS/Scheme [14].

- It is not clear whether our strategy is also suited to shared-memory multi-processors. We would like to conduct a comparison between our method and the method proposed by Consel and Danvy [7].

References

[1] Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In *(To be presented at the PLILP 94 conference)*. Springer-Verlag, September 1994.

[2] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), 1984.

[3] Andrew Black, Charles Consel, Calton Pu, Jonathan Walpole, Crispin Cowan, Tito Autrey, Jon Inouye, Lakshmi Kethana, and Ke Zhang. Dream and reality: Incremental specialization in a commercial operating system. Technical report, Dept. of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, March 1995.

[4] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, apr 1989.

[5] Nicholas Carriero and David Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[6] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, 17(5), September 1995.

[7] Charles Consel and Olivier Danvy. Partial evaluation in parallel. *LISP and Symbolic Computation*, 5(4):315–330, 1993.

[8] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Symposium on Principles of Programming Languages '93*, pages 493–501, Charleston, January 1993. ACM.

[9] H. Dybkjær. Parsers and partial evaluation: An experiment. Technical Report Student Project 85-7-15, DIKU, University of Copenhagen, July 1985.

[10] Yoshihiko Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

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

[12] Anthony J. G. Hey. Experiments in MIMD parallelism. In *PARLE'89 Parallel Architectures and Languages Europa II*, volume 366 of *Lecture Notes in Computer Science*, pages 28–41. Springer-Verlag, 1989.

[13] IEEE. Standard for the Scheme programming language. Technical Report 1178-1990, Institute of Electrical and Electronic Engineers, Inc., New York, 1991.

[14] Suresh Jagannathan. TS/Scheme: Distributed data structures in Lisp. *Lisp and Symbolic Computation*, 7:283–305, 1994.

[15] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[16] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 124–140, Dijon, France, 1985. Springer-Verlag. LNCS 202.

[17] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico*, pages 258–268. ACM, ACM, January 1992.

[18] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[19] R. Leermakers. *The Functional Treatment of Parsing*. Kluwer Academic Publishers, Boston, 1993.

[20] Christian Mossin. Partial evaluation of general parsers. In *Symp. Partial Evaluation and Semantics-Based Program Manipulation '93*, pages 13–21, Copenhagen, Denmark, June 1993. ACM.

[21] Jonathan A. Rees. Another module system for scheme. Part of the Scheme 48 distribution, January 1994.

[22] Revised^4 report on the algorithmic language Scheme. *Lisp Pointers*, IV(3):1–55, July–September 1991.

[23] Michael Sperber and Peter Thiemann. The essence of LR parsing. In William Scherlis, editor, *ACM SIGPLAN Symp. Partial Evaluation and Semantics-Based Program Manipulation '95*, pages 146–155, La Jolla, CA, June 1995. ACM Press.

[24] Michael Sperber and Peter Thiemann. Realistic compilation by partial evaluation. In *Conference on Programming Language Design and Implementation '96*, pages 206–214, Philadelphia, May 1996. ACM. SIGPLAN Notices, 31(5).

[25] Michael Sperber and Peter Thiemann. Two for the price of one: Composing partial evaluation and compilation. (submitted), 1997.

[26] Peter Thiemann. Cogen in six lines. In *International Conference on Functional Programming '96*, pages 180–189, Philadelphia, May 1996. ACM.

[27] Peter Thiemann. Implementing memoization for partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *Programming Language Implementation and Logic Programming (PLILP '96)*, volume 1140 of *Lecture Notes in Computer Science*, pages 198–212, Aachen, Germany, September 1996. Springer-Verlag.

[28] R. W. S. Tregidgo and A. C. Downton. Processor farm analysis and simulation for embedded parallel processing systems. In Stephen J. Turner, editor, *Tools and Techniques for Transputer Applications. Proceedings of the 12th Occam User Group Technical Meeting*, pages 179–189. IOS-Press, 1990.